Wolfram *Mathematica*® Tutorial Collection

ADVANCED NUMERICAL DIFFERENTIAL EQUATION SOLVING IN *MATHEMATICA*

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Content authored by:
Mark Sofroniou and Rob Knapp

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

Contents

Introduction to Advanced Numerical Differential Equation Solving in *Mathematica*

Overview

The *Mathematica* function `NDSolve` is a general numerical differential equation solver. It can handle a wide range of *ordinary differential equations* (ODEs) as well as some *partial differential equations* (PDEs). In a system of ordinary differential equations there can be any number of unknown functions x_i, but all of these functions must depend on a single "independent variable" t, which is the same for each function. Partial differential equations involve two or more independent variables. `NDSolve` can also solve some *differential-algebraic equations* (DAEs), which are typically a mix of differential and algebraic equations.

`NDSolve[{`eqn_1`,`eqn_2`,…},`u`,{`t`,`t_{min}`,`t_{max}`}]`	find a numerical solution for the function u with t in the range t_{min} to t_{max}
`NDSolve[{`eqn_1`,`eqn_2`,…},{`u_1`,`u_2`,…},{`t`,`t_{min}`,`t_{max}`}]`	find numerical solutions for several functions u_i

Finding numerical solutions to ordinary differential equations.

`NDSolve` represents solutions for the functions x_i as `InterpolatingFunction` objects. The `InterpolatingFunction` objects provide approximations to the x_i over the range of values t_{min} to t_{max} for the independent variable t.

In general, `NDSolve` finds solutions iteratively. It starts at a particular value of t, then takes a sequence of steps, trying eventually to cover the whole range t_{min} to t_{max}.

In order to get started, `NDSolve` has to be given appropriate initial or boundary conditions for the x_i and their derivatives. These conditions specify values for $x_i[t]$, and perhaps derivatives $x_i'[t]$, at particular points t. When there is only one t at which conditions are given, the equations and initial conditions are collectively referred to as an initial value problem. A boundary value occurs when there are multiple points t. `NDSolve` can solve nearly all initial value problems that can symbolically be put in normal form (i.e. are solvable for the highest derivative order), but only linear boundary value problems.

This finds a solution for x with t in the range 0 to 2, using an initial condition for x at $t == 1$.

```
In[1]:= NDSolve[{x'[t] == x[t], x[1] == 3}, x, {t, 0, 2}]
Out[1]= {{x → InterpolatingFunction[{{0., 2.}}, <>]}}
```

When you use `NDSolve`, the initial or boundary conditions you give must be sufficient to determine the solutions for the x_i completely. When you use `DSolve` to find symbolic solutions to differential equations, you may specify fewer conditions. The reason is that `DSolve` automatically inserts arbitrary symbolic constants `C[i]` to represent degrees of freedom associated with initial conditions that you have not specified explicitly. Since `NDSolve` must give a numerical solution, it cannot represent these kinds of additional degrees of freedom. As a result, you must explicitly give all the initial or boundary conditions that are needed to determine the solution.

In a typical case, if you have differential equations with up to n^{th} derivatives, then you need to either give initial conditions for up to $(n-1)^{\text{th}}$ derivatives, or give boundary conditions at n points.

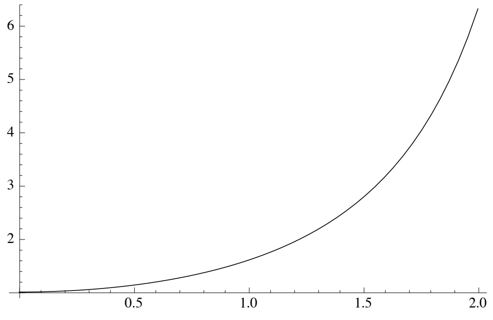
This solves an initial value problem for a second-order equation, which requires two conditions, and are given at $t == 0$.

```
In[2]:= NDSolve[{x''[t] == x[t]^2, x[0] == 1, x'[0] == 0}, x, {t, 0, 2}]
Out[2]= {{x → InterpolatingFunction[{{0., 2.}}, <>]}}
```

This plots the solution obtained.

```
In[3]:= Plot[Evaluate[x[t] /. %], {t, 0, 2}]
```

Out[3]=



Here is a simple boundary value problem.

```
In[4]:= NDSolve[{y''[x] + x y[x] == 0, y[0] == 1, y[1] == -1}, y, {x, 0, 1}]
Out[4]= {{y → InterpolatingFunction[{{0., 1.}}, <>]}}
```

You can use `NDSolve` to solve systems of coupled differential equations as long as each variable has the appropriate number of conditions.

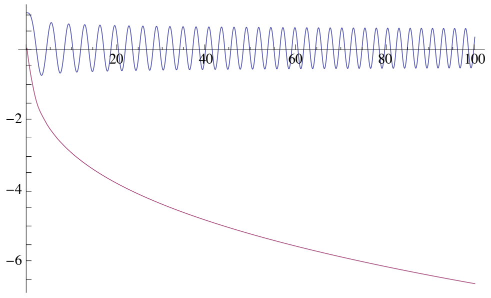
This finds a numerical solution to a pair of coupled equations.

```
In[5]:= sol = NDSolve[{x''[t] == y[t] x[t], y'[t] == -1/(x[t]^2 + y[t]^2),
            x[0] == 1, x'[0] == 0, y[0] == 0}, {x, y}, {t, 0, 100}]
Out[5]= {{x → InterpolatingFunction[{{0., 100.}}, <>], y → InterpolatingFunction[{{0., 100.}}, <>]}}
```

Here is a plot of both solutions.

```
In[6]:= Plot[Evaluate[{x[t], y[t]} /. %], {t, 0, 100}, PlotRange → All, PlotPoints → 200]
```

Out[6]=



You can give initial conditions as equations of any kind. If these equations have multiple solutions, `NDSolve` will generate multiple solutions.

The initial conditions in this case lead to multiple solutions.

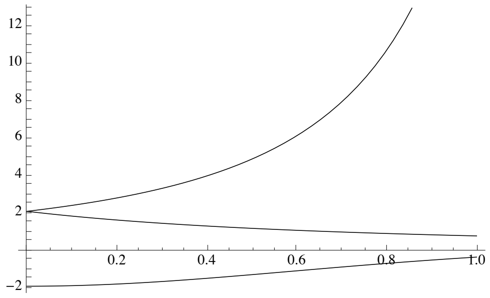
```
In[7]:= NDSolve[{y'[x]^2 - y[x]^3 == 0, y[0]^2 == 4}, y, {x, 1}]

    NDSolve::mxst:
      Maximum number of 10000 steps reached at the point x == 1.1160976563722613`*^-8. ≫

Out[7]= {{y → InterpolatingFunction[{{0., 1.}}, <>]}, {y → InterpolatingFunction[{{0., 1.}}, <>]},
         {y → InterpolatingFunction[{{0., 1.1161×10^-8}}, <>]},
         {y → InterpolatingFunction[{{0., 1.}}, <>]}}
```

`NDSolve` was not able to find the solution for `y'[x] == -Sqrt[y[x]^3], y[0] == -2` because of problems with the branch cut in the square root function.

This shows the real part of the solutions that `NDSolve` was able to find. (The upper two solutions are strictly real.)

```
In[8]:= Plot[Evaluate[Part[Re[y[x] /. %], {1, 2, 4}]], {x, 0, 1}]
```

Out[8]=



`NDSolve` can solve a mixed system of differential and algebraic equations, referred to as *differential-algebraic equations* (DAEs). In fact, the example given is a sort of DAE, where the equations are not expressed explicitly in terms of the derivatives. Typically, however, in DAEs, you are not able to solve for the derivatives at all and the problem must be solved using a different method entirely.

Here is a simple DAE.

```
In[9]:= NDSolve[{x''[t] + y[t] == x[t],
          x[t]^2 + y[t]^2 == 1, x[0] == 0, x'[0] == 1}, {x, y}, {t, 0, 2}]

          NDSolve::ndsz :
            At t == 1.6656481721762058`, step size is effectively zero; singularity or stiff system suspected. ≫

Out[9]= {{x → InterpolatingFunction[{{0., 1.66565}}, <>], y → InterpolatingFunction[{{0., 1.66565}}, <>]}}
```

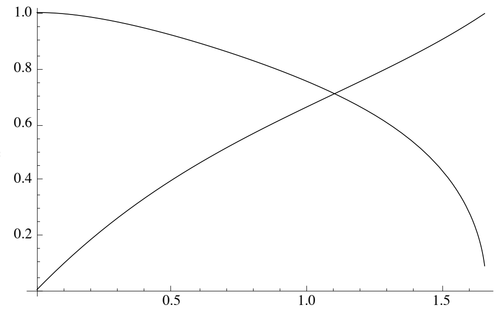
Note that while both of the equations have derivative terms, the variable y appears without any derivatives, so `NDSolve` issues a warning message. When the usual substitution to convert to first-order equations is made, one of the equations does indeed become effectively algebraic.

Also, since y only appears algebraically, it is not necessary to give an initial condition to determine its values. Finding initial conditions that are consistent with DAEs can, in fact, be quite difficult. The tutorial "Numerical Solution of Differential-Algebraic Equations" has more information.

This shows a plot of the solutions.

```
In[10]:= Plot[Evaluate[{x[t], y[t]} /. %], {t, 0, 1.66}]
```

Out[10]=



From the plot, you can see that the derivative of y is tending to vary arbitrarily fast. Even though it does not explicitly appear in the equations, this condition means that the solver cannot continue further.

Unknown functions in differential equations do not necessarily have to be represented by single symbols. If you have a large number of unknown functions, for example, you will often find it more convenient to give the functions names like $x[i]$ or x_i.

This constructs a set of twenty-five coupled differential equations and initial conditions and solves them.

```
In[11]:= n = 25;
         x0[t_] := 0;
         xn[t_] := 1;
         eqns =
           Table[{xi'[t] == n^2 (xi+1[t] - 2 xi[t] + xi-1[t]), xi[0] == (i / n)^10}, {i, n - 1}];
         vars = Table[xi[t], {i, n - 1}];
         NDSolve[eqns, vars, {t, 0, .25}]
```

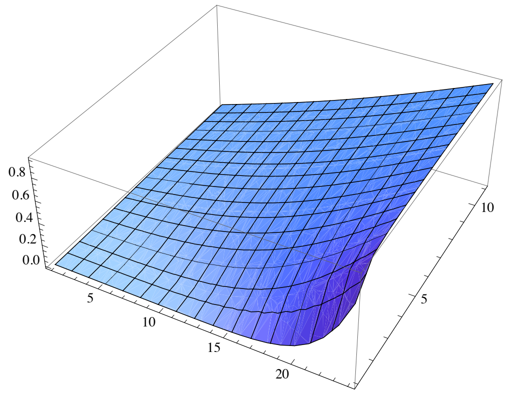
```
Out[16]= {{x1[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x2[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x3[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x4[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x5[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x6[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x7[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x8[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x9[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x10[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x11[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x12[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x13[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x14[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x15[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x16[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x17[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x18[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x19[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x20[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x21[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x22[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x23[t] → InterpolatingFunction[{{0., 0.25}}, <>][t],
  x24[t] → InterpolatingFunction[{{0., 0.25}}, <>][t]}}
```

This actually computes an approximate solution of the heat equation for a rod with constant temperatures at either end of the rod. (For more accurate solutions, you can increase n.)

The result is an approximate solution to the heat equation for a 1-dimensional rod of length 1 with constant temperature maintained at either end. This shows the solutions considered as spatial values as a function of time.

```
In[17]:= ListPlot3D[Table[vars /. First[%], {t, 0, .25, .025}]]
```

Out[17]=



An unknown function can also be specified to have a vector (or matrix) value. The dimensionality of an unknown function is taken from its initial condition. You can mix scalar and vector unknown functions as long as the equations have consistent dimensionality according to the rules of *Mathematica* arithmetic. The `InterpolatingFunction` result will give values with the same dimensionality as the unknown function. Using nonscalar variables is very convenient when a system of differential equations is governed by a process that may be difficult or inefficient to express symbolically.

This uses a vector valued unknown function to solve the same system as earlier.

```
In[18]:= f[x_?VectorQ] := n^2 * ListConvolve[{1, -2, 1}, x, {2, 2}, {1, 0}];
         NDSolve[{X'[t] == f[X[t]], X[0] == (Range[n - 1] / n)^10}, X, {t, 0, .25}]
Out[19]= {{X → InterpolatingFunction[{{0., 0.25}}, <>]}}
```

`NDSolve` is able to solve some partial differential equations directly when you specify more independent variables.

`NDSolve[{`eqn_1`,`eqn_2`,…},`u`,{`t`,`t_{min}`,`t_{max}`},{`x`,`x_{min}`,`x_{max}`},…]`	solve a system of partial differential equations for a function $u[t, x …]$ with t in the range t_{min} to t_{max} and x in the range x_{min} to x_{max}, …
`NDSolve[{`eqn_1`,`eqn_2`,…},{`u_1`,`u_2`,…},{`t`,`t_{min}`,`t_{max}`},{`x`,`x_{min}`,`x_{max}`},…]`	solve a system of partial differential equations for several functions u_i

Finding numerical solutions to partial differential equations.

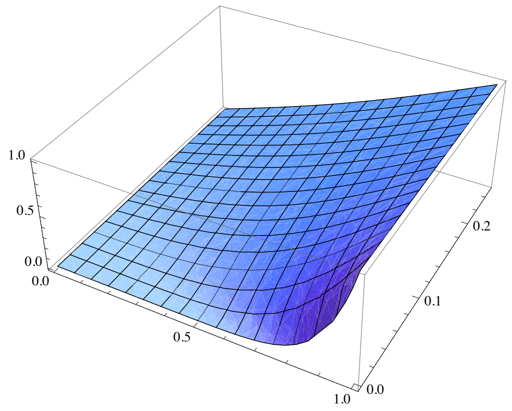
Here is a solution of the heat equation found directly by `NDSolve`.

```
In[20]:= NDSolve[{D[u[x, t], t] == D[u[x, t], x, x], u[x, 0] == x^10,
           u[0, t] == 0, u[1, t] == 1}, u, {x, 0, 1}, {t, 0, .25}]
Out[20]= {{u → InterpolatingFunction[{{0., 1.}, {0., 0.25}}, <>]}}
```

Here is a plot of the solution.

```
In[21]:= Plot3D[Evaluate[First[u[x, t] /. %]], {x, 0, 1}, {t, 0, .25}]
```

Out[21]=



`NDSolve` currently uses the numerical method of lines to compute solutions to partial differential equations. The method is restricted to problems that can be posed with an initial condition in at least one independent variable. For example, the method cannot solve elliptic PDEs such as Laplace's equation because these require boundary values. For the problems it does solve, the method of lines is quite general, handling systems of PDEs or nonlinearity well, and often quite fast. Details of the method are given in "Numerical Solution of Partial Differential Equations".

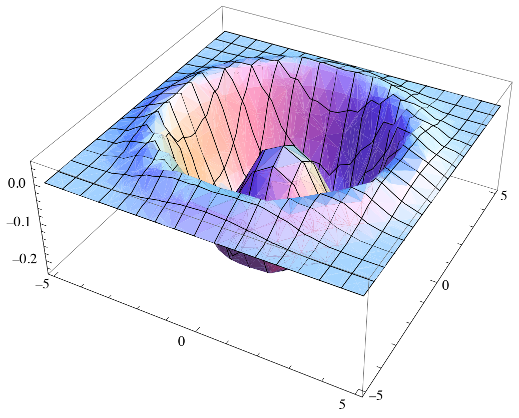
This finds a numerical solution to a generalization of the nonlinear sine-Gordon equation to two spatial dimensions with periodic boundary conditions.

```
In[22]:= 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, -5, y] == u[t, 5, y] == 0, u[t, x, -5] == u[t, x, 5] == 0},
            u, {t, 0, 3}, {x, -5, 5}, {y, -5, 5}]
Out[22]= {{u → InterpolatingFunction[{{0., 3.}, {-5., 5.}, {-5., 5.}}, <>]}}
```

Here is a plot of the result at $t == 3$.

```
In[23]:= Plot3D[First[u[3, x, y] /. %], {x, -5, 5}, {y, -5, 5}]
```

Out[23]=



As mentioned earlier, NDSolve works by taking a sequence of steps in the independent variable t. NDSolve uses an adaptive procedure to determine the size of these steps. In general, if the solution appears to be varying rapidly in a particular region, then NDSolve will reduce the step size to be able to better track the solution.

NDSolve allows you to specify the precision or accuracy of result you want. In general, NDSolve makes the steps it takes smaller and smaller until the solution reached satisfies either the AccuracyGoal *or* the PrecisionGoal you give. The setting for AccuracyGoal effectively determines the absolute error to allow in the solution, while the setting for PrecisionGoal determines the relative error. If you need to track a solution whose value comes close to zero, then you will typically need to increase the setting for AccuracyGoal. By setting AccuracyGoal -> Infinity, you tell NDSolve to use PrecisionGoal only. Generally, AccuracyGoal and PrecisionGoal are used to control the error local to a particular time step. For some differential equations, this error can accumulate, so it is possible that the precision or accuracy of the result at the end of the time interval may be much less than what you might expect from the settings of AccuracyGoal and PrecisionGoal.

`NDSolve` uses the setting you give for `WorkingPrecision` to determine the precision to use in its internal computations. If you specify large values for `AccuracyGoal` or `PrecisionGoal`, then you typically need to give a somewhat larger value for `WorkingPrecision`. With the default setting of `Automatic`, both `AccuracyGoal` and `PrecisionGoal` are equal to half of the setting for `WorkingPrecision`.

`NDSolve` uses error estimates for determining whether it is meeting the specified tolerances. When working with systems of equations, it uses the setting of the option `NormFunction -> ` f to combine errors in different components. The norm is scaled in terms of the tolerances, given so that `NDSolve` tries to take steps such that

$$f\left[\left\{\frac{\mathrm{err}_1}{\mathrm{tol}_r\,\mathrm{Abs}[x_1]+\mathrm{tol}_a},\ \frac{\mathrm{err}_2}{\mathrm{tol}_r\,\mathrm{Abs}[x_2]+\mathrm{tol}_a},\ \ldots\right\}\right]\leq 1$$

where err_i is the i^{th} component of the error and x_i is the i^{th}component of the current solution.

This generates a high-precision solution to a differential equation.

```
In[24]:= NDSolve[{x'''[t] == x[t], x[0] == 1, x'[0] == x''[0] == 0}, x, {t, 1},
          AccuracyGoal -> 20, PrecisionGoal -> 20, WorkingPrecision -> 25]
Out[24]= {{x → InterpolatingFunction[{{0, 1.000000000000000000000000}}, <>]}}
```

Here is the value of the solution at the endpoint.

```
In[25]:= x[1] /. %
Out[25]= {1.168058313375918525580620}
```

Through its adaptive procedure, `NDSolve` is able to solve "stiff" differential equations in which there are several components varying with t at extremely different rates.

`NDSolve` follows the general procedure of reducing step size until it tracks solutions accurately. There is a problem, however, when the true solution has a singularity. In this case, `NDSolve` might go on reducing the step size forever, and never terminate. To avoid this problem, the option `MaxSteps` specifies the maximum number of steps that `NDSolve` will ever take in attempting to find a solution. For ordinary differential equations, the default setting is `MaxSteps -> 10 000`.

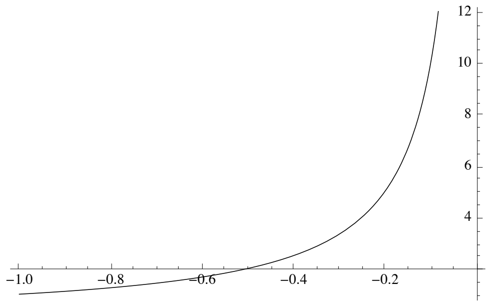
NDSolve stops after taking 10,000 steps.

```
In[26]:= NDSolve[{y'[x] == 1 / x^2, y[-1] == 1}, y[x], {x, -1, 0}]

    NDSolve::mxst : Maximum number of 10000 steps reached at the point x == -1.00413×10^-172. >>

Out[26]= {{y[x] → InterpolatingFunction[{{-1., -1.00413×10^-172}}, <>][x]}}
```

There is in fact a singularity in the solution at x = 0.

```
In[27]:= Plot[Evaluate[y[x] /. %], {x, -1, 0}]
```

Out[27]=



The default setting `MaxSteps -> 10 000` should be sufficient for most equations with smooth solutions. When solutions have a complicated structure, however, you may sometimes have to choose larger settings for `MaxSteps`. With the setting `MaxSteps -> Infinity` there is no upper limit on the number of steps used.

`NDSolve` has several different methods built in for computing solutions as well as a mechanism for adding additional methods. With the default setting `Method -> Automatic`, `NDSolve` will choose a method which should be appropriate for the differential equations. For example, if the equations have stiffness, implicit methods will be used as needed, or if the equations make a DAE, a special DAE method will be used. In general, it is not possible to determine the nature of solutions to differential equations without actually solving them: thus, the default `Automatic` methods are good for solving as wide variety of problems, but the one chosen may not be the best one available for your particular problem. Also, you may want to choose methods, such as symplectic integrators, which preserve certain properties of the solution.

Choosing an appropriate method for a particular system can be quite difficult. To complicate it further, many methods have their own settings, which can greatly affect solution efficiency and accuracy. Much of this documentation consists of descriptions of methods to give you an idea of when they should be used and how to adjust them to solve particular problems. Furthermore, `NDSolve` has a mechanism that allows you to define your own methods and still have the equations and results processed by `NDSolve` just as for the built-in methods.

When `NDSolve` computes a solution, there are typically three phases. First, the equations are processed, usually into a function that represents the right-hand side of the equations in normal form. Next, the function is used to iterate the solution from the initial conditions. Finally, data saved during the iteration procedure is processed into one or more `InterpolatingFunction` objects. Using functions in the `` NDSolve` `` context, you can run these steps separately and, more importantly, have more control over the iteration process. The steps are tied by an `` NDSolve`StateData `` object, which keeps all of the data necessary for solving the differential equations.

The Design of the NDSolve Framework

Features

Supporting a large number of numerical integration methods for differential equations is a lot of work.

In order to cut down on maintenance and duplication of code, common components are shared between methods.

This approach also allows code optimization to be carried out in just a few central routines.

The principal features of the `NDSolve` framework are:

- Uniform design and interface
- Code reuse (common code base)
- Objection orientation (method property specification and communication)
- Data hiding
- Separation of method initialization phase and run-time computation
- Hierarchical and reentrant numerical methods
- Uniform treatment of rounding errors (see [HLW02], [SS03] and the references therein)
- Vectorized framework based on a generalization of the BLAS model [LAPACK99] using optimized in-place arithmetic

- Tensor framework that allows families of methods to share one implementation
- Type and precision dynamic for all methods
- Plug-in capabilities that allow user extensibility and prototyping
- Specialized data structures

Common Time Stepping

A common time-stepping mechanism is used for all one-step methods. The routine handles a number of different criteria including:

- Step sizes in a numerical integration do not become too small in value, which may happen in solving stiff systems
- Step sizes do not change sign unexpectedly, which may be a consequence of user programming error
- Step sizes are not increased after a step rejection
- Step sizes are not decreased drastically toward the end of an integration
- Specified (or detected) singularities are handled by restarting the integration
- Divergence of iterations in implicit methods (e.g. using fixed, large step sizes)
- Unrecoverable integration errors (e.g. numerical exceptions)
- Rounding error feedback (compensated summation) is particularly advantageous for high-order methods or methods that conserve specific quantities during the numerical integration

Data Encapsulation

Each method has its own data object that contains information that is needed for the invocation of the method. This includes, but is not limited to, coefficients, workspaces, step-size control parameters, step-size acceptance/rejection information, and Jacobian matrices. This is a generalization of the ideas used in codes like LSODA ([H83], [P83]).

Method Hierarchy

Methods are reentrant and hierarchical, meaning that one method can call another. This is a generalization of the ideas used in the Generic ODE Solving System, Godess (see [O95], [O98] and the references therein), which is implemented in C++.

Initial Design

The original method framework design allowed a number of methods to be invoked in the solver.

$$\boxed{\texttt{NDSolve}} \longrightarrow \boxed{\texttt{“ExplicitRungeKutta“}}$$

$$\boxed{\texttt{NDSolve}} \longrightarrow \boxed{\texttt{“ImplicitRungeKutta“}}$$

First Revision

This was later extended to allow one method to call another in a sequential fashion, with an arbitrary number of levels of nesting.

$$\boxed{\texttt{NDSolve}} \longrightarrow \boxed{\texttt{“Extrapolation“}} \longrightarrow \boxed{\texttt{“ExplicitMidpoint“}}$$

The construction of compound integration methods is particularly useful in geometric numerical integration.

$$\boxed{\texttt{NDSolve}} \longrightarrow \boxed{\texttt{“Projection“}} \longrightarrow \boxed{\texttt{“ExplicitRungeKutta“}}$$

Second Revision

A more general tree invocation process was required to implement composition methods.

$$\begin{tikzcd}
 & & \boxed{\texttt{“ExplicitEuler“}} \\
 & \vdots & \vdots \\
\boxed{\texttt{NDSolve}} \arrow[r] & \boxed{\texttt{“Composition“}} \arrow[r] \arrow[uur] \arrow[ddr] & \boxed{\texttt{“ImplicitEuler“}} \\
 & \vdots & \vdots \\
 & & \boxed{\texttt{“ExplicitEuler“}}
\end{tikzcd}$$

This is an example of a method composed with its adjoint.

Current State

The tree invocation process was extended to allow for a subfield to be solved by each method, instead of the entire vector field.

This example turns up in the ABC Flow subsection of "Composition and Splitting Methods for NDSolve".

$$\begin{tikzcd}
 & & \boxed{\texttt{“LocallyExact“}\ f_1} \\
\boxed{\texttt{NDSolve}} \arrow[r] & \boxed{\texttt{“Splitting“}\ f = f_1 + f_2} \arrow[ur] \arrow[r] \arrow[dr] & \boxed{\texttt{“ImplicitMidpoint“}\ f_2} \\
 & & \boxed{\texttt{“LocallyExact“}\ f_1}
\end{tikzcd}$$

User Extensibility

Built-in methods can be used as building blocks for the efficient construction of special-purpose (compound) integrators. User-defined methods can also be added.

Method Classes

Methods such as `“ExplicitRungeKutta“` include a number of schemes of different orders. Moreover, alternative coefficient choices can be specified by the user. This is a generalization of the ideas found in RKSUITE [BGS93].

Automatic Selection and User Controllability

The framework provides automatic step-size selection and method-order selection. Methods are user-configurable via method options.

For example a user can select the class of `“ExplicitRungeKutta“` methods, and the code will automatically attempt to ascertain the "optimal" order according to the problem, the relative and absolute local error tolerances, and the initial step-size estimate.

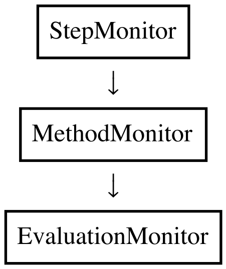
Here is a list of options appropriate for "ExplicitRungeKutta".

```
In[1]:= Options[NDSolve`ExplicitRungeKutta]

Out[1]= {Coefficients → EmbeddedExplicitRungeKuttaCoefficients, DifferenceOrder → Automatic,
         EmbeddedDifferenceOrder → Automatic, StepSizeControlParameters → Automatic,
         StepSizeRatioBounds → {1/8, 4}, StepSizeSafetyFactors → Automatic, StiffnessTest → Automatic}
```

MethodMonitor

In order to illustrate the low-level behaviour of some methods, such as stiffness switching or order variation that occurs at run time , a new `"MethodMonitor"` has been added.

This fits between the relatively coarse resolution of `"StepMonitor"` and the fine resolution of `"EvaluationMonitor"` .



This feature is not officially documented and the functionality may change in future versions.

Shared Features

These features are not necessarily restricted to `NDSolve` since they can also be used for other types of numerical methods.

- Function evaluation is performed using a `NumericalFunction` that dynamically changes type as needed, such as when IEEE floating-point overflow or underflow occurs. It also calls *Mathematica*'s compiler `Compile` for efficiency when appropriate.
- Jacobian evaluation uses symbolic differentiation or finite difference approximations, including automatic or user-specifiable sparsity detection.
- Dense linear algebra is based on LAPACK, and sparse linear algebra uses special-purpose packages such as UMFPACK.

- Common subexpressions in the numerical evaluation of the function representing a differential system are detected and collected to avoid repeated work.
- Other supporting functionality that has been implemented is described in "Norms in NDSolve".

This system dynamically switches type from real to complex during the numerical integration, automatically recompiling as needed.

```
In[2]:= y[1 / 2] /. NDSolve[{y'[t] == Sqrt[y[t]] - 1, y[0] == 1 / 10},
          y, {t, 0, 1}, Method → "ExplicitRungeKutta"]
Out[2]= {-0.349043 + 0.150441 ⅈ}
```

Some Basic Methods

order	method	formula
1	Explicit Euler	$y_{n+1} = y_n + h_n f(t_n, y_n)$
2	Explicit Midpoint	$y_{n+1/2} = y_n + \frac{h_n}{2} f(t_n, y_n)$ $y_{n+1} = y_n + h_n f(t_{n+1/2}, y_{n+1/2})$
1	Backward or Implicit Euler (1-stage RadauIIA)	$y_{n+1} = y_n + h_n f(t_{n+1}, y_{n+1})$
2	Implicit Midpoint (1-stage Gauss)	$y_{n+1} = y_n + h_n f\left(t_{n+1/2}, \frac{1}{2}(y_{n+1} + y_n)\right)$
2	Trapezoidal (2-stage Lobatto IIIA)	$y_{n+1} = y_n + \frac{h_n}{2}(f(t_n, y_n) + f(t_{n+1}, y_{n+1}))$
1	Linearly Implicit Euler	$(I - h_n J)(y_{n+1} - y_n) = h_n f(t_n, y_n)$
2	Linearly Implicit Midpoint	$\left(I - \frac{h_n}{2} J\right)(y_{n+1/2} - y_n) = \frac{h_n}{2} f(t_n, y_n)$ $\left(I - \frac{h_n}{2} J\right) \frac{(\Delta y_n - \Delta y_{n-1/2})}{2} = \frac{h_n}{2} f(t_{n+1/2}, y_{n+1/2}) - \Delta y_{n-1/2}$

Some of the one-step methods that have been implemented.

Here $\Delta y_n = y_{n+1} - y_{n+1/2}$, I denotes the identity matrix, and J denotes the Jacobian matrix $\frac{\partial f}{\partial y}(t_n, y_n)$.

Although the implicit midpoint method has not been implemented as a separate method, it is available through the one-stage Gauss scheme of the `"ImplicitRungeKutta"` method.

ODE Integration Methods

Methods

"ExplicitRungeKutta" Method for NDSolve

Introduction

This loads packages containing some test problems and utility functions.

```
In[3]:= Needs[“DifferentialEquations`NDSolveProblems`“];
        Needs[“DifferentialEquations`NDSolveUtilities`“];
```

Euler's Method

One of the first and simplest methods for solving initial value problems was proposed by Euler:

$$y_{n+1} = y_n + h\, f(t_n,\, y_n). \tag{1}$$

Euler's method is not very accurate.

Local accuracy is measured by how high terms are matched with the Taylor expansion of the solution. Euler's method is first-order accurate, so that errors occur one order higher starting at powers of h^2.

Euler's method is implemented in `NDSolve` as “`ExplicitEuler`“.

```
In[5]:= NDSolve[{y‘[t] == -y[t], y[0] == 1}, y[t], {t, 0, 1},
         Method → “ExplicitEuler“, “StartingStepSize“ → 1 / 10]
Out[5]= {{y[t] → InterpolatingFunction[{{0., 1.}}, <>][t]}}
```

Generalizing Euler's Method

The idea of Runge-Kutta methods is to take successive (weighted) Euler steps to approximate a Taylor series. In this way function evaluations (and not derivatives) are used.

For example, consider the one-step formulation of the *midpoint method*.

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f\left(t_n + \tfrac{1}{2} h, y_n + \tfrac{1}{2} h k_1\right) \\ y_{n+1} &= y_n + h k_2 \end{aligned} \tag{1}$$

The midpoint method can be shown to have a local error of $O\left(h^3\right)$, so it is second-order accurate.

The midpoint method is implemented in `NDSolve` as `"ExplicitMidpoint"`.

```
In[6]:= NDSolve[{y'[t] == -y[t], y[0] == 1}, y[t], {t, 0, 1},
         Method → "ExplicitMidpoint", "StartingStepSize" → 1 / 10]
Out[6]= {{y[t] → InterpolatingFunction[{{0., 1.}}, <>][t]}}
```

Runge-Kutta Methods

Extending the approach in (1), repeated function evaluation can be used to obtain higher order methods.

Denote the Runge-Kutta method for the approximate solution to an initial value problem at $t_{n+1} = t_n + h$, by:

$$\begin{aligned} g_i &= y_n + h \textstyle\sum_{j=1}^{s} a_{i,j} k_j, \\ k_i &= f(t_n + c_i h, g_i), \qquad i = 1, 2, \ldots, s, \\ y_{n+1} &= y_n + h \textstyle\sum_{i=1}^{s} b_i k_i \end{aligned} \tag{1}$$

where s is the number of stages.

It is generally assumed that the *row-sum conditions* hold:

$$c_i = \textstyle\sum_{i=1}^{s} a_{i,j} \tag{2}$$

These conditions effectively determine the points in time at which the function is sampled and are a particularly useful device in the derivation of high-order Runge-Kutta methods.

The *coefficients of the method* are *free parameters* that are chosen to satisfy a Taylor series expansion through some order in the time step h. In practice other conditions such as stability can also constrain the coefficients.

Explicit Runge-Kutta methods are a special case where the matrix A is strictly lower triangular:

$$a_{i,j} = 0, \; j \geq i, \; j = 1, \ldots, s.$$

It has become customary to denote the method coefficients $c = [c_i]^T$, $b = [b_i]^T$, and $A = \left[a_{i,j}\right]$ using a *Butcher table*, which has the following form for explicit Runge-Kutta methods:

$$\begin{array}{c|ccccc} 0 & 0 & 0 & \cdots & 0 & 0 \\ c_2 & a_{2,1} & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} & 0 \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \end{array} \tag{3}$$

The row-sum conditions can be visualized as summing across the rows of the table.

Notice that a consequence of explicitness is $c_1 = 0$, so that the function is sampled at the beginning of the current integration step.

Example

The Butcher table for the explicit midpoint method (1) is given by:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array} \tag{1}$$

FSAL Schemes

A particularly interesting special class of explicit Runge-Kutta methods, used in most modern codes, are those for which the coefficients have a special structure known as First Same As Last (FSAL):

$$a_{s,i} = b_i,\ i = 1,\ \ldots,\ s-1 \text{ and } b_s = 0. \tag{1}$$

For consistent FSAL schemes the Butcher table (3) has the form:

$$\begin{array}{c|ccccc} 0 & 0 & 0 & \cdots & 0 & 0 \\ c_2 & a_{2,1} & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{s-1} & a_{s-1,1} & a_{s-1,2} & \cdots & 0 & 0 \\ 1 & b_1 & b_2 & \cdots & b_{s-1} & 0 \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & 0 \end{array} \tag{2}$$

The advantage of FSAL methods is that the function value k_s at the end of one integration step is the same as the first function value k_1 at the next integration step.

The function values at the beginning and end of each integration step are required anyway when constructing the `InterpolatingFunction` that is used for dense output in `NDSolve`.

Embedded Pairs and Local Error Estimation

An efficient means of obtaining local error estimates for adaptive step-size control is to consider two methods of different orders p and $\hat{p}$ that share the same coefficient matrix (and hence function values).

$$\begin{array}{c|cccccc}
0 & 0 & 0 & \cdots & 0 & 0 \\
c_2 & a_{2,1} & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & 0 & \vdots \\
c_{s-1} & a_{s-1,1} & a_{s-1,2} & \cdots & 0 & 0 \\
c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} & 0 \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s \\
 & \hat{b}_1 & \hat{b}_2 & \cdots & \hat{b}_{s-1} & \hat{b}_s
\end{array} \tag{1}$$

These give two solutions:

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i \, k_i \tag{2}$$

$$\hat{y}_{n+1} = y_n + h \sum_{i=1}^{s} \hat{b}_i \, k_i \tag{3}$$

A commonly used notation is $p(\hat{p})$, typically with $\hat{p} = p - 1$ or $\hat{p} = p + 1$.

In most modern codes, including the default choice in `NDSolve`, the solution is advanced with the more accurate formula so that $\hat{p} = p - 1$, which is known as local extrapolation.

The vector of coefficients $e = \left[b_1 - \hat{b}_1,\, b_2 - \hat{b}_2,\, \ldots,\, b_s - \hat{b}_s\right]^T$ gives an error estimator avoiding subtractive cancellation of y_n in floating-point arithmetic when forming the difference between (2) and (3).

$$\mathrm{err}_n = h \sum_{i=1}^{s} e_i \, k_i$$

The quantity $\|\mathrm{err}_n\|$ gives a scalar measure of the error that can be used for step size selection.

Step Control

The classical Integral (or I) step-size controller uses the formula:

$$h_{n+1} = h_n \left(\frac{\text{Tol}}{\|\text{err}_n\|} \right)^{1/\tilde{p}} \tag{1}$$

where $\tilde{p} = \min(\hat{p},\ p) + 1$.

The error estimate is therefore used to determine the next step size to use from the current step size.

The notation $\text{Tol}/\|\text{err}_n\|$ is explained within "Norms in NDSolve".

Overview

Explicit Runge-Kutta pairs of orders 2(1) through 9(8) have been implemented.

Formula pairs have the following properties:

- First Same As Last strategy.
- Local extrapolation mode, that is, the higher-order formula is used to propagate the solution.
- Stiffness detection capability (see "StiffnessTest Method Option for NDSolve").
- Proportional-Integral step-size controller for stiff and quasi-stiff systems [G91].

Optimal formula pairs of orders 2(1), 3(2), and 4(3) subject to the already stated requirements have been derived using *Mathematica,* and are described in [SS04].

The 5(4) pair selected is due to Bogacki and Shampine [BS89b, S94] and the 6(5), 7(6), 8(7), and 9(8) pairs are due to Verner.

For the selection of higher-order pairs, issues such as local truncation error ratio and stability region compatibility should be considered (see [S94]). Various tools have been written to assess these qualitative features.

Methods are interchangeable so that, for example, it is possible to substitute the 5(4) method of Bogacki and Shampine with a method of Dormand and Prince.

Summation of the method stages is implemented using level 2 BLAS which is often highly optimized for particular processors and can also take advantage of multiple cores.

Example

Define the Brusselator ODE problem, which models a chemical reaction.

```
In[7]:=  system = GetNDSolveProblem["BrusselatorODE"]
```

Out[7]= NDSolveProblem$\left[\left\{\left\{(Y_1)'[T] == 1 - 4\, Y_1[T] + Y_1[T]^2\, Y_2[T],\ (Y_2)'[T] == 3\, Y_1[T] - Y_1[T]^2\, Y_2[T]\right\},\right.\right.$
$\left.\left.\left\{Y_1[0] == \frac{3}{2},\ Y_2[0] == 3\right\},\ \{Y_1[T],\ Y_2[T]\},\ \{T,\ 0,\ 20\},\ \{\},\ \{\},\ \{\}\right\}\right]$

This solves the system using an explicit Runge-Kutta method.

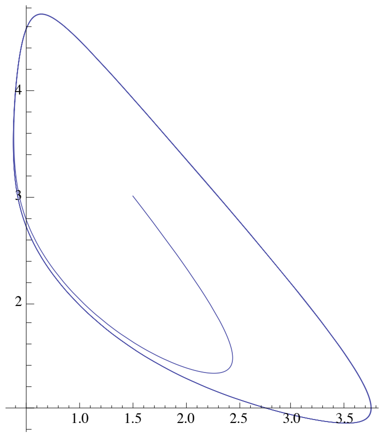
```
In[8]:=  sol = NDSolve[system, Method → "ExplicitRungeKutta"]

Out[8]=  {{Y1[T] → InterpolatingFunction[{{0., 20.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 20.}}, <>][T]}}
```

Extract the interpolating functions from the solution.

```
In[9]:=  ifuns = system["DependentVariables"] /. First[sol];
```

Plot the solution components.

```
In[10]:= ParametricPlot[Evaluate[ifuns], Evaluate[system["TimeData"]]]
```

Out[10]=


Method Comparison

Sometimes you may be interested to find out what methods are being used in `NDSolve`.

Here you can see the coefficients of the default 2(1) embedded pair.

```
In[11]:= NDSolve`EmbeddedExplicitRungeKuttaCoefficients[2, Infinity]
```

Out[11]= $\left\{\left\{\{1\},\ \left\{\frac{1}{2},\ \frac{1}{2}\right\}\right\},\ \left\{\frac{1}{2},\ \frac{1}{2},\ 0\right\},\ \{1,\ 1\},\ \left\{-\frac{1}{2},\ \frac{2}{3},\ -\frac{1}{6}\right\}\right\}$

You also may want to compare some of the different methods to see how they perform for a specific problem.

Utilities

You will make use of a utility function `CompareMethods` for comparing various methods. Some useful `NDSolve` features of this function for comparing methods are:

- The option `EvaluationMonitor`, which is used to count the number of function evaluations
- The option `StepMonitor`, which is used to count the number of accepted and rejected integration steps

This displays the results of the method comparison using a `GridBox`.

```
In[12]:= TabulateResults[labels_List, names_List, data_List] :=
           DisplayForm[
             FrameBox[
              GridBox[
               Apply[{labels, ##} &, MapThread[Prepend, {data, names}]],
               RowLines → True, ColumnLines → True
              ]
             ]
           ] /; SameQ[Length[names], Length[data]];
```

Reference Solution

A number of examples for comparing numerical methods in the literature rely on the fact that a closed-form solution is available, which is obviously quite limiting.

In `NDSolve` it is possible to get very accurate approximations using arbitrary-precision adaptive step size; these are adaptive order methods based on `"Extrapolation"`.

The following reference solution is computed with a method that switches between a pair of `"Extrapolation"` methods, depending on whether the problem appears to be stiff.

```
In[13]:= sol = NDSolve[system, Method → "StiffnessSwitching", WorkingPrecision → 32];

         refsol = First[FinalSolutions[system, sol]];
```

Automatic Order Selection

When you select `"DifferenceOrder" -> Automatic`, the code will automatically attempt to choose the optimal order method for the integration.

Two algorithms have been implemented for this purpose and are described within "SymplecticPartitionedRungeKutta Method for NDSolve".

Example 1

Here is an example that compares built-in methods of various orders, together with the method that is selected automatically.

This selects the order of the methods to choose between and makes a list of method options to pass to `NDSolve`.

```
In[15]:= orders = Join[Range[2, 9], {Automatic}];

         methods = Table[{"ExplicitRungeKutta", "DifferenceOrder" → Part[orders, i],
             "StiffnessTest" → False}, {i, Length[orders]}];
```

Compute the number of integration steps, function evaluations, and the endpoint global error.

```
In[17]:= data = CompareMethods[system, refsol, methods];
```

Display the results in a table.

```
In[18]:= labels = {"Method", "Steps", "Cost", "Error"};

         TabulateResults[labels, orders, data]
```

Out[19]//DisplayForm=

Method	Steps	Cost	Error
2	{124 381, 0}	248 764	1.90685×10^{-8}
3	{4247, 2}	12 749	3.45492×10^{-8}
4	{940, 6}	3786	8.8177×10^{-9}
5	{188, 16}	1430	1.01784×10^{-8}
6	{289, 13}	2418	1.63157×10^{-10}
7	{165, 19}	1842	2.23919×10^{-9}
8	{87, 16}	1341	1.20179×10^{-8}
9	{91, 24}	1842	1.01705×10^{-8}
Automatic	{91, 24}	1843	1.01705×10^{-8}

The default method has order nine, which is close to the optimal order of eight in this example. One function evaluation is needed during the initialization phase to determine the order.

Example 2

A limitation of the previous example is that it did not take into account the accuracy of the solution obtained by each method, so that it did not give a fair reflection of the cost.

Rather than taking a single tolerance to compare methods, it is preferable to use a range of tolerances.

The following example compares various `"ExplicitRungeKutta"` methods of different orders using a variety of tolerances.

This selects the order of the methods to choose between and makes a list of method options to pass to `NDSolve`.

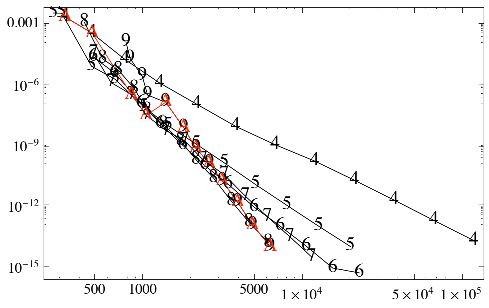
```
In[20]:= orders = Join[Range[4, 9], {Automatic}];

         methods = Table[{"ExplicitRungeKutta", "DifferenceOrder" → Part[orders, i],
             "StiffnessTest" → False}, {i, Length[orders]}];
```

The data comparing accuracy and work is computed using `CompareMethods` for a range of tolerances.

```
In[22]:= data = Table[Map[Rest, CompareMethods[system, refsol,
              methods, AccuracyGoal → tol, PrecisionGoal → tol]], {tol, 3, 14}];
```

The work-error comparison data for the various methods is displayed in the following logarithmic plot, where the global error is displayed on the vertical axis and the number of function evaluations on the horizontal axis. The default order selected (displayed in red) can be seen to increase as the tolerances are increased.

```
In[23]:= ListLogLogPlot[Transpose[data], Joined → True, Axes → False, Frame → True,
          PlotMarkers → Map[Style[#, Medium] &, Join[Drop[orders, -1], {"A"}]],
          PlotStyle → {{Black}, {Black}, {Black}, {Black}, {Black}, {Black}, {Red}}]
```

Out[23]=


The order-selection algorithms are heuristic in that the optimal order may change through the integration but, as the examples illustrate, a reasonable default choice is usually made.

Ideally, a selection of different problems should be used for benchmarking.

Coefficient plug-in

The implementation of `"ExplicitRungeKutta"` provides a default method pair at each order.

Sometimes, however, it is convenient to use a different method, for example:

- To replicate the results of someone else.
- To use a special-purpose method that works well for a specific problem.
- To experiment with a new method.

The Classical Runge-Kutta Method

This shows how to define the coefficients of the classical explicit Runge-Kutta method of order four, approximated to precision p.

```
In[24]:= crkamat = {{1 / 2}, {0, 1 / 2}, {0, 0, 1}};
         crkbvec = {1 / 6, 1 / 3, 1 / 3, 1 / 6};
         crkcvec = {1 / 2, 1 / 2, 1};
         ClassicalRungeKuttaCoefficients[4, p_] := N[{crkamat, crkbvec, crkcvec}, p];
```

The method has no embedded error estimate and hence there is no specification of the coefficient error vector. This means that the method is invoked with fixed step sizes.

Here is an example of the calling syntax.

```
In[27]:= NDSolve[system, Method → {"ExplicitRungeKutta", "DifferenceOrder" → 4,
            "Coefficients" → ClassicalRungeKuttaCoefficients}, StartingStepSize → 1 / 10]
Out[27]= {{Y1[T] → InterpolatingFunction[{{0., 20.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 20.}}, <>][T]}}
```

ode23

This defines the coefficients for a 3(2) FSAL explicit Runge-Kutta pair.

The third-order formula is due to Ralston, and the embedded method was derived by Bogacki and Shampine [BS89a].

This defines a function for computing the coefficients to a desired precision.

```
In[28]:= BSamat = {{1 / 2}, {0, 3 / 4}, {2 / 9, 1 / 3, 4 / 9}};
         BSbvec = {2 / 9, 1 / 3, 4 / 9, 0};
         BScvec  = {1 / 2, 3 / 4, 1};
         BSevec = {-5 / 72, 1 / 12, 1 / 9, -1 / 8};
         BSCoefficients[4, p_] :=
           N[{BSamat, BSbvec, BScvec, BSevec}, p];
```

The method is used in the Texas Instruments TI-85 pocket calculator, Matlab and RKSUITE [S94]. Unfortunately it does not allow for the form of stiffness detection that has been chosen.

A Method of Fehlberg

This defines the coefficients for a 4(5) explicit Runge-Kutta pair of Fehlberg that was popular in the 1960s [F69].

The fourth-order formula is used to propagate the solution, and the fifth-order formula is used only for the purpose of error estimation.

This defines the function for computing the coefficients to a desired precision.

```
In[33]:= Fehlbergamat = {
            {1 / 4},
            {3 / 32, 9 / 32},
            {1932 / 2197, -7200 / 2197, 7296 / 2197}, {439 / 216, -8, 3680 / 513, -845 / 4104},
            {-8 / 27, 2, -3544 / 2565, 1859 / 4104, -11 / 40}};
         Fehlbergbvec = {25 / 216, 0, 1408 / 2565, 2197 / 4104, -1 / 5, 0};
         Fehlbergcvec = {1 / 4, 3 / 8, 12 / 13, 1, 1 / 2};
         Fehlbergevec = {-1 / 360, 0, 128 / 4275, 2197 / 75 240, -1 / 50, -2 / 55};
         FehlbergCoefficients[4, p_] :=
           N[{Fehlbergamat, Fehlbergbvec, Fehlbergcvec, Fehlbergevec}, p];
```

In contrast to the classical Runge-Kutta method of order four, the coefficients include an additional entry that is used for error estimation.

The Fehlberg method is not a FSAL scheme since the coefficient matrix is not of the form (2); it is a six-stage scheme, but it requires six function evaluations per step because of the function evaluation that is required at the end of the step to construct the `InterpolatingFunction`.

A Dormand-Prince Method

Here is how to define a 5(4) pair of Dormand and Prince coefficients [DP80]. This is currently the method used by `ode45` in Matlab.

This defines a function for computing the coefficients to a desired precision.

```
In[38]:= DOPRIamat = {
            {1 / 5},
            {3 / 40, 9 / 40},
            {44 / 45, -56 / 15, 32 / 9},
            {19 372 / 6561, -25 360 / 2187, 64 448 / 6561, -212 / 729},
            {9017 / 3168, -355 / 33, 46 732 / 5247, 49 / 176, -5103 / 18 656},
            {35 / 384, 0, 500 / 1113, 125 / 192, -2187 / 6784, 11 / 84}};
         DOPRIbvec = {35 / 384, 0, 500 / 1113, 125 / 192, -2187 / 6784, 11 / 84, 0};
         DOPRIcvec = {1 / 5, 3 / 10, 4 / 5, 8 / 9, 1, 1};
         DOPRIevec =
           {71 / 57 600, 0, -71 / 16 695, 71 / 1920, -17 253 / 339 200, 22 / 525, -1 / 40};
         DOPRICoefficients[5, p_] :=
           N[{DOPRIamat, DOPRIbvec, DOPRIcvec, DOPRIevec}, p];
```

The Dormand-Prince method is a FSAL scheme since the coefficient matrix is of the form (2); it is a seven-stage scheme, but effectively uses only six function evaluations.

Here is how the coefficients of Dormand and Prince can be used in place of the built-in choice. Since the structure of the coefficients includes an error vector, the implementation is able to ascertain that adaptive step sizes can be computed.

```
In[43]:= NDSolve[system, Method → {"ExplicitRungeKutta", "DifferenceOrder" → 5,
            "Coefficients" → DOPRICoefficients, "StiffnessTest" → False}]

Out[43]= {{Y1[T] → InterpolatingFunction[{{0., 20.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 20.}}, <>][T]}}
```

Method Comparison

Here you solve a system using several explicit Runge-Kutta pairs.

For the Fehlberg 4(5) pair, the option "EmbeddedDifferenceOrder" is used to specify the order of the embedded method.

```
In[44]:= Fehlberg45 = {"ExplicitRungeKutta", "Coefficients" → FehlbergCoefficients,
            "DifferenceOrder" → 4, "EmbeddedDifferenceOrder" → 5, "StiffnessTest" → False};
```

The Dormand and Prince 5(4) pair is defined as follows.

```
In[45]:= DOPRI54 = {"ExplicitRungeKutta", "Coefficients" → DOPRICoefficients,
            "DifferenceOrder" → 5, "StiffnessTest" → False};
```

The 5(4) pair of Bogacki and Shampine is the default order-five method.

```
In[46]:= BS54 = {"ExplicitRungeKutta",
            "Coefficients" → "EmbeddedExplicitRungeKuttaCoefficients",
            "DifferenceOrder" → 5, "StiffnessTest" → False};
```

Put the methods and some descriptive names together in a list.

```
In[47]:= names = {"Fehlberg 4(5)", "Dormand-Prince 5(4)", "Bogacki-Shampine 5(4)"};

         methods = {Fehlberg45, DOPRI54, BS54};
```

Compute the number of integration steps, function evaluations, and the endpoint global error.

```
In[49]:= data = CompareMethods[system, refsol, methods];
```

Display the results in a table.

```
In[50]:= labels = {"Method", "Steps", "Cost", "Error"};

         TabulateResults[labels, names, data]
```

Out[51]//DisplayForm=

Method	Steps	Cost	Error
Fehlberg 4 (5)	{320, 11}	1977	1.52417×10^{-7}
Dormand - Prince 5 (4)	{292, 10}	1814	1.73878×10^{-8}
Bogacki - Shampine 5 (4)	{188, 16}	1430	1.01784×10^{-8}

The default method was the least expensive and provided the most accurate solution.

Method Plug-in

This shows how to implement the classical explicit Runge-Kutta method of order four using the method plug-in environment.

This definition is optional since the method in fact has no data. However, any expression can be stored inside the data object. For example, the coefficients could be approximated here to avoid coercion from rational to floating-point numbers at each integration step.

```
In[52]:= ClassicalRungeKutta /:
           NDSolve`InitializeMethod[ClassicalRungeKutta, __] := ClassicalRungeKutta[];
```

The actual method implementation is written using a stepping procedure.

```
In[53]:= ClassicalRungeKutta[___]["Step"[f_, t_, h_, y_, yp_]] :=
           Block[{deltay, k1, k2, k3, k4},
            k1 = yp;
            k2 = f[t + 1 / 2 h, y + 1 / 2 h k1];
            k3 = f[t + 1 / 2 h, y + 1 / 2 h k2];
            k4 = f[t + h, y + h k3];
            deltay = h (1 / 6 k1 + 1 / 3 k2 + 1 / 3 k3 + 1 / 6 k4);
            {h, deltay}
           ];
```

Notice that the implementation closely resembles the description that you might find in a textbook. There are no memory allocation/deallocation statements or type declarations, for example. In fact the implementation works for machine real numbers or machine complex numbers, and even using arbitrary-precision software arithmetic.

Here is an example of the calling syntax. For simplicity the method only uses fixed step sizes, so you need to specify what step sizes to take.

```
In[54]:= NDSolve[system, Method → ClassicalRungeKutta, StartingStepSize → 1 / 10]

Out[54]= {{Y1[T] → InterpolatingFunction[{{0., 20.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 20.}}, <>][T]}}
```

Many of the methods that have been built into `NDSolve` were first prototyped using top-level code before being implemented in the kernel for efficiency.

Stiffness

Stiffness is a combination of problem, initial data, numerical method, and error tolerances.

Stiffness can arise, for example, in the translation of diffusion terms by divided differences into a large system of ODEs.

In order to understand more about the nature of stiffness it is useful to study how methods behave when applied to a simple problem.

Linear Stability

Consider applying a Runge-Kutta method to a linear scalar equation known as Dahlquist's equation:

$$y'(t) = \lambda\, y(t),\ \lambda \in \mathbb{C},\ \mathrm{Re}(\lambda) < 0. \tag{1}$$

The result is a rational function of polynomials $R(z)$ where $z = h\,\lambda$ (see for example [L87]).

This utility function finds the linear stability function $R(z)$ for Runge-Kutta methods. The form depends on the coefficients and is a polynomial if the Runge-Kutta method is explicit.

Here is the stability function for the fifth-order scheme in the Dormand-Prince 5(4) pair.

```
In[55]:= DOPRIsf = RungeKuttaLinearStabilityFunction[DOPRIamat, DOPRIbvec, z]
```

Out[55]= $1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24} + \frac{z^5}{120} + \frac{z^6}{600}$

This function finds the linear stability function $R(z)$ for Runge-Kutta methods. The form depends on the coefficients and is a polynomial if the Runge-Kutta method is explicit.

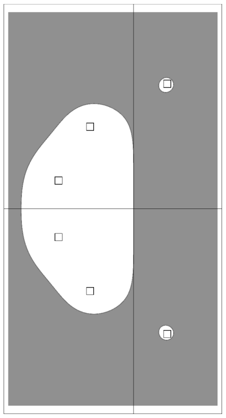
The following package is useful for visualizing linear stability regions for numerical methods for differential equations.

```
In[56]:= Needs["FunctionApproximations`"];
```

You can now visualize the absolute stability region $|R(z)| = 1$.

```
In[57]:= OrderStarPlot[DOPRIsf, 1, z]
```

Out[57]=



Depending on the magnitude of λ in (1), if you choose the step size h such that $|R(h\,\lambda)| < 1$, then errors in successive steps will be damped, and the method is said to be absolutely stable.

If $|R(h\,\lambda)| > 1$, then step-size selection will be restricted by stability and not by local accuracy.

Stiffness Detection

The device for stiffness detection that is used with the option `"StiffnessTest"` is described within "StiffnessTest Method Option for NDSolve".

Recast in terms of explicit Runge-Kutta methods, the condition for stiffness detection can be formulated as:

$$\tilde{\lambda} = \frac{\|k_s - k_{s-1}\|}{\|g_s - g_{s-1}\|} \tag{2}$$

with g_i and k_i defined in (1).

The difference $g_s - g_{s-1}$ can be shown to correspond to a number of applications of the power method applied to $h\,J$.

The difference is therefore a good approximation of the eigenvector corresponding to the leading eigenvalue.

The product $\left|h\,\tilde{\lambda}\right|$ gives an estimate that can be compared to the stability boundary in order to detect stiffness.

An s-stage explicit Runge-Kutta has a form suitable for (2) if $c_{s-1} = c_s = 1$.

$$\begin{array}{c|ccccc} 0 & 0 & 0 & \cdots & 0 & 0 \\ c_2 & a_{2,1} & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & a_{s-1,1} & a_{s-1,2} & \cdots & 0 & 0 \\ 1 & a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} & 0 \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \end{array} \tag{3}$$

The default embedded pairs used in `"ExplicitRungeKutta"` all have the form (3).

An important point is that (2) is very cheap and convenient; it uses already available information from the integration and requires no additional function evaluations.

Another advantage of (3) is that it is straightforward to make use of consistent FSAL methods (1).

Examples

Select a stiff system modeling a chemical reaction.

```
In[58]:= system = GetNDSolveProblem["Robertson"];
```

This applies a built-in explicit Runge-Kutta method to the stiff system.

By default stiffness detection is enabled, since it only has a small impact on the running time.

```
In[59]:= NDSolve[system, Method → "ExplicitRungeKutta"];

   NDSolve::ndstf :
     At T == 0.012555829610695773`, system appears to be stiff. Methods Automatic, BDF or
            StiffnessSwitching may be more appropriate. ≫
```

The coefficients of the Dormand-Prince 5(4) pair are of the form (3) so stiffness detection is enabled.

```
In[60]:= NDSolve[system, Method → {"ExplicitRungeKutta",
             "DifferenceOrder" → 5, "Coefficients" → DOPRICoefficients}];

   NDSolve::ndstf :
     At T == 0.009820727841725293`, system appears to be stiff. Methods Automatic, BDF or
            StiffnessSwitching may be more appropriate. ≫
```

Since no "LinearStabilityBoundary" property has been specified, a default value is chosen. In this case the value corresponds to a generic method of order 5.

```
In[61]:= genlsb = NDSolve`LinearStabilityBoundary[5]

Out[61]= Root[240 + 120 #1 + 60 #1^2 + 20 #1^3 + 5 #1^4 + #1^5 &, 1]
```

You can set up an equation in terms of the linear stability function and solve it exactly to find the point where the contour crosses the negative real axis.

```
In[62]:= DOPRIlsb = Reduce[Abs[DOPRIsf] == 1 && z < 0, z]

Out[62]= z == Root[600 + 300 #1 + 100 #1^2 + 25 #1^3 + 5 #1^4 + #1^5 &, 1]
```

The default generic value is very slightly smaller in magnitude than the computed value.

```
In[63]:= N[{genlsb, DOPRIlsb[[2]]}]

Out[63]= {-3.21705, -3.30657}
```

In general, there may be more than one point of intersection, and it may be necessary to choose the appropriate solution.

The following definition sets the value of the linear stability boundary.

```
In[64]:= DOPRICoefficients[5]["LinearStabilityBoundary"] =
           Root[600 + 300 * #1 + 100 * #1^2 + 25 * #1^3 + 5 * #1^4 + #1^5 &, 1, 0];
```

Using the new value for this example does not affect the time at which stiffness is detected.

```
In[65]:= NDSolve[system, Method → {"ExplicitRungeKutta",
             "DifferenceOrder" → 5, "Coefficients" → DOPRICoefficients}];

    NDSolve::ndstf :
      At T == 0.009820727841725293`, system appears to be stiff. Methods Automatic, BDF or
              StiffnessSwitching may be more appropriate. ≫
```

The Fehlberg 4(5) method does not have the correct coefficient structure (3) required for stiffness detection, since $c_s = 1/2 \neq 1$.

The default value `"StiffnessTest" -> Automatic` checks to see if the method coefficients provide a stiffness detection capability; if they do, then stiffness detection is enabled.

Step Control Revisited

There are some reasons to look at alternatives to the standard Integral step controller (1) when considering mildly stiff problems.

This system models a chemical reaction.

```
In[66]:= system = GetNDSolveProblem["Robertson"];
```

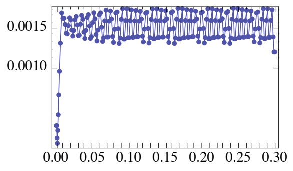
This defines an explicit Runge-Kutta method based on the Dormand-Prince coefficients that does not use stiffness detection.

```
In[67]:= IERK = {"ExplicitRungeKutta", "Coefficients" → DOPRICoefficients,
            "DifferenceOrder" → 5, "StiffnessTest" → False};
```

This solves the system and plots the step sizes that are taken using the utility function `StepDataPlot`.

```
In[68]:= isol = NDSolve[system, Method → IERK];
         StepDataPlot[isol]
```

Out[69]=



Solving a stiff or mildly stiff problem with the standard step-size controller leads to large oscillations, sometimes leading to a number of undesirable step-size rejections.

The study of this issue is known as step-control stability.

It can be studied by matching the linear stability regions for the high- and low-order methods in an embedded pair.

One approach to addressing the oscillation is to derive special methods, but this compromises the local accuracy.

PI Step Control

An appealing alternative to Integral step control (1) is Proportional-Integral or PI step control.

In this case the step size is selected using the local error in two successive integration steps according to the formula:

$$h_{n+1} = h_n \left(\frac{\text{Tol}}{\|\text{err}_n\|}\right)^{k_1/\tilde{p}} \left(\frac{\|\text{err}_{n-1}\|}{\|\text{err}_n\|}\right)^{k_2/\tilde{p}} \qquad (1)$$

This has the effect of damping and hence gives a smoother step-size sequence.

Note that Integral step control (1) is a special case of (1) and is used if a step is rejected:

$$k_1 = 1,\ k_2 = 0\,.$$

The option `"StepSizeControlParameters" -> {k1, k2}` can be used to specify the values of k_1 and k_2.

The scaled error estimate in (1) is taken to be $\|\text{err}_{n-1}\| = \|\text{err}_n\|$ for the first integration step.

Examples

Stiff Problem

This defines a method similar to `IERK` that uses the option `"StepSizeControlParameters"` to specify a PI controller.

Here you use generic control parameters suggested by Gustafsson:

$$k_1 = 3/10,\quad k_2 = 2/5$$

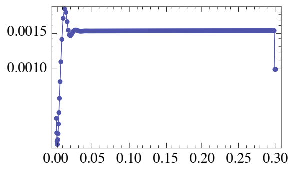
This specifies the step-control parameters.

```
In[70]:= PIERK = {"ExplicitRungeKutta",
            "Coefficients" → DOPRICoefficients, "DifferenceOrder" → 5,
            "StiffnessTest" → False, "StepSizeControlParameters" → {3 / 10, 2 / 5}};
```

Solving the system again, it can be observed that the step-size sequence is now much smoother.

```
In[71]:= pisol = NDSolve[system, Method → PIERK];
         StepDataPlot[pisol]
```

Out[72]=



Nonstiff Problem

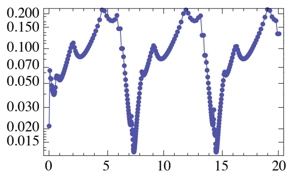
In general the I step controller (1) is able to take larger steps for a nonstiff problem than the PI step controller (1) as the following example illustrates.

Select and solve a nonstiff system using the I step controller.

```
In[73]:= system = GetNDSolveProblem["BrusselatorODE"];
```

```
In[74]:= isol = NDSolve[system, Method → IERK];
         StepDataPlot[isol]
```

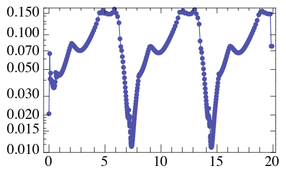
Out[75]=



Using the PI step controller the step sizes are slightly smaller.

```
In[76]:= pisol = NDSolve[system, Method → PIERK];
         StepDataPlot[pisol]
```

Out[77]=



For this reason, the default setting for `"StepSizeControlParameters"` is `Automatic` , which is interpreted as:

- Use the I step controller (1) if `"StiffnessTest" -> False`.
- Use the PI step controller (1) if `"StiffnessTest" -> True`.

Fine-Tuning

Instead of using (1) directly, it is common practice to use safety factors to ensure that the error is acceptable at the next step with high probability, thereby preventing unwanted step rejections.

The option `"StepSizeSafetyFactors" -> {`s_1`, `s_2`}` specifies the safety factors to use in the step-size estimate so that (1) becomes:

$$h_{n+1} = h_n\, s_1 \left(\frac{s_2\, \mathrm{Tol}}{\|\mathrm{err}_n\|}\right)^{k_1/\tilde{p}} \left(\frac{\|\mathrm{err}_{n-1}\|}{\|\mathrm{err}_n\|}\right)^{k_2/\tilde{p}}. \qquad (1)$$

Here s_1 is an absolute factor and s_2 typically scales with the order of the method.

The option `"StepSizeRatioBounds" -> {`sr_{min}`, `sr_{max}`}` specifies bounds on the next step size to take such that:

$$sr_{min} \le \left|\frac{h_{n+1}}{h_n}\right| \le sr_{max}. \qquad (2)$$

Option summary

option name	*default value*	
`"Coefficients"`	`EmbeddedExplicitRungeKuttaCoefficients`	specify the coefficients of the explicit Runge-Kutta method
`"DifferenceOrder"`	`Automatic`	specify the order of local accuracy
`"EmbeddedDifferenceOrder"`	`Automatic`	specify the order of the embedded method in a pair of explicit Runge-Kutta methods
`"StepSizeControlParameters"`	`Automatic`	specify the PI step-control parameters (see (1))
`"StepSizeRatioBounds"`	$\left\{\frac{1}{8}, 4\right\}$	specify the bounds on a relative change in the new step size (see (2))
`"StepSizeSafetyFactors"`	`Automatic`	specify the safety factors to use in the step-size estimate (see (1))
`"StiffnessTest"`	`Automatic`	specify whether to use the stiffness detection capability

Options of the method `"ExplicitRungeKutta"`.

The default setting of `Automatic` for the option `“DifferenceOrder“` selects the default coefficient order based on the problem, initial values-and local error tolerances, balanced against the work of the method for each coefficient set.

The default setting of `Automatic` for the option `“EmbeddedDifferenceOrder“` specifies that the default order of the embedded method is one lower than the method order. This depends on the value of the `“DifferenceOrder“` option.

The default setting of `Automatic` for the option `“StepSizeControlParameters“` uses the values `{1, 0}` if stiffness detection is active and `{3 / 10, 2 / 5}` otherwise.

The default setting of `Automatic` for the option `“StepSizeSafetyFactors“` uses the values `{17 / 20, 9 / 10}` if the I step controller (1) is used and `{9 / 10, 9 / 10}` if the PI step controller (1) is used. The step controller used depends on the values of the options `“StepSizeControlParameters“` and `“StiffnessTest“`.

The default setting of `Automatic` for the option `“StiffnessTest“` will activate the stiffness test if if the coefficients have the form (3).

"ImplicitRungeKutta" Method for NDSolve

Introduction

Implicit Runge-Kutta methods have a number of desirable properties.

The Gauss-Legendre methods, for example, are self-adjoint, meaning that they provide the same solution when integrating forward or backward in time.

This loads packages defining some example problems and utility functions.

```
In[3]:= Needs[“DifferentialEquations`NDSolveProblems`“];
        Needs[“DifferentialEquations`NDSolveUtilities`“];
```

Coefficients

A generic framework for implicit Runge-Kutta methods has been implemented. The focus so far is on methods with interesting geometric properties and currently covers the following schemes:

- `“ImplicitRungeKuttaGaussCoefficients“`
- `“ImplicitRungeKuttaLobattoIIIACoefficients“`

- “`ImplicitRungeKuttaLobattoIIIBCoefficients`“
- “`ImplicitRungeKuttaLobattoIIICCoefficients`“
- “`ImplicitRungeKuttaRadauIACoefficients`“
- “`ImplicitRungeKuttaRadauIIACoefficients`“

The derivation of the method coefficients can be carried out to arbitrary order and arbitrary precision.

Coefficient Generation

- Start with the definition of the polynomial, defining the abscissas of the s stage coefficients. For example, the abscissas for Gauss-Legendre methods are defined as $\frac{d^s}{dx^s} x^s (1-x)^s$.
- Univariate polynomial factorization gives the underlying irreducible polynomials defining the roots of the polynomials.
- `Root` objects are constructed to represent the solutions (using unique root isolation and Jenkins-Traub for the numerical approximation).
- `Root` objects are then approximated numerically for precision coefficients.
- Condition estimates for Vandermonde systems governing the coefficients yield the precision to take in approximating the roots numerically.
- Specialized solvers for nonconfluent Vandermonde systems are then used to solve equations for the coefficients (see [GVL96]).
- One step of iterative refinement is used to polish the approximate solutions and to check that the coefficients are obtained to the requested precision.

This generates the coefficients for the two-stage fourth-order Gauss-Legendre method to 50 decimal digits of precision.

```
In[5]:= NDSolve`ImplicitRungeKuttaGaussCoefficients[4, 50]

Out[5]= {{{0.25000000000000000000000000000000000000000000000000,
            -0.038675134594812882254574390250978727823800875635063},
          {0.53867513459481288225457439025097872782380087563506,
            0.25000000000000000000000000000000000000000000000000}},
         {0.50000000000000000000000000000000000000000000000000,
          0.50000000000000000000000000000000000000000000000000},
         {0.21132486540518711774542560974902127217619912436494,
          0.78867513459481288225457439025097872782380087563506}}
```

The coefficients have the form $\{a, b^T, c^T\}$.

This generates the coefficients for the two-stage fourth-order Gauss-Legendre method exactly. For high-order methods, generating the coefficients exactly can often take a very long time.

```
In[6]:= NDSolve`ImplicitRungeKuttaGaussCoefficients[4, Infinity]
```

Out[6]=
$$\left\{\left\{\left\{\frac{1}{4}, \frac{1}{12}\left(3-2\sqrt{3}\right)\right\}, \left\{\frac{1}{12}\left(3+2\sqrt{3}\right), \frac{1}{4}\right\}\right\}, \left\{\frac{1}{2}, \frac{1}{2}\right\}, \left\{\frac{1}{6}\left(3-\sqrt{3}\right), \frac{1}{6}\left(3+\sqrt{3}\right)\right\}\right\}$$

This generates the coefficients for the six-stage tenth-order RaduaIA implicit Runge-Kutta method to 20 decimal digits of precision.

```
In[7]:= NDSolve`ImplicitRungeKuttaRadauIACoefficients[10, 20]
```

```
Out[7]= {{{0.040000000000000000000, -0.087618018725274235050,
           0.085317987638600293760, -0.055818078483298114837, 0.018118109569972056127},
          {0.040000000000000000000, 0.12875675325490976116, -0.047477730403197434295,
           0.026776985967747870688, -0.0082961444756796453993},
          {0.040000000000000000000, 0.23310008036710237092, 0.16758507013524896344,
           -0.032883343543501401775, 0.0086077606722332473607},
          {0.040000000000000000000, 0.21925333267709602305, 0.33134489917971587453,
           0.14621486784749350665, -0.013656113342429231907},
          {0.040000000000000000000, 0.22493691761630663460, 0.30390571559725175840,
           0.30105430635402060050, 0.072998864317903324306}},
         {0.040000000000000000000, 0.22310390108357074440, 0.31182652297574125408,
          0.28135601514946206019, 0.14371356079122594132},
         {0, 0.13975986434378055215, 0.41640956763108317994,
          0.72315698636187617232, 0.94289580388548231781}}
```

Examples

Load an example problem.

```
In[8]:= system = GetNDSolveProblem["PerturbedKepler"];
        vars = system["DependentVariables"];
```

This problem has two invariants that should remain constant. A numerical method may not be able to conserve these invariants.

```
In[10]:= invs = system["Invariants"]
```

Out[10]=
$$\left\{-\frac{1}{400\left(Y_1[T]^2+Y_2[T]^2\right)^{3/2}}-\frac{1}{\sqrt{Y_1[T]^2+Y_2[T]^2}}+\frac{1}{2}\left(Y_3[T]^2+Y_4[T]^2\right), -Y_2[T]\,Y_3[T]+Y_1[T]\,Y_4[T]\right\}$$

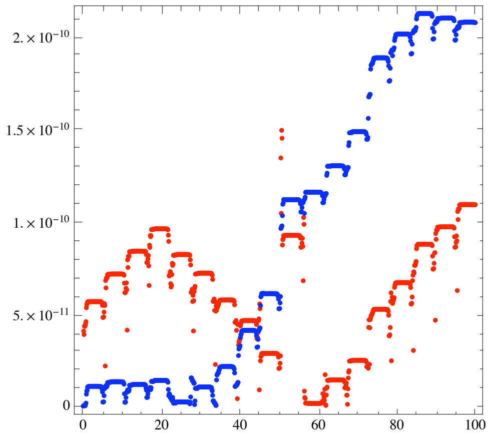
This solves the system using an implicit Runge-Kutta Gauss method. The order of the scheme is selected using the "DifferenceOrder" method option.

```
In[11]:= sol = NDSolve[system, Method →
             {"FixedStep", Method → {"ImplicitRungeKutta", "DifferenceOrder" → 10}},
           StartingStepSize → 1 / 10]
```

Out[11]=
$$\{\{Y_1[T] \to \text{InterpolatingFunction}[\{\{0., 100.\}\}, <>][T],$$
$$Y_2[T] \to \text{InterpolatingFunction}[\{\{0., 100.\}\}, <>][T],$$
$$Y_3[T] \to \text{InterpolatingFunction}[\{\{0., 100.\}\}, <>][T],$$
$$Y_4[T] \to \text{InterpolatingFunction}[\{\{0., 100.\}\}, <>][T]\}\}$$

A plot of the error in the invariants shows an increase as the integration proceeds.

```
In[12]:= InvariantErrorPlot[invs, vars, T, sol,
          PlotStyle → {Red, Blue}, InvariantErrorSampleRate → 1]
```

Out[12]=


The `"ImplicitSolver"` method of `"ImplicitRungeKutta"` has options `AccuracyGoal` and `PrecisionGoal` that specify the absolute and relative error to aim for in solving the nonlinear system of equations.

These options have the same default values as the corresponding options in `NDSolve`, since often there is little point in solving the nonlinear system to much higher accuracy than the local error of the method.

However, for certain types of problems it can be useful to solve the nonlinear system up to the working precision.

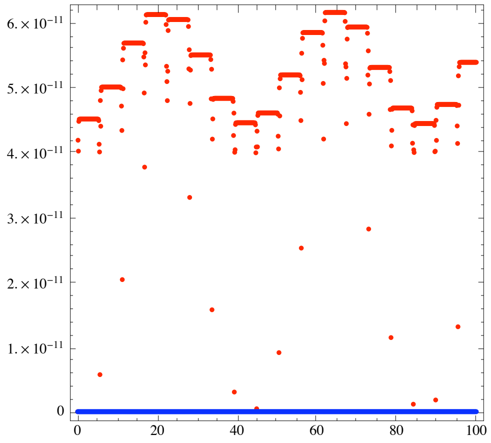
```
In[13]:= sol = NDSolve[system,
            Method → {"FixedStep", Method → {"ImplicitRungeKutta", "DifferenceOrder" → 10,
                 "ImplicitSolver" → {"Newton", AccuracyGoal → MachinePrecision,
                   PrecisionGoal → MachinePrecision,
                   "IterationSafetyFactor" → 1}}}, StartingStepSize → 1 / 10]

Out[13]= {{Y1[T] → InterpolatingFunction[{{0., 100.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 100.}}, <>][T],
           Y3[T] → InterpolatingFunction[{{0., 100.}}, <>][T],
           Y4[T] → InterpolatingFunction[{{0., 100.}}, <>][T]}}
```

The first invariant is the Hamiltonian of the system, and the error is now bounded, as it should be, since the Gauss implicit Runge-Kutta method is a symplectic integrator.

The second invariant is conserved exactly (up to roundoff) since the Gauss implicit Runge-Kutta method conserves quadratic invariants.

```
In[14]:= InvariantErrorPlot[invs, vars, T, sol,
          PlotStyle → {Red, Blue}, InvariantErrorSampleRate → 1]
```

Out[14]=



This defines the implicit midpoint method as the one-stage implicit Runge-Kutta method of order two.

For this problem it can be more efficient to use a fixed-point iteration instead of a Newton iteration to solve the nonlinear system.

```
In[15]:= ImplicitMidpoint = {"FixedStep", Method → {"ImplicitRungeKutta", "Coefficients" ->
                "ImplicitRungeKuttaGaussCoefficients", "DifferenceOrder" → 2,
              "ImplicitSolver" → {"FixedPoint", "AccuracyGoal" → MachinePrecision,
                 "PrecisionGoal" → MachinePrecision, "IterationSafetyFactor" → 1 }}};
```

```
In[16]:= NDSolve[system, {T, 0, 1}, Method → ImplicitMidpoint, StartingStepSize → 1 / 100]
```

```
Out[16]= {{Y1[T] → InterpolatingFunction[{{0., 1.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 1.}}, <>][T],
           Y3[T] → InterpolatingFunction[{{0., 1.}}, <>][T],
           Y4[T] → InterpolatingFunction[{{0., 1.}}, <>][T]}}
```

At present, the implicit Runge-Kutta method framework does not use banded Newton techniques for uncoupling the nonlinear system.

Option Summary

"ImplicitRungeKutta" Options

option name	default value	
"Coefficients"	"ImplicitRungeKuttaGaussCoefficients"	specify the coefficients to use in the implicit Runge-Kutta method
"DifferenceOrder"	Automatic	specify the order of local accuracy of the method
"ImplicitSolver"	"Newton"	specify the solver to use for the nonlinear system; valid settings are FixedPoint or "Newton"
"StepSizeControlParameters"	Automatic	specify the step control parameters
"StepSizeRatioBounds"	$\left\{\frac{1}{8}, 4\right\}$	specify the bounds on a relative change in the new step size
"StepSizeSafetyFactors"	Automatic	specify the safety factors to use in the step size estimate

Options of the method "ImplicitRungeKutta".

The default setting of Automatic for the option "StepSizeSafetyFactors" uses the values {9 / 10, 9 / 10}.

"ImplicitSolver" Options

option name	default value	
AccuracyGoal	Automatic	specify the absolute tolerance to use in solving the nonlinear system
"IterationSafetyFactor"	$\frac{1}{100}$	specify the safety factor to use in solving the nonlinear system
MaxIterations	Automatic	specify the maximum number of iterations to use in solving the nonlinear system
PrecisionGoal	Automatic	specify the relative tolerance to use in solving the nonlinear system

Common options of "ImplicitSolver".

option name	*default value*	
`"JacobianEvaluationParameter"`	$\frac{1}{1000}$	specify when to recompute the Jacobian matrix in Newton iterations
`"LinearSolveMethod"`	`Automatic`	specify the linear solver to use in Newton iterations
`"LUDecompositionEvaluationParameter"`	$\frac{6}{5}$	specify when to compute LU decompositions in Newton iterations

Options specific to the “`Newton`“ method of “`ImplicitSolver`“.

"SymplecticPartitionedRungeKutta" Method for NDSolve

Introduction

When numerically solving Hamiltonian dynamical systems it is advantageous if the numerical method yields a symplectic map.

- The phase space of a Hamiltonian system is a symplectic manifold on which there exists a natural symplectic structure in the canonically conjugate coordinates.
- The time evolution of a Hamiltonian system is such that the Poincaré integral invariants associated with the symplectic structure are preserved.
- A symplectic integrator computes exactly, assuming infinite precision arithmetic, the evolution of a nearby Hamiltonian, whose phase space structure is close to that of the original system.

If the Hamiltonian can be written in separable form, $H(p, q) = T(p) + V(q)$, there exists an efficient class of explicit symplectic numerical integration methods.

An important property of symplectic numerical methods when applied to Hamiltonian systems is that a nearby Hamiltonian is approximately conserved for exponentially long times (see [BG94], [HL97], and [R99]).

Hamiltonian Systems

Consider a differential equation

$$\frac{dy}{dt} = F(t, y),\ y(t_0) = y_0. \tag{1}$$

A d-degree of freedom Hamiltonian system is a particular instance of (1) with $y = (p_1, \ldots, p_d, q_1 \ldots, q_d)^T$, where

$$\frac{dy}{dt} = J^{-1} \nabla H. \tag{2}$$

Here ∇ represents the gradient operator:

$$\nabla = \ (\partial / \partial p_1, \ldots, \partial / \partial p_d, \partial / \partial q_1, \ldots \partial / \partial q_d)^T$$

and J is the skew symmetric matrix:

$$J = \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix}$$

where I and 0 are the identity and zero $d{\times}d$ matrices.

The components of q are often referred to as position or coordinate variables and the components of p as the momenta.

If H is autonomous, $dH/dt = 0$. Then H is a conserved quantity that remains constant along solutions of the system. In applications, this usually corresponds to conservation of energy.

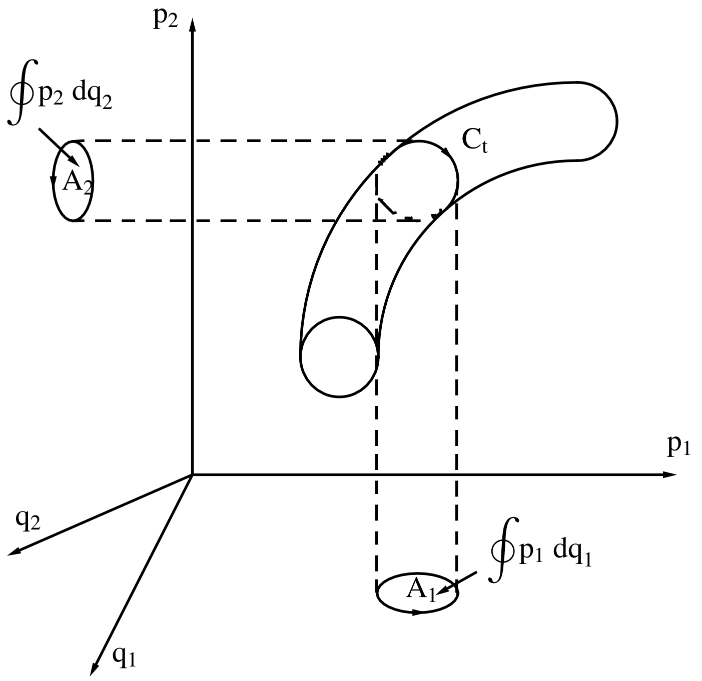
A numerical method applied to a Hamiltonian system (2) is said to be symplectic if it produces a symplectic map. That is, let $(p^*, q^*) = \psi(p, q)$ be a C^1 transformation defined in a domain Ω.:

$$\forall\, (p, q) \in \Omega,\ \psi'^T J\, \psi' = \frac{\partial (p^*, q^*)^T}{\partial (p, q)}\, J\, \frac{\partial (p^*, q^*)}{\partial (p, q)} = J$$

where the Jacobian of the transformation is:

$$\psi' = \frac{\partial (p^*, q^*)}{\partial (p, q)} = \begin{pmatrix} \frac{\partial p^*}{\partial p} & \frac{\partial p^*}{\partial q} \\ \frac{\partial q^*}{\partial p} & \frac{\partial q^*}{\partial q} \end{pmatrix}.$$

The flow of a Hamiltonian system is depicted together with the projection onto the planes formed by canonically conjugate coordinate and momenta pairs. The sum of the oriented areas remains constant as the flow evolves in time.



Partitioned Runge-Kutta Methods

It is sometimes possible to integrate certain components of (1) using one Runge-Kutta method and other components using a different Runge-Kutta method. The overall s-stage scheme is called a partitioned Runge-Kutta method and the free parameters are represented by two Butcher tableaux:

$$\begin{array}{c|ccc} & a_{11} & \cdots & a_{1s} \\ & \vdots & \ddots & \vdots \\ & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array} \qquad \begin{array}{c|ccc} & A_{11} & \cdots & A_{1s} \\ & \vdots & \ddots & \vdots \\ & A_{s1} & \cdots & A_{ss} \\ \hline & B_1 & \cdots & B_s \end{array}. \tag{1}$$

Symplectic Partitioned Runge-Kutta (SPRK) Methods

For general Hamiltonian systems, symplectic Runge-Kutta methods are necessarily implicit.

However, for separable Hamiltonians $H(p, q, t) = T(p) + V(q, t)$ there exist explicit schemes corresponding to symplectic partitioned Runge-Kutta methods.

Instead of (1) the free parameters now take either the form:

$$\begin{array}{c|cccc} & 0 & \cdots & 0 & 0 \\ & b_1 & 0 & \cdots & \vdots \\ & \vdots & \ddots & \ddots & \vdots \\ & b_1 & \cdots & b_{s-1} & 0 \\ \hline & b_1 & \cdots & b_{s-1} & b_s \end{array} \qquad \begin{array}{c|cccc} & B_1 & \cdots & 0 & 0 \\ & B_1 & B_2 & \cdots & \vdots \\ & \vdots & \vdots & \ddots & \vdots \\ & B_1 & B_2 & \cdots & B_s \\ \hline & B_1 & B_2 & \cdots & B_s \end{array} \tag{1}$$

or the form:

$$\begin{array}{c|cccc} & b_1 & \cdots & 0 & 0 \\ & b_1 & b_2 & \cdots & \vdots \\ & \vdots & \vdots & \ddots & \vdots \\ & b_1 & b_2 & \cdots & b_s \\ \hline & b_1 & b_2 & \cdots & b_s \end{array} \qquad \begin{array}{c|cccc} & 0 & \cdots & 0 & 0 \\ & B_1 & 0 & \cdots & \vdots \\ & \vdots & \ddots & \ddots & \vdots \\ & B_1 & \cdots & B_{s-1} & 0 \\ \hline & B_1 & \cdots & B_{s-1} & B_s \end{array}. \tag{2}$$

The $2d$ free parameters of (2) are sometimes represented using the shorthand notation $[b_1, \ldots, b_s]\,(B_1, \ldots \mathrm{B}_s)$.

The differential system for a separable Hamiltonian system can be written as:

$$\frac{dp_i}{dt} = f(q, t) = -\frac{\partial V(q, t)}{\partial q_i}, \quad \frac{dq_i}{dt} = g(p) = \frac{\partial T(p)}{\partial p_i}, \quad i = 1, \ldots, d.$$

In general the force evaluations $-\partial V(q, t)/\partial q$ are computationally dominant and (2) is preferred over (1) since it is possible to save one force evaluation per time step when dense output is required.

Standard Algorithm

The structure of (2) permits a particularly simple implementation (see for example [SC94]).

Algorithm 1 (Standard SPRK)

```
$P_0 = p_n$
$Q_1 = q_n$

for $i = 1, \ldots, s$
```

$$P_i = P_{i-1} + h_{n+1}\, b_i\, f\,(Q_i,\, t_n + C_i\, h_{n+1})$$
$$Q_{i+1} = Q_i + h_{n+1}\, B_i\, g(P_i)$$

Return $p_{n+1} = P_s$ and $q_{n+1} = Q_{s+1}$.

The time-weights are given by: $C_j = \sum_{i=1}^{j-1} B_i,\ j = 1,\, \ldots,\, s$.

If $B_s = 0$ then Algorithm 1 effectively reduces to an $s-1$ stage scheme since it has the First Same As Last (FSAL) property.

Example

This loads some useful packages.

```
In[1]:= Needs["DifferentialEquations`NDSolveProblems`"];
        Needs["DifferentialEquations`NDSolveUtilities`"];
```

The Harmonic Oscillator

The Harmonic oscillator is a simple Hamiltonian problem that models a material point attached to a spring. For simplicity consider the unit mass and spring constant for which the Hamiltonian is given in separable form:

$$H(p,\, q) = T(p) + V(q) = p^2 \big/ 2 + q^2 \big/ 2.$$

The equations of motion are given by:

$$\frac{dp}{dt} = -\frac{\partial H}{\partial q} = -q, \quad \frac{dq}{dt} = \frac{\partial H}{\partial p} = p, \quad q(0) = 1, \quad p(0) = 0. \tag{1}$$

Input

```
In[3]:= system = GetNDSolveProblem["HarmonicOscillator"];
        eqs = {system["System"], system["InitialConditions"]};
        vars = system["DependentVariables"];
        H = system["Invariants"];
        time = {T, 0, 100};
        step = 1 / 25;
```

Explicit Euler Method

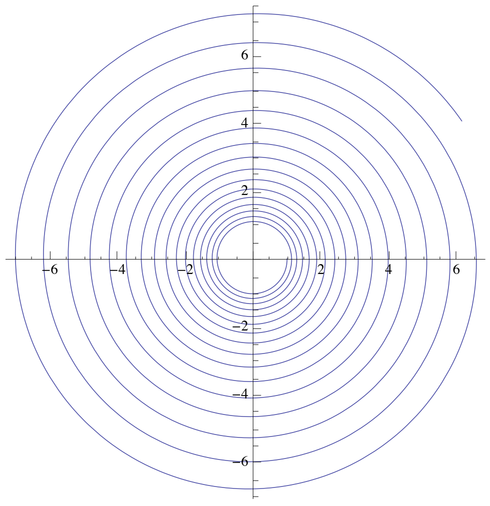
Numerically integrate the equations of motion for the Harmonic oscillator using the explicit Euler method.

```
In[9]:= solee = NDSolve[eqs, vars, time, Method → "ExplicitEuler",
           StartingStepSize → step, MaxSteps → Infinity];
```

Since the method is dissipative, the trajectory spirals into or away from the fixed point at the origin.

```
In[10]:= ParametricPlot[Evaluate[vars /. First[solee]], Evaluate[time], PlotPoints → 100]
```

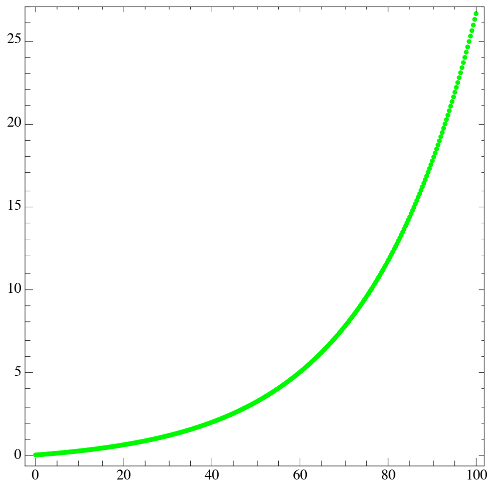
Out[10]=



A dissipative method typically exhibits linear error growth in the value of the Hamiltonian.

```
In[11]:= InvariantErrorPlot[H, vars, T, solee, PlotStyle → Green]
```

Out[11]=



Symplectic Method

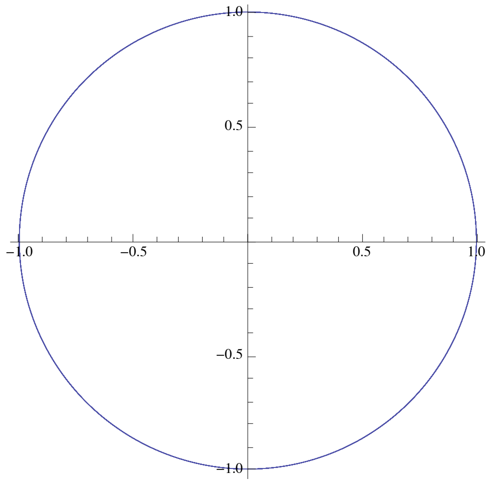
Numerically integrate the equations of motion for the Harmonic oscillator using a symplectic partitioned Runge-Kutta method.

```
In[12]:= sol = NDSolve[eqs, vars, time, Method → {"SymplecticPartitionedRungeKutta",
               "DifferenceOrder" → 2, "PositionVariables" → {Y₁[T]}},
            StartingStepSize → step, MaxSteps → Infinity];
```

The solution is now a closed curve.

In[13]:=
```
ParametricPlot[Evaluate[vars /. First[sol]], Evaluate[time]]
```

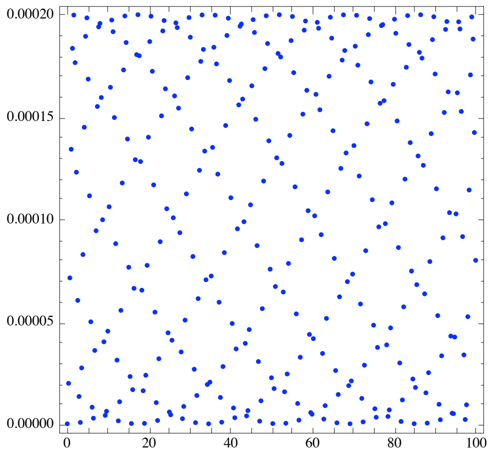
Out[13]=



In contrast to dissipative methods, symplectic integrators yield an error in the Hamiltonian that remains bounded.

In[14]:=
```
InvariantErrorPlot[H, vars, T, sol, PlotStyle → Blue]
```

Out[14]=



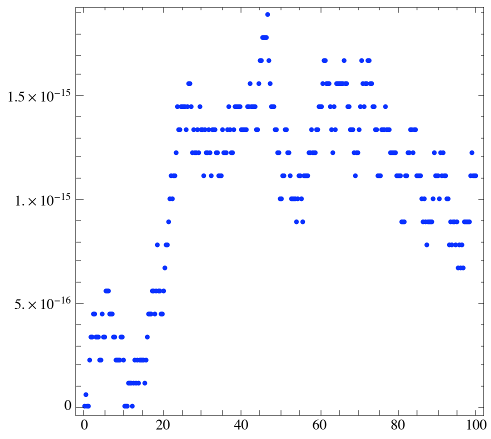
Rounding Error Reduction

In certain cases, lattice symplectic methods exist and can avoid step-by-step roundoff accumulation, but such an approach is not always possible [ET92].

Consider the previous example where the combination of step size and order of the method is now chosen such that the error in the Hamiltonian is around the order of unit roundoff in IEEE double-precision arithmetic.

```
In[15]:= solnoca = NDSolve[eqs, vars, time, Method → {"SymplecticPartitionedRungeKutta",
             "DifferenceOrder" → 10, "PositionVariables" → {Y1[T]}},
            StartingStepSize → step, MaxSteps → Infinity, "CompensatedSummation" → False];

         InvariantErrorPlot[H, vars, T, solnoca, PlotStyle → Blue]
```

Out[16]=

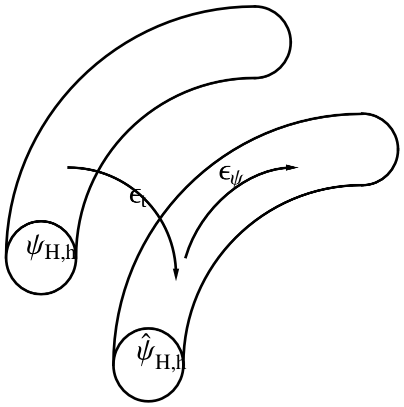


There is a curious drift in the error in the Hamiltonian that is actually a numerical artifact of floating-point arithmetic.

This phenomenon can have an impact on long time integrations.

This section describes the formulation used by `"SymplecticPartitionedRungeKutta"` in order to reduce the effect of such errors.

There are two types of errors in integrating a flow numerically, those along the flow and those transverse to the flow. In contrast to dissipative systems, the rounding errors in Hamiltonian systems that are transverse to the flow are not damped asymptotically.



Many numerical methods for ordinary differential equations involve computations of the form:

$$y_{n+1} = y_n + \delta_n$$

where the increments δ_n are usually smaller in magnitude than the approximations y_n.

Let $e(x)$ denote the exponent and $m(x)$, $1 > m(x) \geq 1/\beta$, the mantissa of a number x in precision p radix β arithmetic: $x = m(x) \times \beta^{e(x)}$.

Then you can write:

$$y_n = m(y_n) \times \beta^{e(y_n)} = y_n^h + y_n^l \times \beta^{e(\delta_n)}$$

and

$$\delta_n = m(\delta_n) \times \beta^{e(\delta_n)} = \delta_n^h + \delta_n^l \times \beta^{e(y_n)-p}.$$

Aligning according to exponents these quantities can be represented pictorially as:

	y_n^l	y_n^h
δ_n^l	δ_n^h	

where numbers on the left have a smaller scale than numbers on the right.

Of interest is an efficient way of computing the quantities δ_n^l that effectively represent the radix β digits discarded due to the difference in the exponents of y_n and δ_n.

Compensated Summation

The basic motivation for compensated summation is to simulate $2n$ bit addition using only n bit arithmetic.

Example

This repeatedly adds a fixed amount to a starting value. Cumulative roundoff error has a significant influence on the result.

```
In[17]:= reps = 10^6;
         base = 0.;
         inc = 0.1;
         Do[base = base + inc, {reps}];
         InputForm[base]
Out[21]//InputForm= 100000.00000133288
```

In many applications the increment may vary and the number of operations is not known in advance.

Algorithm

Compensated summation (see for example [B87] and [H96]) computes the rounding error along with the sum so that

$$y_{n+1} = y_n + h\,f(y_n)$$

is replaced by:

Algorithm 2 (Compensated Summation)

```
$yerr = 0$
for $i = 1, \ldots, N$
    $\Delta y_n = h\,f(y_n) + yerr$
    $y_{n+1} = y_n + \Delta y_n$
    $yerr = (y_n - y_{n+1}) + \Delta y_n$
```

The algorithm is carried out component-wise for vectors.

Example

The function `CompensatedPlus` (in the `Developer`` context) implements the algorithm for compensated summation.

By repeatedly feeding back the rounding error from one sum into the next, the effect of rounding errors is significantly reduced.

```
In[22]:= err = 0.;
         base = 0.;
         inc = 0.1;
         Do[
           {base, err} =
            Developer`CompensatedPlus[base , inc, err],
           {reps}];
         InputForm[base]
Out[26]//InputForm= 100000.
```

An undocumented option `CompensatedSummation` controls whether built-in integration methods in `NDSolve` use compensated summation.

An Alternative Algorithm

There are various ways that compensated summation can be used.

One way is to compute the error in every addition update in the main loop in Algorithm 1.

An alternative algorithm, which was proposed because of its more general applicability, together with reduced arithmetic cost, is given next. The essential ingredients are the increments $\Delta P_i = P_i - p_n$ and $\Delta Q_i = Q_i - q_n$.

Algorithm 3 (Increment SPRK)

```
$\Delta P_0 = 0$
$\Delta Q_1 = 0$

for $i = 1, \ldots, s$
    $\Delta P_i = \Delta P_{i-1} + h_{n+1}\, b_i\, f(q_n + \Delta Q_i,\, t_n + C_i\, h_{n+1})$
    $\Delta Q_{i+1} = \Delta Q_i + h_{n+1}\, B_i\, g(p_n + \Delta P_i)$

Return $\Delta p_{n+1} = \Delta P_s$ and $\Delta q_{n+1} = \Delta Q_{s+1}$.
```

The desired values $p_{n+1} = p_n + \Delta p_{n+1}$ and $q_{n+1} = q_n + \Delta q_{n+1}$ are obtained using compensated summation.

Compensated summation could also be used in every addition update in the main loop of Algorithm 3, but our experiments have shown that this adds a non-negligible overhead for a relatively small gain in accuracy.

Numerical Illustration

Rounding Error Model

The amount of expected roundoff error in the relative error of the Hamiltonian for the harmonic oscillator (1) will now be quantified. A probabilistic average case analysis is considered in preference to a worst case upper bound.

For a one-dimensional random walk with equal probability of a deviation, the expected absolute distance after N steps is $O(\sqrt{n})$.

The relative error for a floating-point operation $+$, $-$, $*$, $/$ using IEEE round to nearest mode satisfies the following bound [K93]:

$$\epsilon_{\text{round}} \leq 1/2\, \beta^{-p+1} \approx 1.11022 \times 10^{-16}$$

where the base $\beta = 2$ is used for representing floating-point numbers on the machine and $p = 53$ for IEEE double-precision.

Therefore the roundoff error after n steps is expected to be approximately:

$$k\, \epsilon \sqrt{n}$$

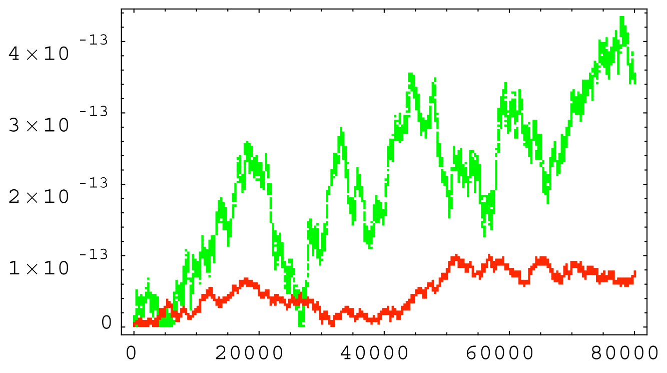
for some constant k.

In the examples that follow a constant step size of 1/25 is used and the integration is performed over the interval [0, 80000] for a total of 2×10^{6} integration steps. The error in the Hamiltonian is sampled every 200 integration steps.

The 8th-order 15-stage (FSAL) method D of Yoshida is used. Similar results have been obtained for the 6th-order 7-stage (FSAL) method A of Yoshida with the same number of integration steps and a step size of 1/160.

Without Compensated Summation

The relative error in the Hamiltonian is displayed here for the standard formulation in Algorithm 1 (green) and for the increment formulation in Algorithm 3 (red) for the Harmonic oscillator (1).



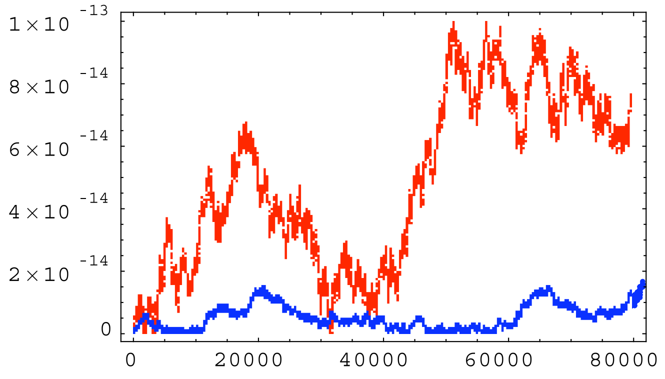
Algorithm 1 for a 15-stage method corresponds to $n = 15 \times 2 \times 10^6 = 3 \times 10^7$.

In the incremental Algorithm 3 the internal stages are all of the order of the step size and the only significant rounding error occurs at the end of each integration step; thus $n = 2 \times 10^6$, which is in good agreement with the observed improvement.

This shows that for Algorithm 3, with sufficiently small step sizes, the rounding error growth is independent of the number of stages of the method, which is particularly advantageous for high order.

With Compensated Summation

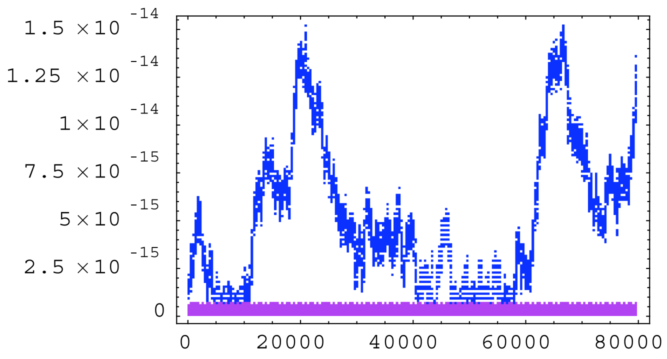
The relative error in the Hamiltonian is displayed here for the increment formulation in Algorithm 3 without compensated summation (red) and with compensated summation (blue) for the Harmonic oscillator (1).



Using compensated summation with Algorithm 3, the error growth appears to satisfy a random walk with deviation $h\epsilon$ so that it has been reduced by a factor proportional to the step size.

Arbitrary Precision

The relative error in the Hamiltonian is displayed here for the increment formulation in Algorithm 3 with compensated summation using IEEE double-precision arithmetic (blue) and with 32-decimal-digit software arithmetic (purple) for the Harmonic oscillator (1).



However, the solution obtained using software arithmetic is around an order of magnitude slower than machine arithmetic, so strategies to reduce the effect of roundoff error are worthwhile.

Examples

Electrostatic Wave

Here is a non-autonomous Hamiltonian (it has a time-dependent potential) that models n perturbing electrostatic waves, each with the same wave number and amplitude, but different temporal frequencies ω_i (see [CR91]).

$$H(p, q) = \frac{p^2}{2} + \frac{q^2}{2} + \epsilon \sum_{i=1}^{n} (\cos(q - \omega_i)). \tag{1}$$

This defines a differential system from the Hamiltonian (1) for dimension $n = 3$ with frequencies $\omega_1 = 7$, $\omega_2 = 14$, $\omega_3 = 21$.

```
In[27]:= H = p[t]^2 / 2 + q[t]^2 / 2 + Sum[Cos[q[t] - 7 i t], {i, 3}];
         eqs = {p'[t] == -D[H, q[t]], q'[t] == D[H, p[t]]};
         ics = {p[0] == 0, q[0] == 4483 / 400};
         vars = {q[t], p[t]};
         time = {t, 0, 10000 × 2 π};
         step = 2 π / 105;
```

A general technique for computing Poincaré sections is described within "EventLocator Method for NDSolve". Specifying an empty list for the variables avoids storing all the data of the numerical integration.

The integration is carried out with a symplectic method with a relatively large number of steps and the solutions are collected using `Sow` and `Reap` when the time is a multiple of 2π.

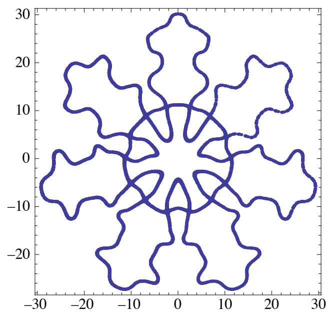
The "`Direction`" option of "`EventLocator`" is used to control the sign in the detection of the event.

```
In[33]:= sprkmethod = {"SymplecticPartitionedRungeKutta",
            "DifferenceOrder" → 4, "PositionVariables" -> {q[t]}};

         sprkdata =
           Block[{k = 1},
             Reap[
              NDSolve[{eqs, ics}, {}, time,
                Method → {"EventLocator", "Direction" → 1, "Event" :→ (t - 2 k Pi),
                   "EventAction" :→ (k++; Sow[{q[t], p[t]}]), Method → sprkmethod},
                StartingStepSize → step, MaxSteps → Infinity
               ];
             ]
           ];

            NDSolve::noout: No functions were specified for output from NDSolve.
```

This displays the solution at time intervals of 2π.

```
In[35]:= ListPlot[sprkdata[[-1, 1]], Axes → False,
          Frame → True, AspectRatio → 1, PlotRange → All]
```

Out[35]=

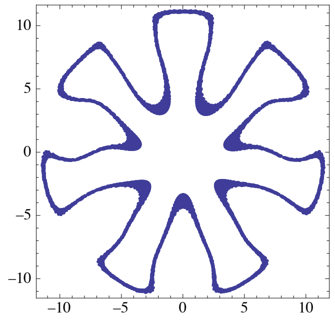


For comparison a Poincaré section is also computed using an explicit Runge-Kutta method of the same order.

```
In[36]:= rkmethod = {"FixedStep", Method → {"ExplicitRungeKutta", "DifferenceOrder" → 4}};

         rkdata =
           Block[{k = 1},
            Reap[
             NDSolve[{eqs, ics}, {}, time,
               Method → {"EventLocator", "Direction" → 1, "Event" :→ (t - 2 k Pi),
                  "EventAction" :→ (k++; Sow[{q[t], p[t]}]), Method → rkmethod},
               StartingStepSize → step, MaxSteps → Infinity
              ];
            ]
           ];

            NDSolve::noout : No functions were specified for output from NDSolve.
```

Fine structural details are clearly resolved in a less satisfactory way with this method.

```
In[38]:= ListPlot[rkdata[[-1, 1]], Axes → False,
          Frame → True, AspectRatio → 1, PlotRange → All]
```

Out[38]=


Toda Lattice

The Toda lattice models particles on a line interacting with pairwise exponential forces and is governed by the Hamiltonian:

$$H(p, q) = \sum_{k=1}^{n} \left(\frac{1}{2} p_k{}^2 + (\exp(q_{k+1} - q_k) - 1) \right).$$

Consider the case when periodic boundary conditions $q_{n+1} = q_1$ are enforced.

The Toda lattice is an example of an *isospectral flow*. Using the notation

$$a_k = -\frac{1}{2} p_k,\ b_k = \frac{1}{2} \exp\left(\frac{1}{2} (q_{k+1} - q_k) \right)$$

then the eigenvalues of the following matrix are conserved quantities of the flow:

$$L = \begin{pmatrix} a_1 & b_1 & & & & b_n \\ b_1 & a_2 & b_2 & & 0 & \\ & b_2 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & b_{n-2} & a_{n-1} & b_{n-1} \\ b_n & & & & b_{n-1} & a_n \end{pmatrix}.$$

Define the input for the Toda lattice problem for $n = 3$.

```
In[39]:= n = 3;

         periodicRule = {q_{n+1}[t] → q_1[t]};

         H = Sum[p_k[t]^2/2 + (Exp[q_{k+1}[t] - q_k[t]] - 1), {k, 1, n}] /. periodicRule;

         eigenvalueRule = {a_k_[t] :→ -p_k[t] / 2, b_k_[t] :→ 1 / 2 Exp[1 / 2 (q_{k+1}[t] - q_k[t])]};

         L = {{a_1[t], b_1[t], b_3[t]}, {b_1[t], a_2[t], b_2[t]}, {b_3[t], b_2[t], a_3[t]}} /. eigenvalueRule /. periodicRule;

         eqs = {q_1'[t] == D[H, p_1[t]], q_2'[t] == D[H, p_2[t]], q_3'[t] == D[H, p_3[t]],
            p_1'[t] == -D[H, q_1[t]], p_2'[t] == -D[H, q_2[t]], p_3'[t] == -D[H, q_3[t]]};
         ics = {q_1[0] == 1, q_2[0] == 2, q_3[0] == 4, p_1[0] == 0, p_2[0] == 1, p_3[0] == 1 / 2};
         eqs = {eqs, ics};
         vars = {q_1[t], q_2[t], q_3[t], p_1[t], p_2[t], p_3[t]};
         time = {t, 0, 50};
```

$$H = \sum_{k=1}^{n} \left(\frac{p_k[t]^2}{2} + (\mathrm{Exp}[q_{k+1}[t] - q_k[t]] - 1) \right)$$

$$L = \begin{pmatrix} a_1[t] & b_1[t] & b_3[t] \\ b_1[t] & a_2[t] & b_2[t] \\ b_3[t] & b_2[t] & a_3[t] \end{pmatrix}$$

Define a function to compute the eigenvalues of a matrix of numbers, sorted in increasing order. This avoids computing the eigenvalues symbolically.

```
In[49]:= NumberMatrixQ[m_] := MatrixQ[m, NumberQ];
         NumberEigenvalues[m_?NumberMatrixQ] := Sort[Eigenvalues[m]];
```

Integrate the equations for the Toda lattice using the "ExplicitMidpoint" method.

```
In[51]:= emsol =
           NDSolve[eqs, vars, time, Method → "ExplicitMidpoint", StartingStepSize → 1 / 10];
```

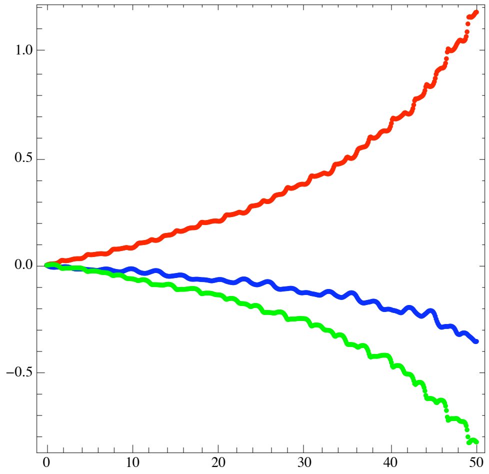
The absolute error in the eigenvalues is now plotted throughout the integration interval.

Options are used to specify the dimension of the result of `NumberEigenvalues` (since it is not an explicit list) and that the absolute error specified using `InvariantErrorFunction` should include the sign of the error (the default uses `Abs`).

The eigenvalues are clearly not conserved by the “ExplicitMidpoint“ method.

```
In[52]:= InvariantErrorPlot[NumberEigenvalues[L],
          vars, t, emsol, InvariantErrorFunction → (#1 - #2 &),
          InvariantDimensions → {n}, PlotStyle → {Red, Blue, Green}]
```

Out[52]=



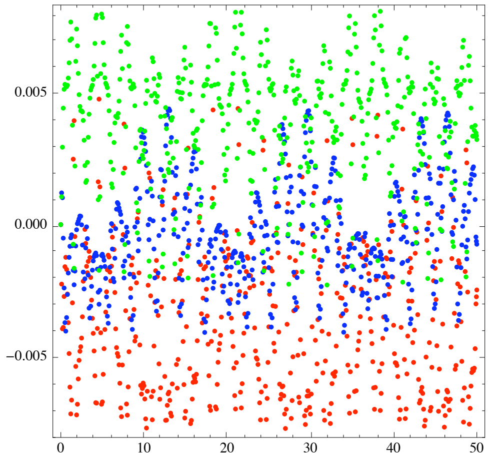
Integrate the equations for the Toda lattice using the “SymplecticPartitionedRungeKutta“ method.

```
In[53]:= sprksol = NDSolve[eqs, vars, time,
             Method → {“SymplecticPartitionedRungeKutta“, DifferenceOrder → 2,
               “PositionVariables“ → {q₁[t], q₂[t], q₃[t]}}, StartingStepSize → 1 / 10];
```

The error in the eigenvalues now remains bounded throughout the integration.

```
In[54]:= InvariantErrorPlot[NumberEigenvalues[L],
          vars, t, sprksol, InvariantErrorFunction → (#1 - #2 &),
          InvariantDimensions → {n}, PlotStyle → {Red, Blue, Green}]
```

Out[54]=



Some recent work on numerical methods for isospectral flows can be found in [CIZ97], [CIZ99], [DLP98a], and [DLP98b].

Available Methods

Default Methods

The following table lists the current default choice of SPRK methods.

Order	f evaluations	Method	Symmetric	FSAL
1	1	Symplectic Euler	No	No
2	1	Symplectic pseudo Leapfrog	Yes	Yes
3	3	McLachlan and Atela [MA92]	No	No
4	5	Suzuki [S90]	Yes	Yes
6	11	Sofroniou and Spaletta [SS05]	Yes	Yes
8	19	Sofroniou and Spaletta [SS05]	Yes	Yes
10	35	Sofroniou and Spaletta [SS05]	Yes	Yes

Unlike the situation for explicit Runge-Kutta methods, the coefficients for high-order SPRK methods are only given numerically in the literature. Yoshida [Y90] only gives coefficients accurate to 14 decimal digits of accuracy for example.

Since `NDSolve` also works for arbitrary precision, you need a process for obtaining the coefficients to the same precision as that to be used in the solver.

When the closed form of the coefficients is not available, the order equations for the symmetric composition coefficients can be refined in arbitrary precision using `FindRoot`, starting from the known machine-precision solution.

Alternative Methods

Due to the modular design of the new `NDSolve` framework it is straightforward to add an alternative method and use that instead of one of the default methods.

Several checks are made before any integration is carried out:

- The two vectors of coefficients should be nonempty, the same length, and numerical approximations should yield number entries of the correct precision.
- Both coefficient vectors should sum to unity so that they yield a consistent (order 1) method.

Example

Select the perturbed Kepler problem.

```
In[55]:= system = GetNDSolveProblem["PerturbedKepler"];
         time = {T, 0, 290};
         step = 1 / 25;
```

Define a function for computing a numerical approximation to the coefficients for a fourth-order method of Forest and Ruth [FR90], Candy and Rozmus [CR91], and Yoshida [Y90].

```
In[58]:= YoshidaCoefficients[4, prec_] :=
           N[
            {{Root[-1 + 12 * #1 - 48 * #1^2 + 48 * #1^3 &, 1, 0],
              Root[1 - 24 * #1^2 + 48 * #1^3 &, 1, 0], Root[1 - 24 * #1^2 + 48 * #1^3 &, 1, 0],
              Root[-1 + 12 * #1 - 48 * #1^2 + 48 * #1^3 &, 1, 0]},
             {Root[-1 + 6 * #1 - 12 * #1^2 + 6 * #1^3 &, 1, 0], Root[1 - 3 * #1 + 3 * #1^2 + 3 * #1^3 &,
               1, 0], Root[-1 + 6 * #1 - 12 * #1^2 + 6 * #1^3 &, 1, 0], 0}},
            prec];
```

Here are machine-precision approximations for the coefficients.

```
In[59]:= YoshidaCoefficients[4, MachinePrecision]

Out[59]= {{0.675604, -0.175604, -0.175604, 0.675604}, {1.35121, -1.70241, 1.35121, 0.}}
```

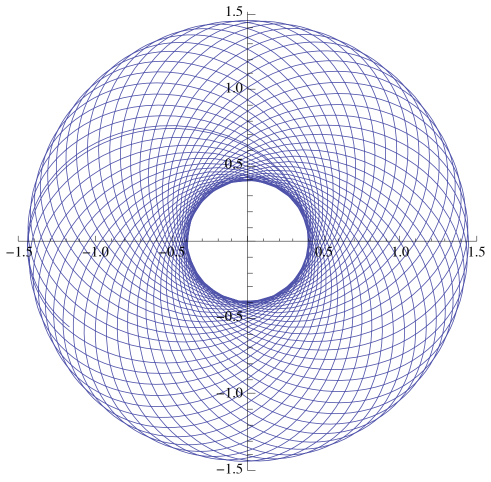
This invokes the symplectic partitioned Runge-Kutta solver using Yoshida's coefficients.

```
In[60]:= Yoshida4 =
           {"SymplecticPartitionedRungeKutta", "Coefficients" → YoshidaCoefficients,
            "DifferenceOrder" → 4, "PositionVariables" → {Y1[T], Y2[T]}};

         Yoshida4sol = NDSolve[system, time,
            Method → Yoshida4, StartingStepSize → step, MaxSteps → Infinity];
```

This plots the solution of the position variables, or coordinates, in the Hamiltonian formulation.

```
In[62]:= ParametricPlot[Evaluate[{Y1[T], Y2[T]} /. Yoshida4sol], Evaluate[time]]
```

Out[62]=



Automatic Order Selection

Given that a variety of methods of different orders are available, it is useful to have a means of automatically selecting an appropriate method. In order to accomplish this we need a measure of work for each method.

A reasonable measure of work for an SPRK method is the number of stages s (or $s-1$ if the method is FSAL).

Definition (Work per unit step)

Given a step size h_k and a work estimate $\mathcal{A}_k$ for one integration step with a method of order k, the work per unit step is given by $\mathcal{W}_k = \mathcal{A}_k / h_k$.

Let Π be a nonempty set of method orders, Π_k denote the kth element of Π, and $|\Pi|$ denote the cardinality (number of elements).

A comparison of work for the default SPRK methods gives $\Pi = \{2, 3, 4, 6, 8, 10\}$.

A prerequisite is a procedure for estimating the starting step h_k of a numerical method of order k (see for example [GSB87] or [HNW93]).

The first case to be considered is when the starting step estimate h can be freely chosen. By bootstrapping from low order, the following algorithm finds the order that locally minimizes the work per unit step.

Algorithm 4 (h free)

```
Set $W = \infty$

for $k = 1, \ldots, |\Pi|$
    compute $h_{\Pi_k}$
    if $\mathcal{W} > \mathcal{A}_{\Pi_k} / h_{\Pi_k}$ set $\mathcal{W} = \mathcal{A}_{\Pi_k} / h_{\Pi_k}$
    else if $k = |\Pi|$ return $\Pi_k$
    else return $\Pi_{k-1}$.
```

The second case to be considered is when the starting step estimate h is given. The following algorithm then gives the order of the method that minimizes the computational cost while satisfying given absolute and relative local error tolerances.

Algorithm 5 (h specified)

```
for $k = 1, \ldots, |\Pi|$
    compute $h_{\Pi_k}$
    if $h_{\Pi_k} > h$ or $k = |\Pi|$ return $\Pi_k$.
```

Algorithms 4 and 5 are heuristic since the optimal step size and order may change through the integration, although symplectic integration often involves fixed choices. Despite this, both algorithms incorporate salient integration information, such as local error tolerances, system dimension, and initial conditions, to avoid poor choices.

Examples

Consider Kepler's problem that describes the motion in the configuration plane of a material point that is attracted toward the origin with a force inversely proportional to the square of the distance:

$$H(p, q) = \frac{1}{2}\left(p_1{}^2 + p_2{}^2\right) - \frac{1}{\sqrt{q_1{}^2+q_2{}^2}}. \tag{1}$$

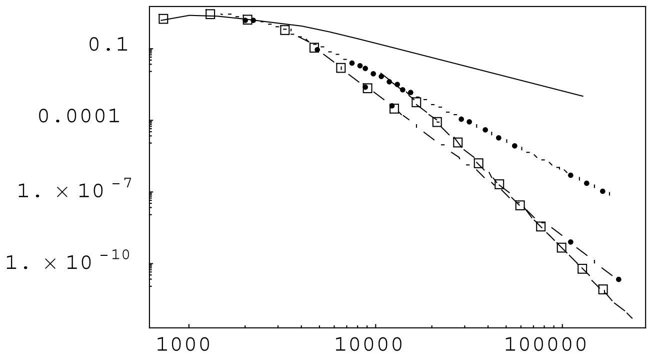
For initial conditions take

$$p_1(0) = 0,\ p_2(0) = \sqrt{\frac{1+e}{1-e}}\ , q_1(0) = 1 - e,\ q_2(0) = 0$$

with eccentricity $e = 3/5$.

Algorithm 4

The following figure shows the methods chosen automatically at various tolerances for the Kepler problem (1) according to Algorithm 4 on a log-log scale of maximum absolute phase error versus work.

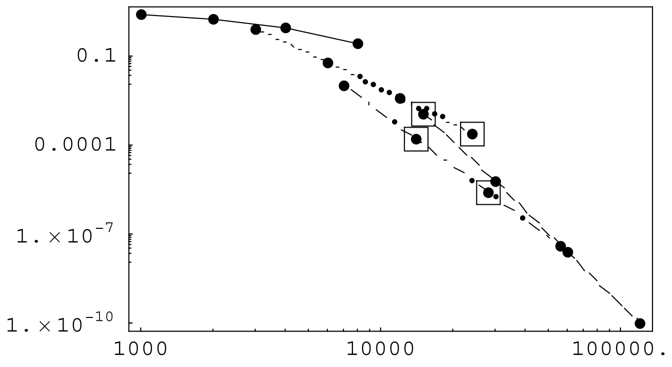


It can be observed that the algorithm does a reasonable job of staying near the optimal method, although it switches over to the 8th-order method slightly earlier than necessary.

This can be explained by the fact that the starting step size routine is based on low-order derivative estimation and this may not be ideal for selecting high-order methods.

Algorithm 5

The following figure shows the methods chosen automatically with absolute local error tolerance of 10^{-9} and step sizes 1/16, 1/32, 1/64, 1/128 for the Kepler problem (1) according to Algorithm 5 on a log-log scale of maximum absolute phase error versus work.



With the local tolerance and step size fixed the code can only choose the order of the method.

For large step sizes a high-order method is selected, whereas for small step sizes a low-order method is selected. In each case the method chosen minimizes the work to achieve the given tolerance.

Option Summary

option name	default value	
`"Coefficients"`	`"SymplecticPartitionedRungeKuttaCoefficients"`	specify the coefficients of the symplectic partitioned Runge-Kutta method
`"DifferenceOrder"`	`Automatic`	specify the order of local accuracy of the method
`"PositionVariables"`	`{}`	specify a list of the position variables in the Hamiltonian formulation

Options of the method “`SymplecticPartitionedRungeKutta`“.

Controller Methods

"Composition" and "Splitting" Methods for NDSolve

Introduction

In some cases it is useful to split the differential system into subsystems and solve each subsystem using appropriate integration methods. Recombining the individual solutions often allows certain dynamical properties, such as volume, to be conserved. More information on splitting and composition can be found in [MQ02, HLW02], and specific aspects related to `NDSolve` are discussed in [SS05, SS06].

Definitions

Of concern are initial value problems $y'(t) = f(y(t))$, where $y(0) = y_0 \in \mathbb{R}^n$.

"Composition"

Composition is a useful device for raising the order of a numerical integration scheme.

In contrast to the Aitken-Neville algorithm used in extrapolation, composition can conserve geometric properties of the base integration method (e.g. symplecticity).

Let $\Phi^{(i)}_{f,\gamma_i h}$ be a basic integration method that takes a step of size $\gamma_i h$ with $\gamma_1, \ldots, \gamma_s$ given real numbers.

Then the s-stage composition method $\Psi_{f,h}$ is given by

$$\Psi_{f,h} = \Phi^{(s)}_{f,\gamma_s h} \circ \cdots \circ \Phi^{(1)}_{f,\gamma_1 h}.$$

Often interest is in composition methods $\Psi_{f,h}$ that involve the same base method $\Phi = \Phi^{(i)}$, $i = 1, \ldots, s$.

An interesting special case is symmetric composition: $\gamma_i = \gamma_{s-i+1}$, $i = 1, \ldots, \lfloor s/2 \rfloor$.

The most common types of composition are:

- Symmetric composition of symmetric second-order methods
- Symmetric composition of first-order methods (e.g. a method Φ with its adjoint Φ^*)
- Composition of first-order methods

"Splitting"

An s-stage splitting method is a generalization of a composition method in which f is broken up in an additive fashion:

$$f = f_1 + \cdots + f_k, \quad k \leq s.$$

The essential point is that there can often be computational advantages in solving problems involving f_i instead of f.

An s-stage splitting method is a composition of the form

$$\Psi_{f,h} = \Phi^{(s)}_{f_s,\gamma_s h} \circ \cdots \circ \Phi^{(1)}_{f_1,\gamma_1 h},$$

with $f_1, \ldots, f_s$ not necessarily distinct.

Each base integration method now only solves part of the problem, but a suitable composition can still give rise to a numerical scheme with advantageous properties.

If the vector field f_i is integrable, then the exact solution or flow $\varphi_{f_i,h}$ can be used in place of a numerical integration method.

A splitting method may also use a mixture of flows and numerical methods.

An example is Lie-Trotter splitting [T59]:

Split $f = f_1 + f_2$ with $\gamma_1 = \gamma_2 = 1$; then $\Psi_{f,h} = \varphi^{(2)}_{f_2,h} \circ \varphi^{(1)}_{f_1,h}$ yields a first-order integration method.

Computationally it can be advantageous to combine flows using the group property

$$\varphi_{f_i,h_1+h_2} = \varphi_{f_i,h_2} \circ \varphi_{f_i,h_1}.$$

Implementation

Several changes to the new `NDSolve` framework were needed in order to implement splitting and composition methods.

- Allow a method to call an arbitrary number of submethods.
- Add the ability to pass around a function for numerically evaluating a subfield, instead of the entire vector field.
- Add a `"LocallyExact"` method to compute the flow; analytically solve a subsystem and advance the (local) solution numerically.
- Add cache data for identical methods to avoid repeated initialization. Data for numerically evaluating identical subfields is also cached.

A simplified input syntax allows omitted vector fields and methods to be filled in cyclically. These must be defined unambiguously:

$\{f_1, f_2, f_1, f_2\}$ can be input as $\{f_1, f_2\}$.

$\{f_1, f_2, f_3, f_2, f_1\}$ cannot be input as $\{f_1, f_2, f_3\}$ since this corresponds to $\{f_1, f_2, f_3, f_1, f_2\}$.

Nested Methods

The following example constructs a high-order splitting method from a low-order splitting using `“Composition“`.

$$\begin{tikzcd}
 & & & \boxed{\texttt{“LocallyExact“}\ f_1} \\
 & & \boxed{\texttt{“Splitting“}\ f = f_1 + f_2} \arrow[ur] \arrow[r] \arrow[dr] & \boxed{\texttt{ImplicitMidpoint}\ f_2} \\
 & & & \boxed{\texttt{“LocallyExact“}\ f_1} \\
 & \vdots & & \vdots \\
 & & & \boxed{\texttt{“LocallyExact“}\ f_1} \\
\boxed{\texttt{NDSolve}} \arrow[r] & \boxed{\texttt{“Composition“}} \arrow[uuuur] \arrow[r] \arrow[ddddr] & \boxed{\texttt{“Splitting“}\ f = f_1 + f_2} \arrow[ur] \arrow[r] \arrow[dr] & \boxed{\texttt{ImplicitMidpoint}\ f_2} \\
 & & & \boxed{\texttt{“LocallyExact“}\ f_1} \\
 & \vdots & & \vdots \\
 & & & \boxed{\texttt{“LocallyExact“}\ f_1} \\
 & & \boxed{\texttt{“Splitting“}\ f = f_1 + f_2} \arrow[ur] \arrow[r] \arrow[dr] & \boxed{\texttt{ImplicitMidpoint}\ f_2} \\
 & & & \boxed{\texttt{“LocallyExact“}\ f_1}
\end{tikzcd}$$

Simplification

A more efficient integrator can be obtained in the previous example using the group property of flows and calling the `“Splitting“` method directly.

$$\begin{tikzcd}
 & & \boxed{\texttt{“LocallyExact“}\ f_1} \\
 & & \boxed{\texttt{ImplicitMidpoint}\ f_2} \\
 & \vdots & \vdots \\
 & & \boxed{\texttt{“LocallyExact“}\ f_1} \\
\boxed{\texttt{NDSolve}} \arrow[r] & \boxed{\texttt{“Splitting“}\ f = f_1 + f_2} \arrow[uuuur] \arrow[r] \arrow[ddddr] & \boxed{\texttt{ImplicitMidpoint}\ f_2} \\
 & & \boxed{\texttt{“LocallyExact“}\ f_1} \\
 & \vdots & \vdots \\
 & & \boxed{\texttt{ImplicitMidpoint}\ f_2} \\
 & & \boxed{\texttt{“LocallyExact“}\ f_1}
\end{tikzcd}$$

Examples

The following examples will use a second-order symmetric splitting known as the Strang splitting [S68], [M68]. The splitting coefficients are automatically determined from the structure of the equations.

This defines a method known as symplectic leapfrog in terms of the method `"SymplecticPartitionedRungeKutta"`.

```
In[2]:= SymplecticLeapfrog = {"SymplecticPartitionedRungeKutta",
           "DifferenceOrder" → 2, "PositionVariables" :> qvars};
```

Load a package with some useful example problems.

```
In[3]:= Needs["DifferentialEquations`NDSolveProblems`"];
```

Symplectic Splitting

Symplectic Leapfrog

`"SymplecticPartitionedRungeKutta"` is an efficient method for solving separable Hamiltonian systems $H(p, q) = T(p) + V(q)$ with favorable long-time dynamics.

`"Splitting"` is a more general-purpose method, but it can be used to construct partitioned symplectic methods (though it is somewhat less efficient than `"SymplecticPartitionedRungeKutta"`).

Consider the harmonic oscillator that arises from a linear differential system that is governed by the separable Hamiltonian $H(p, q) = p^2/2 + q^2/2$.

```
In[5]:= system = GetNDSolveProblem["HarmonicOscillator"]

Out[5]= NDSolveProblem[{{Y₁′[T] == Y₂[T], Y₂′[T] == -Y₁[T]},
          {Y₁[0] == 1, Y₂[0] == 0}, {Y₁[T], Y₂[T]}, {T, 0, 10}, {}, {1/2 (Y₁[T]^2 + Y₂[T]^2)}}]
```

Split the Hamiltonian vector field into independent components governing momentum and position. This is done by setting the relevant right-hand sides of the equations to zero.

```
In[6]:= eqs = system["System"];
        Y1 = eqs;
        Part[Y1, 1, 2] = 0;
        Y2 = eqs;
        Part[Y2, 2, 2] = 0;
```

This composition of weighted (first-order) Euler integration steps corresponds to the symplectic (second-order) leapfrog method.

```
In[11]:= tfinal = 1;
         time = {T, 0, tfinal};
         qvars = {Subscript[Y, 1][T]};
         splittingsol = NDSolve[system, time, StartingStepSize → 1 / 10,
            Method → {"Splitting", "DifferenceOrder" → 2, "Equations" → {Y1, Y2, Y1},
              "Method" → {"ExplicitEuler", "ExplicitEuler", "ExplicitEuler"}}]

Out[14]= {{Y₁[T] → InterpolatingFunction[{{0., 1.}}, <>][T],
           Y₂[T] → InterpolatingFunction[{{0., 1.}}, <>][T]}}
```

The method `"ExplicitEuler"` could only have been specified once, since the second and third instances would have been filled in cyclically.

This is the result at the end of the integration step.

```
In[15]:= InputForm[splittingsol /. T → tfinal]

Out[15]//InputForm= {{Subscript[Y, 1][1] -> 0.5399512509335085, Subscript[Y, 2][1] -> -0.8406435124348495}}
```

This invokes the built-in integration method corresponding to the symplectic leapfrog integrator.

```
In[16]:= sprksol =
          NDSolve[system, time, StartingStepSize → 1 / 10, Method → SymplecticLeapfrog]

Out[16]= {{Y1[T] → InterpolatingFunction[{{0., 1.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 1.}}, <>][T]}}
```

The result at the end of the integration step is identical to the result of the splitting method.

```
In[17]:= InputForm[sprksol /. T → tfinal]

Out[17]//InputForm= {{Subscript[Y, 1][1] -> 0.5399512509335085, Subscript[Y, 2][1] -> -0.8406435124348495}}
```

Composition of Symplectic Leapfrog

This takes the symplectic leapfrog scheme as the base integration method and constructs a fourth-order symplectic integrator using a symmetric composition of Ruth-Yoshida [Y90].

```
In[18]:= YoshidaCoefficients =
           RootReduce[{1 / (2 - 2 ^ (1 / 3)), -2 ^ (1 / 3) / (2 - 2 ^ (1 / 3)), 1 / (2 - 2 ^ (1 / 3))}];

         YoshidaCompositionCoefficients[4, p_] := N[YoshidaCoefficients, p];

         splittingsol = NDSolve[system, time, StartingStepSize → 1 / 10,
            Method → {"Composition", "Coefficients" → YoshidaCompositionCoefficients,
              "DifferenceOrder" → 4, "Method" → {SymplecticLeapfrog}}]

Out[20]= {{Y1[T] → InterpolatingFunction[{{0., 1.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 1.}}, <>][T]}}
```

This is the result at the end of the integration step.

```
In[21]:= InputForm[splittingsol /. T → tfinal]

Out[21]//InputForm= {{Subscript[Y, 1][1] -> 0.5403078808898406, Subscript[Y, 2][1] -> -0.8414706295697821}}
```

This invokes the built-in symplectic integration method using coefficients for the fourth-order methods of Ruth and Yoshida.

```
In[22]:= SPRK4[4, prec_] := N[{{Root[-1 + 12 * #1 - 48 * #1^2 + 48 * #1^3 &, 1, 0],
            Root[1 - 24 * #1^2 + 48 * #1^3 &, 1, 0], Root[1 - 24 * #1^2 + 48 * #1^3 &, 1, 0],
            Root[-1 + 12 * #1 - 48 * #1^2 + 48 * #1^3 &, 1, 0]},
           {Root[-1 + 6 * #1 - 12 * #1^2 + 6 * #1^3 &, 1, 0], Root[1 - 3 * #1 + 3 * #1^2 + 3 * #1^3 &,
             1, 0], Root[-1 + 6 * #1 - 12 * #1^2 + 6 * #1^3 &, 1, 0], 0}}, prec];

          sprksol = NDSolve[system, time, StartingStepSize → 1 / 10,
            Method → {"SymplecticPartitionedRungeKutta", "Coefficients" → SPRK4,
              "DifferenceOrder" → 4, "PositionVariables" → qvars}]

Out[23]= {{Y1[T] → InterpolatingFunction[{{0., 1.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 1.}}, <>][T]}}
```

The result at the end of the integration step is identical to the result of the composition method.

```
In[24]:= InputForm[sprksol /. T → tfinal]

Out[24]//InputForm= {{Subscript[Y, 1][1] -> 0.5403078808898406, Subscript[Y, 2][1] -> -0.8414706295697821}}
```

Hybrid Methods

While a closed-form solution often does not exist for the entire vector field, in some cases it is possible to analytically solve a system of differential equations for part of the vector field.

When a solution can be found by `DSolve`, direct numerical evaluation can be used to locally advance the solution.

This idea is implemented in the method `"LocallyExact"`.

Harmonic Oscillator Test Example

This example checks that the solution for the exact flows of split components of the harmonic oscillator equations is the same as applying Euler's method to each of the split components.

```
In[25]:= system = GetNDSolveProblem["HarmonicOscillator"];
         eqs = system["System"];
         Y1 = eqs;
         Part[Y1, 1, 2] = 0;
         Y2 = eqs;
         Part[Y2, 2, 2] = 0;
         tfinal = 1;
         time = {T, 0, tfinal};

In[33]:= solexact = NDSolve[system, time, StartingStepSize → 1 / 10,
             Method → {NDSolve`Splitting, "DifferenceOrder" → 2,
               "Equations" → {Y1, Y2, Y1}, "Method" → {"LocallyExact"}}];

In[34]:= InputForm[solexact /. T → 1]

Out[34]//InputForm= {{Subscript[Y, 1][1] -> 0.5399512509335085, Subscript[Y, 2][1] -> -0.8406435124348495}}
```

```
In[37]:= soleuler = NDSolve[system, time, StartingStepSize → 1 / 10,
             Method → {NDSolve`Splitting, "DifferenceOrder" → 2,
               "Equations" → {Y1, Y2, Y1}, "Method" → {"ExplicitEuler"}}];

         InputForm[soleuler /. T → tfinal]
Out[38]//InputForm= {{Subscript[Y, 1][1] -> 0.5399512509335085, Subscript[Y, 2][1] -> -0.8406435124348495}}
```

Hybrid Numeric-Symbolic Splitting Methods (ABC Flow)

Consider the Arnold, Beltrami, and Childress flow, a widely studied model for volume-preserving three-dimensional flows.

```
In[39]:= system = GetNDSolveProblem["ArnoldBeltramiChildress"]
```

Out[39]= $\text{NDSolveProblem}\Big[\Big\{\Big\{Y_1'[T] == \frac{3}{4}\text{Cos}[Y_2[T]] + \text{Sin}[Y_3[T]],\ Y_2'[T] == \text{Cos}[Y_3[T]] + \text{Sin}[Y_1[T]],\ Y_3'[T] == \text{Cos}[Y_1[T]] + \frac{3}{4}\text{Sin}[Y_2[T]]\Big\},\ \Big\{Y_1[0] == \frac{1}{4},\ Y_2[0] == \frac{1}{3},\ Y_3[0] == \frac{1}{2}\Big\},\ \{Y_1[T], Y_2[T], Y_3[T]\},\ \{T, 0, 100\},\ \{\},\ \{\}\Big\}\Big]$

When applied directly, a volume-preserving integrator would not in general preserve symmetries. A symmetry-preserving integrator, such as the implicit midpoint rule, would not preserve volume.

This defines a splitting of the system by setting some of the right-hand side components to zero.

```
In[40]:= eqs = system["System"];
         Y1 = eqs;
         Part[Y1, 2, 2] = 0;
         Y2 = eqs;
         Part[Y2, {1, 3}, 2] = 0;

In[45]:= Y1
```

Out[45]= $\Big\{Y_1'[T] == \frac{3}{4}\text{Cos}[Y_2[T]] + \text{Sin}[Y_3[T]],\ Y_2'[T] == 0,\ Y_3'[T] == \text{Cos}[Y_1[T]] + \frac{3}{4}\text{Sin}[Y_2[T]]\Big\}$

```
In[46]:= Y2
```

Out[46]= $\{Y_1'[T] == 0,\ Y_2'[T] == \text{Cos}[Y_3[T]] + \text{Sin}[Y_1[T]],\ Y_3'[T] == 0\}$

The system for `Y1` is solvable exactly by `DSolve` so that you can use the `"LocallyExact"` method.

`Y2` is not solvable, however, so you need to use a suitable numerical integrator in order to obtain the desired properties in the splitting method.

This defines a method for computing the implicit midpoint rule in terms of the built-in “ImplicitRungeKutta“ method.

```
In[47]:= ImplicitMidpoint = {“FixedStep“, Method → {“ImplicitRungeKutta“, “Coefficients“ →
           “ImplicitRungeKuttaGaussCoefficients“, “DifferenceOrder“ → 2,
          ImplicitSolver → {FixedPoint, AccuracyGoal → MachinePrecision,
            PrecisionGoal → MachinePrecision, “IterationSafetyFactor“ → 1}}};
```

This defines a second-order, volume-preserving, reversing symmetry-group integrator [MQ02].

```
In[48]:= splittingsol = NDSolve[system,
           StartingStepSize → 1 / 10,
           Method → {“Splitting“, “DifferenceOrder“ → 2,
             “Equations“ → {Y2, Y1, Y2},
             “Method“ → {“LocallyExact“, ImplicitMidpoint, “LocallyExact“}}]

Out[48]= {{Y1[T] → InterpolatingFunction[{{0., 100.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 100.}}, <>][T],
           Y3[T] → InterpolatingFunction[{{0., 100.}}, <>][T]}}
```

Lotka-Volterra Equations

Various numerical integrators for this system are compared within "Numerical Methods for Solving the Lotka-Volterra Equations".

Euler's Equations

Various numerical integrators for Euler's equations are compared within "Rigid Body Solvers".

Non-Autonomous Vector Fields

Consider the Duffing oscillator, a forced planar non-autonomous differential system.

```
In[49]:= system = GetNDSolveProblem[“DuffingOscillator“]
```

Out[49]=
$$\text{NDSolveProblem}\left[\left\{\left\{Y_1'[T] == Y_2[T],\ Y_2'[T] == \frac{3\,\text{Cos}[T]}{10} + Y_1[T] - Y_1[T]^3 + \frac{Y_2[T]}{4}\right\},\ \{Y_1[0] == 0,\ Y_2[0] == 1\},\ \{Y_1[T],\ Y_2[T]\},\ \{T,\ 0,\ 10\},\ \{\},\ \{\}\right\}\right]$$

This defines a splitting of the system.

In[50]:=
$$\mathbf{Y1} = \left\{Y_1'[T] == Y_2[T],\ Y_2'[T] == \frac{Y_2[T]}{4}\right\};$$
$$\mathbf{Y2} = \left\{Y_1'[T] == 0,\ Y_2'[T] == \frac{3\,\text{Cos}[T]}{10} + Y_1[T] - Y_1[T]^3\right\};$$

The splitting of the time component among the vector fields is ambiguous, so the method issues an error message.

```
In[52]:= splittingsol = NDSolve[system, StartingStepSize → 1 / 10,
            Method → {"Splitting", "DifferenceOrder" → 2,
              "Equations" → {Y2, Y1, Y1}, "Method" → {"LocallyExact"}}]
```

NDSolve::spltdep :
The differential system $\left\{0, \frac{3\,\text{Cos}[T]}{10} + Y_1[T] - Y_1[T]^3\right\}$ in the method Splitting depends on T which is ambiguous. The differential system should be in autonomous form. ≫

NDSolve::initf : The initialization of the method NDSolve`Splitting failed.

```
Out[52]= {{Y₁[T] → InterpolatingFunction[{{0., 0.}}, <>][T],
           Y₂[T] → InterpolatingFunction[{{0., 0.}}, <>][T]}}
```

The equations can be extended by introducing a new "dummy" variable `Z[T]` such that `Z[T] == T` and specifying how it should be distributed in the split differential systems.

$$\texttt{In[53]:=}\quad Y1 = \left\{Y_1'[T] == Y_2[T],\; Y_2'[T] == \frac{Y_2[T]}{4},\; Z'[T] == 1\right\};$$

$$Y2 = \left\{Y_1'[T] == 0,\; Y_2'[T] == \frac{3\,\text{Cos}[Z[T]]}{10} + Y_1[T] - Y_1[T]^3,\; Z'[T] == 0\right\};$$

```
eqs = Join[system["System"], {Z'[T] == 1}];
ics = Join[system["InitialConditions"], {Z[0] == 1}];
vars = Join[system["DependentVariables"], {Z[T]}];
time = system["TimeData"];
```

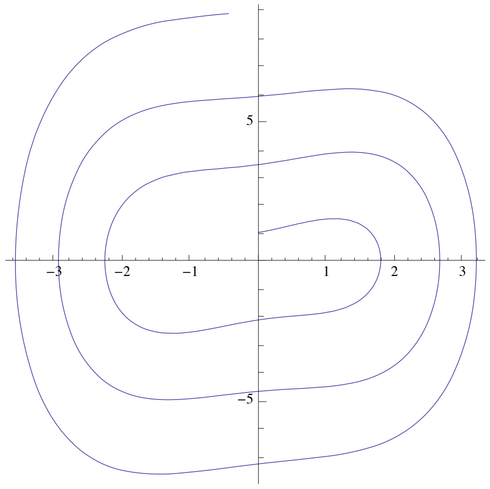
This defines a geometric splitting method that satisfies $\lambda_1 + \lambda_2 = -\delta$ for any finite time interval, where λ_1 and λ_2 are the Lyapunov exponents [MQ02].

```
In[59]:= splittingsol = NDSolve[{eqs, ics}, vars, time, StartingStepSize → 1 / 10,
            Method → {NDSolve`Splitting, "DifferenceOrder" → 2,
              "Equations" → {Y2, Y1, Y2}, "Method" → {"LocallyExact"}}]

Out[59]= {{Y₁[T] → InterpolatingFunction[{{0., 10.}}, <>][T],
           Y₂[T] → InterpolatingFunction[{{0., 10.}}, <>][T],
           Z[T] → InterpolatingFunction[{{0., 10.}}, <>][T]}}
```

Here is a plot of the solution.

In[60]:=
```
ParametricPlot[Evaluate[system["DependentVariables"[]] /. First[splittingsol]],
 Evaluate[time], AspectRatio -> 1]
```

Out[60]=



Option Summary

The default coefficient choice in `"Composition"` tries to automatically select between `"SymmetricCompositionCoefficients"` and `"SymmetricCompositionSymmetricMethodCoefficients"` depending on the properties of the methods specified using the `Method` option.

option name	*default value*	
`"Coefficients"`	`Automatic`	specify the coefficients to use in the composition method
`"DifferenceOrder"`	`Automatic`	specify the order of local accuracy of the method
`Method`	`None`	specify the base methods to use in the numerical integration

Options of the method `"Composition"`.

option name	default value	
`“Coefficients“`	`{}`	specify the coefficients to use in the splitting method
`“DifferenceOrder“`	`Automatic`	specify the order of local accuracy of the method
`“Equations“`	`{}`	specify the way in which the equations should be split
`Method`	`None`	specify the base methods to use in the numerical integration

Options of the method `“Splitting“`.

Submethods

"LocallyExact" Method for NDSolve

Introduction

A differential system can sometimes be solved by analytic means. The function `DSolve` implements many of the known algorithmic techniques.

However, differential systems that can be solved in closed form constitute only a small subset. Despite this fact, when a closed-form solution does not exist for the entire vector field, it is often possible to analytically solve a system of differential equations for part of the vector field. An example of this is the method `“Splitting“`, which breaks up a vector field f into subfields $f_1, \ldots, f_n$ such that $f = f_1 + \cdots + f_n$.

The idea underlying the method `“LocallyExact“` is that rather than using a standard numerical integration scheme, when a solution can be found by `DSolve` direct numerical evaluation can be used to locally advance the solution.

Since the method `“LocallyExact“` makes no attempt to adaptively adjust step sizes, it is primarily intended for use as a submethod between integration steps.

Examples

Load a package with some predefined problems.

```
In[1]:= Needs[“DifferentialEquations`NDSolveProblems`“];
```

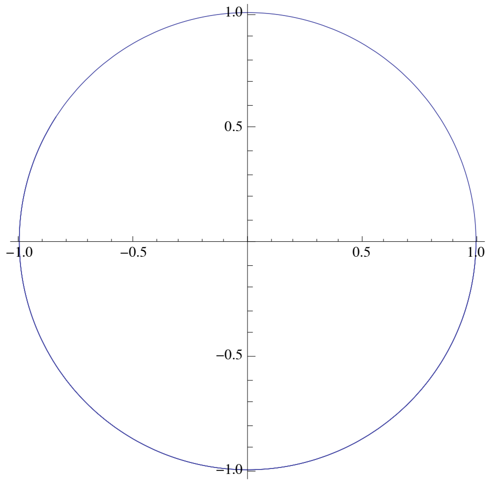
Harmonic Oscillator

Numerically solve the equations of motion for a harmonic oscillator using the method "LocallyExact". The result is two interpolating functions that approximate the solution and the first derivative.

```
In[2]:= system = GetNDSolveProblem["HarmonicOscillator"];
        vars = system["DependentVariables"];
        tdata = system["TimeData"];

        sols =
         vars /. First[NDSolve[system, StartingStepSize → 1 / 10, Method → "LocallyExact"]]
Out[5]= {InterpolatingFunction[{{0., 10.}}, <>][T], InterpolatingFunction[{{0., 10.}}, <>][T]}
```

The solution evolves on the unit circle.

```
In[6]:= ParametricPlot[Evaluate[sols], Evaluate[tdata], AspectRatio → 1]
Out[6]=
```



Global versus Local

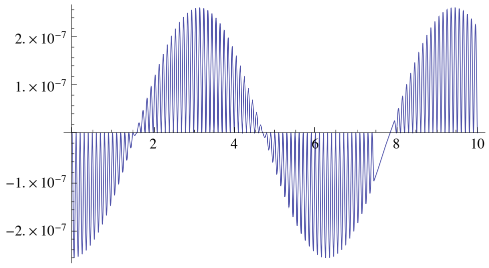
The method "LocallyExact" is not intended as a substitute for a closed-form (global) solution.

Despite the fact that the method "LocallyExact" uses the analytic solution to advance the solution, it only produces solutions at the grid points in the numerical integration (or even inside grid points if called appropriately). Therefore, there can be errors due to sampling at interpolation points that do not lie exactly on the numerical integration grid.

Plot the error in the first solution component of the harmonic oscillator and compare it with the exact flow.

```
In[7]:= Plot[Evaluate[First[sols] - Cos[T]], Evaluate[tdata]]
```

Out[7]=



Simplification

The method "LocallyExact" has an option "SimplificationFunction" that can be used to simplify the results of DSolve.

Here is the linearized component of the differential system that turns up in the splitting of the Lorenz equations using standard values for the parameters.

```
In[8]:= eqs = {Y₁'[T] == σ (Y₂[T] - Y₁[T]), Y₂'[T] == r Y₁[T] - Y₂[T], Y₃'[T] == -b Y₃[T]} /.
            {σ → 10, r → 28, b → 8 / 3};
        ics = {Y₁[0] == -8, Y₂[0] == 8, Y₃[0] == 27};
        vars = {Y₁[T], Y₂[T], Y₃[T]};
```

This subsystem is exactly solvable by `DSolve`.

```
In[11]:= DSolve[eqs, vars, T]
```

Out[11]=

$$\Big\{\Big\{Y_1[T] \to \frac{1}{2402}\left(1201\, e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T} + 9\sqrt{1201}\, e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T} + 1201\, e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T} - 9\sqrt{1201}\, e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T}\right) C[1] - \frac{10\left(e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T} - e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T}\right) C[2]}{\sqrt{1201}},$$

$$Y_2[T] \to -\frac{28\left(e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T} - e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T}\right) C[1]}{\sqrt{1201}} + \frac{1}{2402}\left(1201\, e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T} - 9\sqrt{1201}\, e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T} + 1201\, e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T} + 9\sqrt{1201}\, e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T}\right) C[2],\ Y_3[T] \to e^{-8T/3}\, C[3]\Big\}\Big\}$$

Often the results of `DSolve` can be simplified. This defines a function to simplify an expression and also prints out the input and the result.

```
In[12]:= myfun[x_] :=
           Module[{simpx},
            Print[“Before simplification “, x];
            simpx = FullSimplify[ExpToTrig[x]];
            Print[“After simplification “, simpx];
            simpx
           ];
```

The function can be passed as an option to the method “`LocallyExact`“.

```
In[13]:= NDSolve[{eqs, ics}, vars, {T, 0, 1}, StartingStepSize → 1 / 10,
          Method → {“LocallyExact“, “SimplificationFunction“ → myfun}]
```

Before simplification

$$\left\{\frac{1}{2402}\left(1201\,e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T}+9\sqrt{1201}\,e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T}+1201\,e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T}-9\sqrt{1201}\,e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T}\right)Y_1[T]-\frac{10\left(e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T}-e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T}\right)Y_2[T]}{\sqrt{1201}},\right.$$
$$-\frac{28\left(e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T}-e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T}\right)Y_1[T]}{\sqrt{1201}}+\frac{1}{2402}\left(1201\,e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T}-9\sqrt{1201}\,e^{\frac{1}{2}\left(-11-\sqrt{1201}\right)T}+1201\,e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T}+9\sqrt{1201}\,e^{\frac{1}{2}\left(-11+\sqrt{1201}\right)T}\right)Y_2[T],\ \left.e^{-8T/3}\,Y_3[T]\right\}$$

After simplification

$$\left\{\frac{1}{1201}e^{-11T/2}\left(1201\,\mathrm{Cosh}\left[\frac{\sqrt{1201}\,T}{2}\right]Y_1[T]+\sqrt{1201}\,\mathrm{Sinh}\left[\frac{\sqrt{1201}\,T}{2}\right]\left(-9\,Y_1[T]+20\,Y_2[T]\right)\right),\ e^{-11T/2}\,\mathrm{Cosh}\left[\frac{\sqrt{1201}\,T}{2}\right]Y_2[T]+\right.$$
$$\left.\frac{e^{-11T/2}\,\mathrm{Sinh}\left[\frac{\sqrt{1201}\,T}{2}\right]\left(56\,Y_1[T]+9\,Y_2[T]\right)}{\sqrt{1201}},\ e^{-8T/3}\,Y_3[T]\right\}$$

```
Out[13]= {{Y1[T] → InterpolatingFunction[{{0., 1.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 1.}}, <>][T],
           Y3[T] → InterpolatingFunction[{{0., 1.}}, <>][T]}}
```

The simplification is performed only once during the initialization phase that constructs the data object for the numerical integration method.

Option Summary

option name	*default value*	
"SimplificationFunction"	None	function to use in simplifying the result of DSolve

Option of the method "LocallyExact".

"DoubleStep" Method for NDSolve

Introduction

The method `"DoubleStep"` performs a single application of Richardson's extrapolation for any one-step integration method.

Although it is not always optimal, it is a general scheme for equipping a method with an error estimate (hence adaptivity in the step size) and extrapolating to increase the order of local accuracy.

`"DoubleStep"` is a special case of extrapolation but has been implemented as a separate method for efficiency.

Given a method of order p:

- Take a step of size h to get a solution y_1.
- Take two steps of size $h/2$ to get a solution y_2.
- Find an error estimate of order p as:

$$e = \frac{y_2 - y_1}{2^p - 1}. \tag{1}$$

- The correction term e can be used for error estimation enabling an adaptive step-size scheme for any base method.
- Either use y_2 for the new solution, or form an improved approximation using local extrapolation as:

$$\hat{y}_2 = y_2 + e. \tag{2}$$

- If the base numerical integration method is symmetric, then the improved approximation has order $p+2$; otherwise it has order $p+1$.

Examples

Load some package with example problems and utility functions.

```
In[5]:= Needs["DifferentialEquations`NDSolveProblems`"];
        Needs["DifferentialEquations`NDSolveUtilities`"];
```

Select a nonstiff problem from the package.

```
In[7]:= nonstiffsystem = GetNDSolveProblem["BrusselatorODE"];
```

Select a stiff problem from the package.

```
In[8]:= stiffsystem = GetNDSolveProblem["Robertson"];
```

Extending Built-in Methods

The method "ExplicitEuler" carries out one integration step using Euler's method. It has no local error control and hence uses fixed step sizes.

This integrates a differential system using one application of Richardson's extrapolation (see (2)) with the base method "ExplicitEuler".

The local error estimate (1) is used to dynamically adjust the step size throughout the integration.

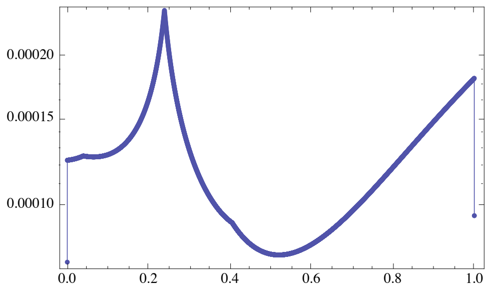
```
In[9]:= eesol = NDSolve[nonstiffsystem, {T, 0, 1},
          Method → {"DoubleStep", Method → "ExplicitEuler"}]
```

```
Out[9]= {{Y1[T] → InterpolatingFunction[{{0., 1.}}, <>][T],
          Y2[T] → InterpolatingFunction[{{0., 1.}}, <>][T]}}
```

This illustrates how the step size varies during the numerical integration.

```
In[10]:= StepDataPlot[eesol]
```

Out[10]=



The stiffness detection device (described within "StiffnessTest Method Option for NDSolve") ascertains that the "ExplicitEuler" method is restricted by stability rather than local accuracy.

```
In[11]:= NDSolve[stiffsystem, Method → {"DoubleStep", Method → "ExplicitEuler"}]

      NDSolve::ndstf :
        At T == 0.007253212186800964`, system appears to be stiff. Methods Automatic, BDF or
              StiffnessSwitching may be more appropriate. ≫
```

```
Out[11]= {{Y1[T] → InterpolatingFunction[{{0., 0.00725321}}, <>][T],
          Y2[T] → InterpolatingFunction[{{0., 0.00725321}}, <>][T],
          Y3[T] → InterpolatingFunction[{{0., 0.00725321}}, <>][T]}}
```

An alternative base method is more appropriate for this problem.

```
In[12]:= liesol =
          NDSolve[stiffsystem, Method → {"DoubleStep", Method → "LinearlyImplicitEuler"}]

Out[12]= {{Y1[T] → InterpolatingFunction[{{0., 0.3}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 0.3}}, <>][T],
           Y3[T] → InterpolatingFunction[{{0., 0.3}}, <>][T]}}
```

User-Defined Methods and Method Properties

Integration methods can be added to the `NDSolve` framework.

In order for these to work like built-in methods it can be necessary to specify various method properties. These properties can then be used by other methods to build up compound integrators.

Here is how to define a top-level plug-in for the classical Runge-Kutta method (see "NDSolve Method Plug-in Framework: Classical Runge-Kutta" and "ExplicitRungeKutta Method for NDSolve" for more details).

```
In[13]:= ClassicalRungeKutta[___]["Step"[f_, t_, h_, y_, yp_]] :=
           Block[{deltay, k1, k2, k3, k4},
            k1 = yp;
            k2 = f[t + 1 / 2 h, y + 1 / 2 h k1];
            k3 = f[t + 1 / 2 h, y + 1 / 2 h k2];
            k4 = f[t + h, y + h k3];
            deltay = h (1 / 6 k1 + 1 / 3 k2 + 1 / 3 k3 + 1 / 6 k4);
            {h, deltay}
           ];
```

Method properties used by `"DoubleStep"` are now described.

Order and Symmetry

This attempts to integrate a system using one application of Richardson's extrapolation based on the classical Runge-Kutta method.

```
In[14]:= NDSolve[nonstiffsystem, Method → {"DoubleStep", Method → ClassicalRungeKutta}];

    NDSolve::mtdp :
      ClassicalRungeKutta does not have a correctly defined property DifferenceOrder in DoubleStep. ≫

    NDSolve::initf : The initialization of the method NDSolve`DoubleStep failed. ≫
```

Without knowing the order of the base method, `"DoubleStep"` is unable to carry out Richardson's extrapolation.

This defines a method property to communicate to the framework that the classical Runge-Kutta method has order four.

```
In[15]:= ClassicalRungeKutta[___]["DifferenceOrder"] := 4;
```

The method “DoubleStep“ is now able to ascertain that ClassicalRungeKutta is of order four and can use this information when refining the solution and estimating the local error.

```
In[16]:= NDSolve[nonstiffsystem, Method → {“DoubleStep“, Method → ClassicalRungeKutta}]

Out[16]= {{Y1[T] → InterpolatingFunction[{{0., 20.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 20.}}, <>][T]}}
```

The order of the result of Richardson's extrapolation depends on whether the extrapolated method has a local error expansion in powers of h or h^2 (the latter occurs if the base method is symmetric).

If no method property for symmetry is defined, the “DoubleStep“ method assumes by default that the base integrator is not symmetric.

This explicitly specifies that the classical Runge-Kutta method is not symmetric using the “SymmetricMethodQ“ property.

```
In[17]:= ClassicalRungeKutta[___][“SymmetricMethodQ“] := False;
```

Stiffness Detection

Details of the scheme used for stiffness detection can be found within "StiffnessTest Method Option for NDSolve".

Stiffness detection relies on knowledge of the linear stability boundary of the method, which has not been defined.

Computing the exact linear stability boundary of a method under extrapolation can be quite complicated. Therefore a default value is selected which works for all methods. This corresponds to considering the p-th order power series approximation to the exponential at 0 and ignoring higher order terms.

- If “LocalExtrapolation“ is True then a generic value is selected corresponding to a method of order p + 2 (symmetric) or p + 1.
- If “LocalExtrapolation“ is False then the property “LinearStabilityBoundary“ of the base method is checked. If no value has been specified then a default for a method of order p is selected.

This computes the linear stability boundary for a generic method of order 4.

```
In[18]:= Reduce[Abs[Sum[z^i/i!, {i, 0, 4}]] == 1 && z < 0, z]

Out[18]= z == Root[24 + 12 #1 + 4 #1^2 + #1^3 &, 1]
```

A default value for the "LinearStabilityBoundary" property is used.

```
In[19]:= NDSolve[stiffsystem,
           Method → {"DoubleStep", Method → ClassicalRungeKutta, "StiffnessTest" → True}];
```

```
NDSolve::ndstf : At T == 0.00879697198122793`, system appears to be stiff. Methods
        Automatic, BDF or StiffnessSwitching may be more appropriate. ≫
```

This shows how to specify the linear stability boundary of the method for the framework. This value will only be used if "DoubleStep" is invoked with "LocalExtrapolation" → True .

```
In[20]:= ClassicalRungeKutta[___]["LinearStabilityBoundary"] :=
           Root[24 + 12 #1 + 4 #1^2 + #1^3 &, 1];
```

"DoubleStep" assumes by default that a method is not appropriate for stiff problems (and hence uses stiffness detection) when no "StiffMethodQ" property is specified. This shows how to define the property.

```
In[21]:= ClassicalRungeKutta[___]["StiffMethodQ"] := False;
```

Higher Order

The following example extrapolates the classical Runge-Kutta method of order four using two applications of (2).

The inner specification of "DoubleStep" constructs a method of order five.

A second application of "DoubleStep" is used to obtain a method of order six, which uses adaptive step sizes.

Nested applications of "DoubleStep" are used to raise the order and provide an adaptive step-size estimate.

```
In[22]:= NDSolve[nonstiffsystem,
          Method → {"DoubleStep", Method → {"DoubleStep", Method -> ClassicalRungeKutta}}]

Out[22]= {{Y1[T] → InterpolatingFunction[{{0., 20.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 20.}}, <>][T]}}
```

In general the method "Extrapolation" is more appropriate for constructing high-order integration schemes from low-order methods.

Option Summary

option name	*default value*	
`"LocalExtrapolation"`	`True`	specify whether to advance the solution using local extrapolation according to (2)
`Method`	`None`	specify the method to use as the base integration scheme
`"StepSizeRatioBounds"`	$\left\{\frac{1}{8}, 4\right\}$	specify the bounds on a relative change in the new step size h_{n+1} from the current step size h_n as low $\leq h_{n+1} / h_n \leq$ high
`"StepSizeSafetyFactors"`	`Automatic`	specify the safety factors to incorporate into the error estimate (1) used for adaptive step sizes
`"StiffnessTest"`	`Automatic`	specify whether to use the stiffness detection capability

Options of the method `"DoubleStep"`.

The default setting of `Automatic` for the option `"StiffnessTest"` indicates that the stiffness test is activated if a nonstiff base method is used.

The default setting of `Automatic` for the option `"StepSizeSafetyFactors"` uses the values `{9 / 10, 4 / 5}` for a stiff base method and `{9 / 10, 13 / 20}` for a nonstiff base method.

"EventLocator" Method for NDSolve

Introduction

It is often useful to be able to detect and precisely locate a change in a differential system. For example, with the detection of a singularity or state change, the appropriate action can be taken, such as restarting the integration.

An event for a differential system:

$$Y'(t) = f(t, Y(t))$$

is a point along the solution at which a real-valued event function is zero:

$$g(t, Y(t)) = 0$$

It is also possible to consider Boolean-valued event functions, in which case the event occurs when the function changes from `True` to `False` or vice versa.

The “EventLocator“ method that is built into NDSolve works effectively as a controller method; it handles checking for events and taking the appropriate action, but the integration of the differential system is otherwise left completely to an underlying method.

In this section, examples are given to demonstrate the basic use of the “EventLocator“ method and options. Subsequent sections show more involved applications of event location, such as period detection, Poincaré sections, and discontinuity handling.

These initialization commands load some useful packages that have some differential equations to solve and define some utility functions.

```
In[1]:= Needs[“DifferentialEquations`NDSolveProblems`“];
        Needs[“DifferentialEquations`NDSolveUtilities`“];
        Needs[“DifferentialEquations`InterpolatingFunctionAnatomy`“];
        Needs[“GUIKit`“];
```

A simple example is locating an event, such as when a pendulum started at a non-equilibrium position will swing through its lowest point and stopping the integration at that point.

This integrates the pendulum equation up to the first point at which the solution $y[t]$ crosses the axis.

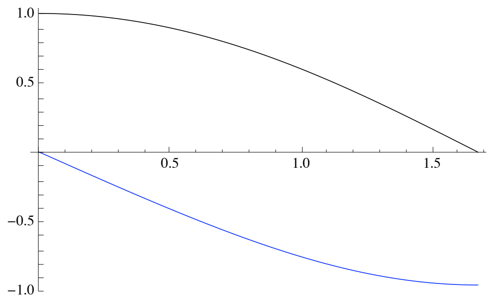
```
In[5]:= sol = NDSolve[{y‘‘[t] + Sin[y[t]] == 0, y‘[0] == 0, y[0] == 1},
          y, {t, 0, 10}, Method → {“EventLocator“, “Event“ → y[t]}]
Out[5]= {{y → InterpolatingFunction[{{0., 1.67499}}, <>]}}
```

From the solution you can see that the pendulum reaches its lowest point $y[t] = 0$ at about $t = 1.675$. Using the InterpolatingFunctionAnatomy package, it is possible to extract the value from the InterpolatingFunction object.

This extracts the point at which the event occurs and makes a plot of the solution (black) and its derivative (blue) up to that point.

```
In[6]:= end = InterpolatingFunctionDomain[First[y /. sol]][[1, -1]];
        Plot[Evaluate[{y[t], y‘[t]} /. First[sol]],
         {t, 0, end}, PlotStyle → {{Black}, {Blue}}]
```

Out[7]=



When you use the event locator method, the events to be located and the action to take upon finding an event are specified through method options of the “EventLocator“ method.

The default action on detecting an event is to stop the integration as demonstrated earlier. The event action can be any expression. It is evaluated with numerical values substituted for the problem variables whenever an event is detected.

This prints the time and values each time the event *y*' [*t*] = *y*[*t*] is detected for a damped pendulum.

```
In[8]:= NDSolve[{y''[t] + .1 y'[t] + Sin[y[t]] == 0, y'[0] == 0, y[0] == 1},
          y, {t, 0, 10}, Method → {"EventLocator", "Event" → y'[t] - y[t],
            "EventAction" :→ Print["y'[", t, "] = y[", t, "] = ", y[t]]}]

        y'[2.49854] = y[2.49854] = -0.589753

        y'[5.7876] = y[5.7876] = 0.501228

        y'[9.03428] = y[9.03428] = -0.426645

Out[8]= {{y → InterpolatingFunction[{{0., 10.}}, <>]}}
```

Note that in the example, the "EventAction" option was given using RuleDelayed (:→) to prevent it from evaluating except when the event is located.

You can see from the printed output that when the action does not stop the integration, multiple instances of an event can be detected. Events are detected when the sign of the event expression changes. You can restrict the event to be only for a sign change in a particular direction using the "Direction" option.

This collects the points at which the velocity changes from negative to positive for a damped driven pendulum. Reap and Sow are programming constructs that are useful for collecting data when you do not, at first, know how much data there will be. Reap[*expr*] gives the value of *expr* together with all expressions to which Sow has been applied during its evaluation. Here Reap encloses the use of NDSolve and Sow is a part of the event action, which allows you to collect data for each instance of an event.

```
In[9]:= Reap[NDSolve[{y''[t] + .1 y'[t] + Sin[y[t]] == .1 Cos[t], y'[0] == 0, y[0] == 1},
           y, {t, 0, 50}, Method → {"EventLocator", "Event" → y'[t],
             "Direction" → 1, "EventAction" :→ Sow[{t, y[t], y'[t]}]}]]

Out[9]= {{{y → InterpolatingFunction[{{0., 50.}}, <>]}},
         {{{3.55407, -0.879336, 1.87524×10^-15}, {10.4762, -0.832217, -5.04805×10^-16},
           {17.1857, -0.874939, -4.52416×10^-15}, {23.7723, -0.915352, 1.62717×10^-15},
           {30.2805, -0.927186, -1.17094×10^-16}, {36.7217, -0.910817, -2.63678×10^-16},
           {43.1012, -0.877708, 1.33227×10^-15}, {49.4282, -0.841083, -8.66494×10^-16}}}}
```

You may notice from the output of the previous example that the events are detected when the derivative is only approximately zero. When the method detects the presence of an event in a step of the underlying integrator (by a sign change of the event expression), then it uses a numerical method to approximately find the position of the root. Since the location process is numerical, you should expect only approximate results. Location method options `AccuracyGoal`, `PrecisionGoal`, and `MaxIterations` can be given to those location methods that use `FindRoot` to control tolerances for finding the root.

For Boolean valued event functions, an event occurs when the function switches from `True` to `False` or vice versa. The `"Direction"` option can be used to restrict the event only from changes from `True` to `False` (`"Direction" -> -1`) or only from changes from `False` to `True` (`"Direction" -> 1`).

This opens up a small window with a button, which when clicked changes the value of the variable stop to `True` from its initialized value of `False`.

```
In[10]:= NDSolve`stop = False;
         GUIRun[Widget["Panel", {Widget["Button", {
                "label" → "Stop",
                BindEvent["action",
                 Script[NDSolve`stop = True]]}]}]];
```

This integrates the pendulum equation up until the button is clicked (or the system runs out of memory).

```
In[12]:= NDSolve[{y''[t] + Sin[y[t]] == 0, y[0] == 1, y'[0] == 0}, y, {t, 0, ∞},
          Method → {"EventLocator", "Event" :→ NDSolve`stop}, MaxSteps → ∞]
Out[12]= {{y → InterpolatingFunction[{{0., 620015.}}, <>]}}
```

Take note that in this example, the `"Event"` option was specified with `RuleDelayed (:>)` to prevent the immediate value of `stop` from being evaluated and set up as the function.

You can specify more than one event. If the event function evaluates numerically to a list, then each component of the list is considered to be a separate event. You can specify different actions, directions, etc. for each of these events by specifying the values of these options as lists of the appropriate length.

This integrates the pendulum equation up until the point at which the button is clicked. The number of complete swings of the pendulum is kept track of during the integration.

```
In[13]:= NDSolve`stop = False;
         swings = 0; {
          NDSolve[{y''[t] + Sin[y[t]] == 0, y[0] == 0, y'[0] == 1}, y,
            {t, 0, 1000000}, Method → {"EventLocator", "Event" :→ {y[t], NDSolve`stop},
              "EventAction" :→ {swings++, Throw[Null, "StopIntegration"]},
              "Direction" → {1, All}}, MaxSteps → Infinity], swings}
Out[13]= {{{y → InterpolatingFunction[{{0., 24903.7}}, <>]}}, 3693}
```

As you can see from the previous example, it is possible to mix real- and Boolean-valued event functions. The expected number of components and type of each component are based on the values at the initial condition and needs to be consistent throughout the integration.

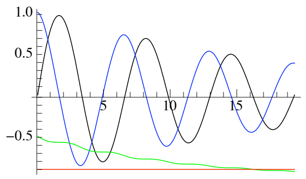
The "EventCondition" option of "EventLocator" allows you to specify additional Boolean conditions that need to be satisfied for an event to be tested. It is advantageous to use this instead of a Boolean event when possible because the root finding process can be done more efficiently.

This stops the integration of a damped pendulum at the first time that $y(t) = 0$ once the decay has reduced the energy integral to -0.9.

```
In[14]:= sol = NDSolve[{y''[t] + .1 y'[t] + Sin[y[t]] == 0, y'[0] == 1, y[0] == 0},
           y, {t, 0, 100}, Method → {"EventLocator", "Event" → y[t],
             "EventCondition" → (y'[t]^2 / 2 - Cos[y[t]] < -0.9),
             "EventAction" :→ Throw[end = t, "StopIntegration"]}]
Out[14]= {{y → InterpolatingFunction[{{0., 19.4446}}, <>]}}
```

This makes a plot of the solution (black), the derivative (blue), and the energy integral (green). The energy theshold is shown in red.

```
In[15]:= Plot[Evaluate[{y[t], y'[t], y'[t]^2 / 2 - Cos[y[t]], -.9} /. First[sol]],
          {t, 0, end}, PlotStyle → {{Black}, {Blue}, {Green}, {Red}}]
Out[15]=
```



The Method option of "EventLocator" allows the specification of the numerical method to use in the integration.

Event Location Methods

The "EventLocator" method works by taking a step of the underlying method and checking to see if the sign (or parity) of any of the event functions is different at the step endpoints. Event functions are expected to be real- or Boolean-valued, so if there is a change, there must be an event in the step interval. For each event function which has an event occurrence in a step, a refinement procedure is carried out to locate the position of the event within the interval.

There are several different methods which can be used to refine the position. These include simply taking the solution at the beginning or the end of the integration interval, a linear interpolation of the event value, and using bracketed root-finding methods. The appropriate method to use depends on a trade off between execution speed and location accuracy.

If the event action is to stop the integration then the particular value at which the integration is stopped depends on the value obtained from the “EventLocationMethod“ option of “EventLocator“.

Location of a single event is usually fast enough so that the method used will not significantly influence the overall computation time. However, when an event is detected multiple times, the location refinement method can have a substantial effect.

"StepBegin" and "StepEnd" Methods

The crudest methods are appropriate for when the exact position of the event location does not really matter or does not reflect anything with precision in the underlying calculation. The stop button example from the previous section is such a case: time steps are computed so quickly that there is no way that you can time the click of a button to be within a particular time step, much less at a particular point within a time step. Thus, based on the inherent accuracy of the event, there is no point in refining at all. You can specify this by using the “StepBegin“ or “StepEnd“ location methods. In any example where the definition of the event is heuristic or somewhat imprecise, this can be an appropriate choice.

"LinearInterpolation" Method

When event results are needed for the purpose of points to plot in a graph, you only need to locate the event to the resolution of the graph. While just using the step end is usually too crude for this, a single linear interpolation based on the event function values suffices.

Denote the event function values at successive mesh points of the numerical integration:

$$w_n = g(t_n,\, y_n),\ w_{n+1} = g(t_{n+1},\, y_{n+1})$$

Linear interpolation gives:

$$w_e = \left| \frac{w_n}{w_{n+1} - w_n} \right|$$

A linear approximation of the event time is then:

$$t_e = t_n + w_e\, h_n$$

Linear interpolation could also be used to approximate the solution at the event time. However, since derivative values $f_n = f(t_n, y_n)$ and $f_{n+1} = f(t_{n+1}, y_{n+1})$ are available at the mesh points, a better approximation of the solution at the event can be computed cheaply using cubic Hermite interpolation as:

$$y_e = k_n\, y_n + k_{n+1}\, y_{n+1} + l_n\, f_n + l_{n+1}\, f_{n+1}$$

for suitably defined interpolation weights:

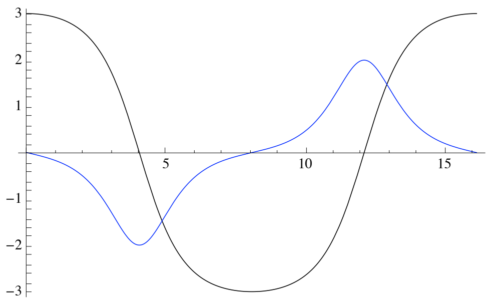
$$\begin{aligned} k_n &= (w_e - 1)^2\,(2\, w_e + 1) \\ k_{n+1} &= (3 - 2\, w_e)\, w_e^2 \\ l_n &= h_n\,(w_e - 1)^2\, w_e \\ l_{n+1} &= h_n\,(w_e - 1)\, w_e^2 \end{aligned}$$

You can specify refinement based on a single linear interpolation with the setting `"LinearInterpolation"`.

This computes the solution for a single period of the pendulum equation and plots the solution for that period.

```
In[16]:= sol = First[NDSolve[{y''[t] + Sin[y[t]] == 0, y[0] == 3, y'[0] == 0},
             y, {t, 0, ∞}, Method → {"EventLocator",
              "Event" → y'[t],
             "EventAction" :→ Throw[end = t, "StopIntegration"], "Direction" → -1,
             "EventLocationMethod" -> "LinearInterpolation",
             Method -> "ExplicitRungeKutta"}]];
         Plot[Evaluate[{y[t], y'[t]} /. sol], {t, 0, end}, PlotStyle → {{Black}, {Blue}}]
```

Out[17]=

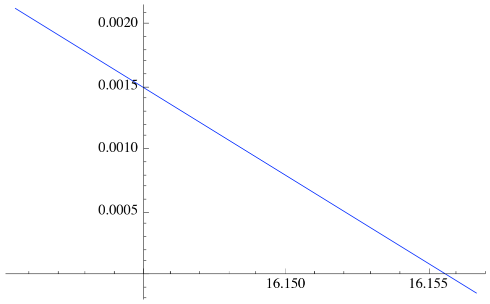


At the resolution of the plot over the entire period, you cannot see that the endpoint may not be exactly where the derivative hits the axis. However, if you zoom in enough, you can see the error.

This shows a plot just near the endpoint.

```
In[18]:= Plot[Evaluate[y'[t] /. sol], {t, end * (1 - .001), end}, PlotStyle → Blue]
```

Out[18]=



The linear interpolation method is sufficient for most viewing purposes, such as the Poincaré section examples shown in the following section. Note that for Boolean-valued event functions, linear interpolation is effectively only one bisection step, so the linear interpolation method may be inadequate for graphics.

Brent's Method

The default location method is the event location method `"Brent"`, finding the location of the event using `FindRoot` with Brent's method. Brent's method starts with a bracketed root and combines steps based on interpolation and bisection, guaranteeing a convergence rate at least as good as bisection. You can control the accuracy and precision to which `FindRoot` tries to get the root of the event function using method options for the `"Brent"` event location method. The default is to find the root to the same accuracy and precision as `NDSolve` is using for local error control.

For methods that support continuous or dense output, the argument for the event function can be found quite efficiently simply by using the continuous output formula. However, for methods that do not support continuous output, the solution needs to be computed by taking a step of the underlying method, which can be relatively expensive. An alternate way of getting a solution approximation that is not accurate to the method order, but is consistent with using `FindRoot` on the `InterpolatingFunction` object returned from `NDSolve` is to use cubic Hermite interpolation, obtaining approximate solution values in the middle of the step by interpolation based on the solution values and solution derivative values at the step ends.

Comparison

This example integrates the pendulum equation for a number of different event location methods and compares the time when the event is found.

This defines the event location methods to use.

```
In[19]:= eventmethods = {"StepBegin", "StepEnd", "LinearInterpolation", Automatic};
```

This integrates the system and prints out the method used and the value of the independent variable when the integration is terminated.

```
In[20]:= Map[
           NDSolve[{y''[t] + Sin[y[t]] == 0, y[0] == 3, y'[0] == 0},
             y, {t, 0, ∞}, Method → {"EventLocator",
                "Event" → y'[t],
               "EventAction" :→ Throw[Print[#, ": t = ", t, " y'[t] = ", y'[t]],
                 "StopIntegration"], "Direction" → -1, Method -> "ExplicitRungeKutta",
               "EventLocationMethod" → #}] &,
           eventmethods
          ];

StepBegin: t = 15.8022 y'[t] = 0.0508999

StepEnd: t = 16.226 y'[t] = -0.00994799

LinearInterpolation: t = 16.1567 y'[t] = -0.000162503

Automatic: t = 16.1555 y'[t] = -2.35922×10^-16
```

Examples

Falling Body

This system models a body falling under the force of gravity encountering air resistance (see [M04]).

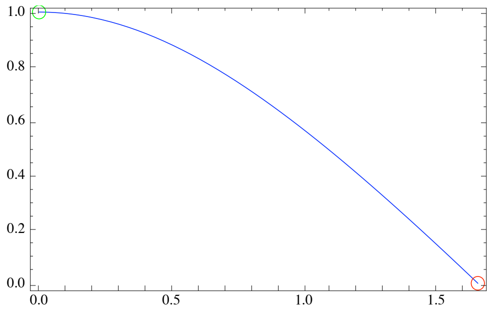
The event action stores the time when the falling body hits the ground and stops the integration.

```
In[21]:= sol = y[t] /. First[NDSolve[{y''[t] == -1 + y'[t]^2, y[0] == 1, y'[0] == 0},
              y, {t, 0, Infinity}, Method → {"EventLocator", "Event" :→ y[t],
                "EventAction" :→ Throw[tend = t, "StopIntegration"]}]]

Out[21]= InterpolatingFunction[{{0., 1.65745}}, <>][t]
```

This plots the solution and highlights the initial and final points (green and red) by encircling them.

```
In[22]:= plt = Plot[sol, {t, 0, tend}, Frame → True,
            Axes → False, PlotStyle → Blue, DisplayFunction → Identity];

         grp = Graphics[
            {{Green, Circle[{0, 1}, 0.025]}, {Red, Circle[{tend, sol /. t → tend}, 0.025]}}];

         Show[plt, grp, DisplayFunction → $DisplayFunction]
```

Out[24]=



Period of the Van der Pol Oscillator

The Van der Pol oscillator is an example of an extremely stiff system of ODEs. The event locator method can call any method for actually doing the integration of the ODE system. The default method, `Automatic`, automatically switches to a method appropriate for stiff systems when necessary, so that stiffness does not present a problem.

This integrates the Van der Pol system for a particular value of the parameter $\mu = 1000$ up to the point where the variable y_2 reaches its initial value and direction.

```
In[25]:= vsol = NDSolve[(                  y1'[t] == y2[t]                 y1[0] == 2 ),
                         ( y2'[t] == 1000 (1 - y1[t]^2) y2[t] - y1[t]  y2[0] == 0 )
           {y1, y2}, {t, 3000},
           Method → {"EventLocator", "Event" → y2[t], "Direction" → -1}]

Out[25]= {{y1 → InterpolatingFunction[{{0., 1614.29}}, <>],
           y2 → InterpolatingFunction[{{0., 1614.29}}, <>]}}
```

Note that the event at the initial condition is not considered.

By selecting the endpoint of the `NDSolve` solution, it is possible to write a function that returns the period as a function of μ.

This defines a function that returns the period as a function of μ.

```
In[26]:= vper[μ_] := Module[{vsol},
            vsol = First[y₂ /. NDSolve[(              y₁'[t] == y₂[t]              y₁[0] == 2 ),
                                       y₂'[t] == μ (1 - y₁[t]^2) y₂[t] - y₁[t]   y₂[0] == 0
                 {y₁, y₂}, {t, Max[100, 3 μ]},
                 Method → {"EventLocator", "Event" → y₂[t], "Direction" → -1}]];
            InterpolatingFunctionDomain[vsol][[1, -1]]];
```

This uses the function to compute the period at $\mu = 1000$.

```
In[27]:= vper[1000]
Out[27]= 1614.29
```

Of course, it is easy to generalize this method to any system with periodic solutions.

Poincaré Sections

Using Poincaré sections is a useful technique for visualizing the solutions of high-dimensional differential systems.

For an interactive graphical interface see the package EquationTrekker.

The Hénon-Heiles System

Define the Hénon-Heiles system that models stellar motion in a galaxy.

This gets the Hénon-Heiles system from the NDSolveProblems package.

```
In[28]:= system = GetNDSolveProblem["HenonHeiles"];
         vars = system["DependentVariables"];
         eqns = {system["System"], system["InitialConditions"]}
```

$$\text{Out[29]=}\ \left\{\left\{(Y_1)'[T] == Y_3[T],\ (Y_2)'[T] == Y_4[T],\ (Y_3)'[T] == -Y_1[T]\ (1 + 2\,Y_2[T]),\ (Y_4)'[T] == -Y_1[T]^2 + (-1 + Y_2[T])\ Y_2[T]\right\},\ \left\{Y_1[0] == \frac{3}{25},\ Y_2[0] == \frac{3}{25},\ Y_3[0] == \frac{3}{25},\ Y_4[0] == \frac{3}{25}\right\}\right\}$$

The Poincaré section of interest in this case is the collection of points in the $Y_2 - Y_4$ plane when the orbit passes through $Y_1 = 0$.

Since the actual result of the numerical integration is not required, it is possible to avoid storing all the data in InterpolatingFunction by specifying the output variables list (in the second argument to NDSolve) as empty, or {}. This means that NDSolve will produce no InterpolatingFunction as output, avoiding storing a lot of unnecessary data. NDSolve does give a message NDSolve::noout warning there will be no output functions, but it can safely be turned off in this case since the data of interest is collected from the event actions.

The linear interpolation event location method is used because the purpose of the computation here is to view the results in a graph with relatively low resolution. If you were doing an example where you needed to zoom in on the graph to great detail or to find a feature, such as a fixed point of the Poincaré map, it would be more appropriate to use the default location method.

This turns off the message warning about no output.

```
In[30]:= Off[NDSolve::noout];
```

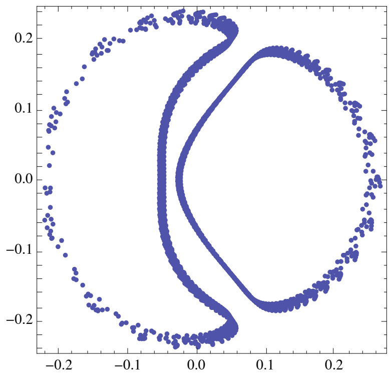
This integrates the Hénon-Heiles system using a fourth-order explicit Runge-Kutta method with fixed step size of 0.25. The event action is to use Sow on the values of Y_2 and Y_4.

```
In[31]:= data =
           Reap[
            NDSolve[eqns, {}, {T, 10 000},
              Method → {"EventLocator", "Event" → Y1[T], "EventAction" :→
                  Sow[{Y2[T], Y4[T]}], "EventLocationMethod" -> "LinearInterpolation",
                "Method" → {"FixedStep", "Method" → {"ExplicitRungeKutta",
                      "DifferenceOrder" → 4}}},
              StartingStepSize → 0.25, MaxSteps → ∞];
           ];
```

This plots the Poincaré section. The collected data is found in the last part of the result of Reap and the list of points is the first part of that.

```
In[32]:= psdata = data[[-1, 1]];
         ListPlot[psdata, Axes → False, Frame → True, AspectRatio → 1]
```

Out[33]=



Since the Hénon-Heiles system is Hamiltonian, a symplectic method gives much better qualitative results for this example.

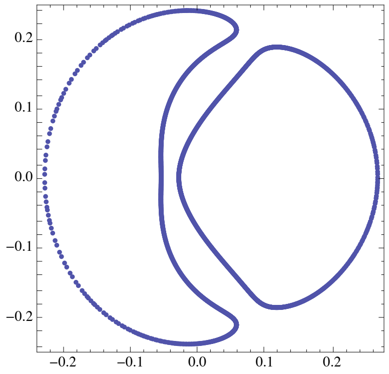
This integrates the Hénon-Heiles system using a fourth-order symplectic partitioned Runge-Kutta method with fixed step size of 0.25. The event action is to use `Sow` on the values of Y_2 and Y_4.

```
In[34]:= sdata =
           Reap[
            NDSolve[eqns, {}, {T, 10 000},
              Method → {"EventLocator", "Event" → Y1[T], "EventAction" :→
                  Sow[{Y2[T], Y4[T]}], "EventLocationMethod" -> "LinearInterpolation",
                "Method" → {"SymplecticPartitionedRungeKutta", "DifferenceOrder" → 4,
                  "PositionVariables" → {Y1[T], Y2[T]}}},
              StartingStepSize → 0.25, MaxSteps → ∞];
           ];
```

This plots the Poincaré section. The collected data is found in the last part of the result of `Reap` and the list of points is the first part of that.

```
In[35]:= psdata = sdata[[-1, 1]];
         ListPlot[psdata, Axes → False, Frame → True, AspectRatio → 1]
```

Out[36]=



The ABC Flow

This loads an example problem of the Arnold-Beltrami-Childress (ABC) flow that is used to model chaos in laminar flows of the three-dimensional Euler equations.

```
In[37]:= system = GetNDSolveProblem["ArnoldBeltramiChildress"];
         eqs = system["System"];
         vars = system["DependentVariables"];
         icvars = vars /. T → 0;
```

This defines a splitting `Y1`, `Y2` of the system by setting some of the right-hand side components to zero.

```
In[41]:= Y1 = eqs; Y1[[2, 2]] = 0; Y1
```

Out[41]= $\left\{(Y_1)'[T] == \frac{3}{4}\text{Cos}[Y_2[T]] + \text{Sin}[Y_3[T]],\ (Y_2)'[T] == 0,\ (Y_3)'[T] == \text{Cos}[Y_1[T]] + \frac{3}{4}\text{Sin}[Y_2[T]]\right\}$

```
In[42]:= Y2 = eqs; Y2[[{1, 3}, 2]] = 0; Y2
```

Out[42]= $\{(Y_1)'[T] == 0,\ (Y_2)'[T] == \text{Cos}[Y_3[T]] + \text{Sin}[Y_1[T]],\ (Y_3)'[T] == 0\}$

This defines the implicit midpoint method.

```
In[43]:= ImplicitMidpoint =
           {“ImplicitRungeKutta“, “Coefficients“ → “ImplicitRungeKuttaGaussCoefficients“,
            “DifferenceOrder“ → 2, “ImplicitSolver“ → {“FixedPoint“,
              AccuracyGoal → 10, PrecisionGoal → 10, “IterationSafetyFactor“ → 1}};
```

This constructs a second-order splitting method that retains volume and reversing symmetries.

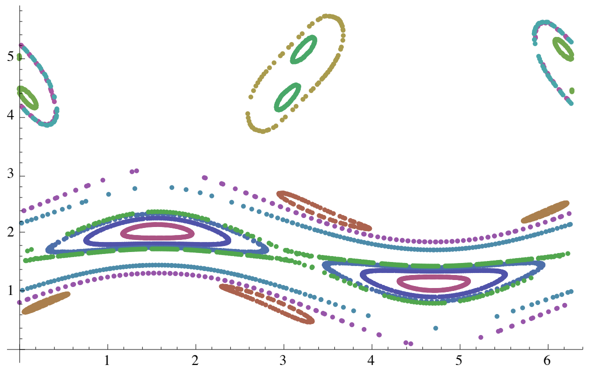
```
In[44]:= ABCSplitting = {“Splitting“,
            “DifferenceOrder“ → 2,
            “Equations“ → {Y2, Y1, Y2},
            “Method“ → {“LocallyExact“, ImplicitMidpoint, “LocallyExact“}};
```

This defines a function that gives the Poincaré section for a particular initial condition.

```
In[45]:= psect[ics_] :=
           Module[{reapdata},
            reapdata =
             Reap[
              NDSolve[{eqs, Thread[icvars == ics]}, {}, {T, 1000},
                Method → {“EventLocator“,
                  “Event“ → Y2[T], “EventAction“ :→ Sow[{Y1[T], Y3[T]}],
                  “EventLocationMethod“ -> “LinearInterpolation“, Method → ABCSplitting},
                StartingStepSize → 1 / 4, MaxSteps → ∞]
             ];
            reapdata[[-1, 1]]
           ];
```

This finds the Poincaré sections for several different initial conditions and flattens them together into a single list of points.

```
In[46]:= data =
           Mod[Map[psect, {{4.267682454609692, 0, 0.9952906114885919},
               {1.6790790859443243, 0, 2.1257099470901704},
               {2.9189523719753327, 0, 4.939152797323216},
               {3.1528896559036776, 0, 4.926744120488727},
               {0.9829282640373566, 0, 1.7074633238173198},
               {0.4090394012299985, 0, 4.170087631574883},
               {6.090600411133905, 0, 2.3736566160602277},
               {6.261716134007686, 0, 1.4987884558838156},
               {1.005126683795467, 0, 1.3745418575363608},
               {1.5880780704325377, 0, 1.3039536044289253},
               {3.622408133554125, 0, 2.289597511313432},
               {0.030948690635763183, 0, 4.306922133429981},
               {5.906038850342371, 0, 5.000045498132029}}],
             2 π];
         ListPlot[data, ImageSize → Medium]
```

Out[47]=


Bouncing Ball

This example is a generalization of an example in [SGT03]. It models a ball bouncing down a ramp with a given profile. The example is good for demonstrating how you can use multiple invocations of `NDSolve` with event location to model some behavior.

This defines a function that computes the solution from one bounce to the next. The solution is computed until the next time the path intersects the ramp.

```
In[48]:= OneBounce[k_, ramp_][{t0_, x0_, xp0_, y0_, yp0_}] :=
          Module[{sol, t1, x1, xp1, y1, yp1, gramp, gp},
           sol = First[NDSolve[
               {x''[t] == 0, x'[t0] == xp0, x[t0] == x0,
                y''[t] == -9.8 , y'[t0] == yp0, y[t0] == y0},
               {x, y},
               {t, t0, ∞}, Method → {"EventLocator", "Event" :→ y[t] - ramp[x[t]]},
               MaxStepSize → 0.01]];
           t1 = InterpolatingFunctionDomain[x /. sol][[1, -1]];
           {x1, xp1, y1, yp1} =
             Reflection[k, ramp][{x[t1], x'[t1], y[t1], y'[t1]} /. sol];
           Sow[{x[t] /. sol, t0 ≤ t ≤ t1}, "X"];
           Sow[{ y[t] /. sol, t0 ≤ t ≤ t1}, "Y"];
           Sow[{x1, y1}, "Bounces"];
           {t1, x1, xp1, y1, yp1}]
```

This defines the function for the bounce when the ball hits the ramp. The formula is based on reflection about the normal to the ramp assuming only the fraction k of energy is left after a bounce.

```
In[49]:= Reflection[k_, ramp_][{x_, xp_, y_, yp_}] := Module[{gramp, gp, xpnew, ypnew},
           gramp = -ramp'[x];
           If[Not[NumberQ[gramp]],
            Print["Could not compute derivative "];
            Throw[$Failed]];
           gramp = {-ramp'[x], 1};
           If[ gramp.{xp, yp} == 0,
            Print["No reflection"];
            Throw[$Failed]];
           gp = {1, -1} Reverse[gramp];
           {xpnew, ypnew} = (k / (gramp.gramp)) (gp gp.{xp, yp} - gramp gramp.{xp, yp});
           {x, xpnew, y, ypnew}]
```

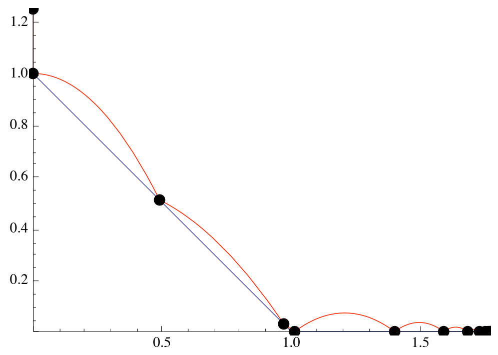
This defines the function that runs the bouncing ball simulation for a given reflection ratio, ramp, and starting position.

```
In[50]:= BouncingBall[k_, ramp_, {x0_, y0_}] :=
          Module[{data, end, bounces, xmin, xmax, ymin, ymax},
           If[y0 < ramp[x0],
            Print["Start above the ramp"];
            Return[$Failed]];
           data = Reap[
             Catch[Sow[{x0, y0}, "Bounces"];
              NestWhile[OneBounce[k, ramp], {0, x0, 0, y0, 0},
               Function[1 - #1[[1]] / #2[[1]] > 0.01], 2, 25]], _, Rule];
           end = data[[1, 1]];
           data = Last[data];
           bounces = ("Bounces" /. data);
           xmax = Max[bounces[[All, 1]]];
           xmin = Min[bounces[[All, 1]]];
           ymax = Max[bounces[[All, 2]]];
           ymin = Min[bounces[[All, 2]]];
           Show[{Plot[ramp[x], {x, xmin, xmax}, PlotRange → {{xmin, xmax}, {ymin, ymax}},
               Epilog → {PointSize[.025], Map[Point, bounces]},
               AspectRatio → (ymax - ymin) / (xmax - xmin)],
             ParametricPlot[Evaluate[{Piecewise["X" /. data], Piecewise["Y" /. data]}],
              {t, 0, end}, PlotStyle → RGBColor[1, 0, 0]]}]]
```

This is the example that is done in [SGT03].

```
In[51]:= ramp[x_] := If[x < 1, 1 - x, 0];
         BouncingBall[.7, ramp, {0, 1.25}]
```

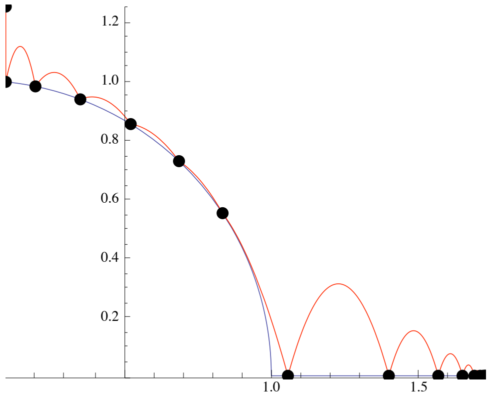
Out[52]=



The ramp is now defined to be a quarter circle.

```
In[53]:= circle[x_] := If[x < 1, Sqrt[1 - x^2], 0];
         BouncingBall[.7, circle, {.1, 1.25}]
```

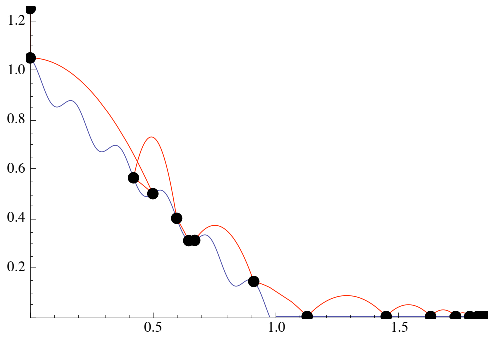
Out[54]=



This adds a slight waviness to the ramp.

```
In[55]:= wavyramp[x_] := If[x < 1, 1 - x + .05 Cos[11 Pi x] , 0];
         BouncingBall[.75, wavyramp, {0, 1.25}]
```

Out[56]=



Event Direction

Ordinary Differential Equation

This example illustrates the solution of the restricted three-body problem, a standard nonstiff test system of four equations. The example traces the path of a spaceship traveling around the moon and returning to the earth (see p. 246 of [SG75]). The ability to specify multiple events and the direction of the zero crossing is important.

The initial conditions have been chosen to make the orbit periodic. The value of μ corresponds to a spaceship traveling around the moon and the earth.

In[57]:=
$$\mu = \frac{1}{82.45};$$
$$\mu^* = 1 - \mu;$$
$$r_1 = \sqrt{(y_1[t] + \mu)^2 + y_2[t]^2};$$
$$r_2 = \sqrt{(y_1[t] - \mu^*)^2 + y_2[t]^2};$$
$$\text{eqns} = \Big\{\{y_1'[t] == y_3[t],\ y_1[0] == 1.2\},\ \{y_2'[t] == y_4[t],\ y_2[0] == 0\},$$
$$\Big\{y_3'[t] == 2\, y_4[t] + y_1[t] - \frac{\mu^*\,(y_1[t] + \mu)}{r_1^3} - \frac{\mu\,(y_1[t] - \mu^*)}{r_2^3},\ y_3[0] == 0\Big\},$$
$$\Big\{y_4'[t] == -2\, y_3[t] + y_2[t] - \frac{\mu^*\, y_2[t]}{r_1^3} - \frac{\mu\, y_2[t]}{r_2^3},$$
$$y_4[0] == -1.04935750983031990726\grave{}20.020923474937767\Big\}\Big\};$$

The event function is the derivative of the distance from the initial conditions. A local maximum or minimum occurs when the value crosses zero.

In[62]:=
```
ddist = 2 (y3[t] (y1[t] - 1.2) + y4[t] y2[t]);
```

There are two events, which for this example are the same. The first event (with `Direction 1`) corresponds to the point where the distance from the initial point is a local minimum, so that the spaceship returns to its original position. The event action is to store the time of the event in the variable `tfinal` and to stop the integration. The second event corresponds to a local maximum. The event action is to store the time that the spaceship is farthest from the starting position in the variable `tfar`.

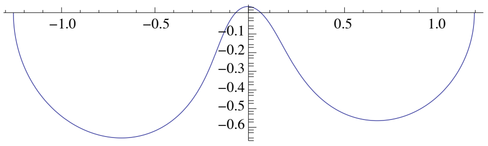
In[63]:=
```
sol = First[NDSolve[eqns, {y1, y2, y3, y4}, {t, ∞},
    Method → {"EventLocator",
      "Event" -> {ddist, ddist},
      "Direction" → {1, -1},
      "EventAction" :→ {Throw[tfinal = t, "StopIntegration"], tfar = t},
      Method → "ExplicitRungeKutta"}]]
```

Out[63]=
```
{y1 → InterpolatingFunction[{{0., 6.19217}}, <>], y2 → InterpolatingFunction[{{0., 6.19217}}, <>],
 y3 → InterpolatingFunction[{{0., 6.19217}}, <>], y4 → InterpolatingFunction[{{0., 6.19217}}, <>]}
```

The first two solution components are coordinates of the body of infinitesimal mass, so plotting one against the other gives the orbit of the body.

This displays one half-orbit when the spaceship is at the furthest point from the initial position.

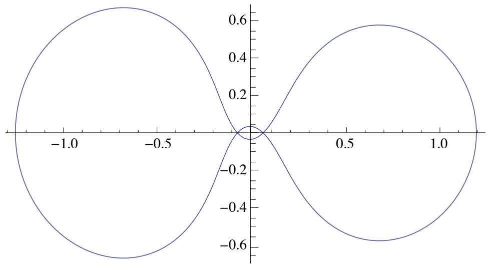
In[64]:=
```
ParametricPlot[{y1[t], y2[t]} /. sol, {t, 0, tfar}]
```

Out[64]=


This displays one complete orbit when the spaceship returns to the initial position.

```
In[65]:= ParametricPlot[{y₁[t], y₂[t]} /. sol, {t, 0, tfinal}]
```

Out[65]=



Delay Differential Equation

The following system models an infectious disease (see [HNW93], [ST00] and [ST01]).

```
In[66]:= system = {y1'[t] == -y1[t] y2[t - 1] + y2[t - 10],
            y1[t /; t ≤ 0] == 5, y2'[t] == y1[t] y2[t - 1] - y2[t],
            y2[t /; t ≤ 0] == 1 / 10, y3'[t] == y2[t] - y2[t - 10], y3[t /; t ≤ 0] == 1};
         vars = {y1[t], y2[t], y3[t]};
```

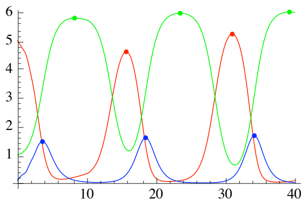
Collect the data for a local maximum of each component as the integration proceeds. A separate tag for Sow and Reap is used to distinguish the components.

```
In[68]:= data =
           Reap[
            sol = First[NDSolve[system, vars, {t, 0, 40},
                Method → {"EventLocator",
                  "Event" :→ {y1'[t], y2'[t], y3'[t]},
                  "EventAction" :→
                   {Sow[{t, y1[t]}, 1], Sow[{t, y2[t]}, 2], Sow[{t, y3[t]}, 3]},
                  "Direction" → {-1, -1, -1}}]],
            {1, 2, 3}
           ];
```

Display the local maxima together with the solution components.

```
In[69]:= colors = {{Red}, {Blue}, {Green}};
         plots = Plot[Evaluate[vars /. sol], {t, 0, 40}, PlotStyle → colors];
         max = ListPlot[Part[data, -1, All, 1], PlotStyle → colors];
         Show[plots, max]
```

Out[72]=



Discontinuous Equations and Switching Functions

In many applications the function in a differential system may not be analytic or continuous everywhere.

A common discontinuous problem that arises in practice involves a switching function g:

$$y' = \begin{cases} f_I(t, y) & \text{if} \quad g(t, y) > 0 \\ f_{II}(t, y) & \text{if} \quad g(t, y) < 0 \end{cases}$$

In order to illustrate the difficulty in crossing a discontinuity, consider the following example [GØ84] (see also [HNW93]):

$$y' = \begin{cases} t^2 + 2y^2 & \text{if} \quad \left(t + \frac{1}{20}\right)^2 + \left(y + \frac{3}{20}\right)^2 \le 1 \\ 2\,t\text{^}2 + 3\,y[t]\text{^}2 - 2 & \text{if} \quad \left(t + \frac{1}{20}\right)^2 + \left(y + \frac{3}{20}\right)^2 > 1 \end{cases}$$

Here is the input for the entire system. The switching function is assigned to the symbol `event`, and the function defining the system depends on the sign of the switching function.

```
In[73]:= t0 = 0;
         ics0 = 3/10;
         event = (t + 1/20)^2 + (y[t] + 3/20)^2 - 1;
         system = {y'[t] == If[event <= 0, t^2 + 2 y[t]^2, 2 t^2 + 3 y[t]^2 - 2], y[t0] == ics0};
```

The symbol `odemethod` is used to indicate the numerical method that should be used for the integration. For comparison, you might want to define a different method, such as "`ExplicitRungeKutta`", and rerun the computations in this section to see how other methods behave.

```
In[77]:= odemethod = Automatic;
```

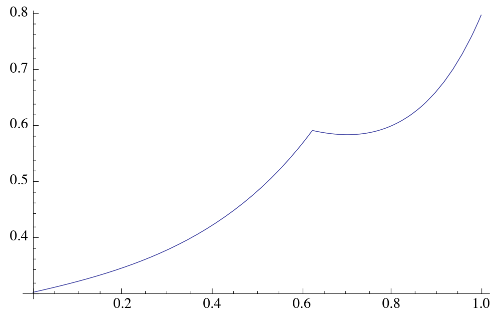
This solves the system on the interval [0, 1] and collects data for the mesh points of the integration using `Reap` and `Sow`.

```
In[78]:= data = Reap[
            sol = y[t] /. First[NDSolve[system, y, {t, t0, 1},
                 Method → odemethod, MaxStepFraction → 1, StepMonitor :→ Sow[t]]]
           ][[2, 1]];
         sol

Out[79]= InterpolatingFunction[{{0., 1.}}, <>][t]
```

Here is a plot of the solution.

```
In[80]:= dirsol = Plot[sol, {t, t0, 1}]
```

Out[80]=



Despite the fact that a solution has been obtained, it is not clear whether it has been obtained efficiently.

The following example shows that the crossing of the discontinuity presents difficulties for the numerical solver.

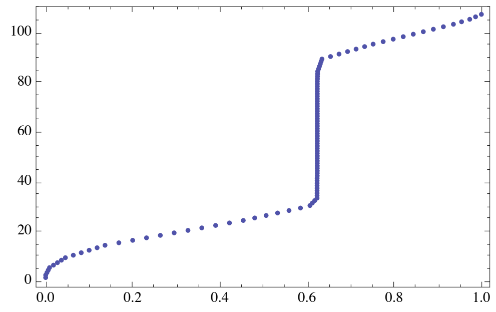
This defines a function that displays the mesh points of the integration together with the number of integration steps that are taken.

```
In[81]:= StepPlot[data_, opts___?OptionQ] :=
           Module[{sdata},
            sdata = Transpose[{data, Range[Length[data]]}];
            ListPlot[sdata, opts, Axes → False, Frame → True, PlotRange → All]
           ];
```

As the integration passes the discontinuity (near 0.6 in value), the integration method runs into difficulty, and a large number of small steps are taken—a number of rejected steps can also sometimes be observed.

```
In[82]:= StepPlot[data]
```

Out[82]=



One of the most efficient methods of crossing a discontinuity is to break the integration by restarting at the point of discontinuity.

The following example shows how to use the `"EventLocator"` method to accomplish this.

This numerically integrates the first part of the system up to the point of discontinuity. The switching function is given as the event. The direction of the event is restricted to a change from negative to positive. When the event is found, the solution and the time of the event are stored by the event action.

```
In[83]:= system1 = {y'[t] == t^2 + 2 y[t]^2, y[t0] == ics0};

         data1 = Reap[sol1 = y[t] /. First[NDSolve[system1, y, {t, t0, 1},
                 Method → {"EventLocator", "Event" -> event, Direction → 1,
                   EventAction :→ Throw[t1 = t; ics1 = y[t]; , "StopIntegration"],
                   Method → odemethod}, MaxStepFraction → 1, StepMonitor :→ Sow[t]]]
            ][[2, 1]];
         sol1
Out[85]= InterpolatingFunction[{{0., 0.623418}}, <>][t]
```

Using the discontinuity found by the "EventLocator" method as a new initial condition, the integration can now be continued.

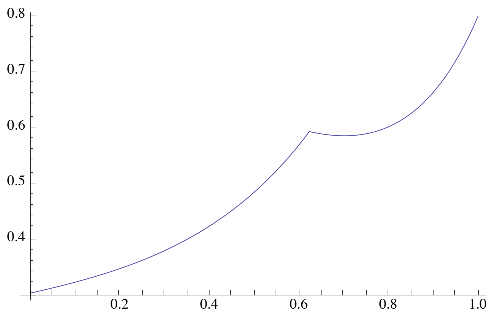
This defines a system and initial condition, solves the system numerically, and collects the data used for the mesh points.

```
In[86]:= system2 = {y'[t] == 2 t^2 + 3 y[t]^2 - 2, y[t1] == ics1};

         data2 = Reap[
              sol2 = y[t] /. First[NDSolve[system2, y, {t, t1, 1},
                 Method → odemethod, MaxStepFraction → 1, StepMonitor :→ Sow[t]]]
            ][[2, 1]];
         sol2
Out[88]= InterpolatingFunction[{{0.623418, 1.}}, <>][t]
```

A plot of the two solutions is very similar to that obtained by solving the entire system at once.

```
In[89]:= evsol = Plot[If[t ≤ t1, sol1, sol2], {t, 0, 1}]
```

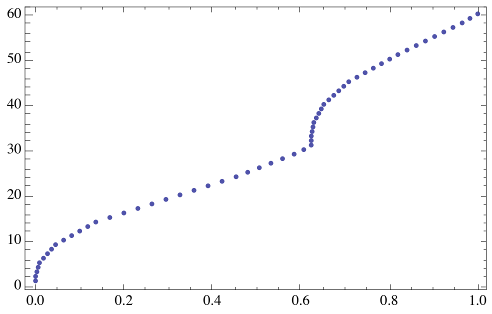
Out[89]=



Examining the mesh points, it is clear that far fewer steps were taken by the method and that the problematic behavior encountered near the discontinuity has been eliminated.

```
In[90]:= StepPlot[Join[data1, data2]]
```

Out[90]=



The value of the discontinuity is given as 0.6234 in [HNW93], which coincides with the value found by the "EventLocator" method.

In this example it is possible to analytically solve the system and use a numerical method to check the value.

The solution of the system up to the discontinuity can be represented in terms of Bessel and gamma functions.

```
In[91]:= dsol = FullSimplify[First[DSolve[system1, y[t], t]]]
```

Out[91]= $\left\{y[t] \to \left(t \left(3\,\text{BesselJ}\left[-\frac{3}{4}, \frac{t^2}{\sqrt{2}}\right] \text{Gamma}\left[\frac{1}{4}\right] + 10 \times 2^{1/4}\,\text{BesselJ}\left[\frac{3}{4}, \frac{t^2}{\sqrt{2}}\right] \text{Gamma}\left[\frac{3}{4}\right]\right)\right) \Big/ \left(\sqrt{2} \left(-3\,\text{BesselJ}\left[\frac{1}{4}, \frac{t^2}{\sqrt{2}}\right] \text{Gamma}\left[\frac{1}{4}\right] + 10 \times 2^{1/4}\,\text{BesselJ}\left[-\frac{1}{4}, \frac{t^2}{\sqrt{2}}\right] \text{Gamma}\left[\frac{3}{4}\right]\right)\right)\right\}$

Substituting in the solution into the switching function, a local minimization confirms the value of the discontinuity.

```
In[92]:= FindRoot[event /. dsol, {t, 3 / 5}]
Out[92]= {t → 0.623418}
```

Avoiding Wraparound in PDEs

Many evolution equations model behavior on a spatial domain that is infinite or sufficiently large to make it impractical to discretize the entire domain without using specialized discretization methods. In practice, it is often the case that it is possible to use a smaller computational domain for as long as the solution of interest remains localized.

In situations where the boundaries of the computational domain are imposed by practical considerations rather than the actual model being studied, it is possible to pick boundary conditions appropriately. Using a pseudospectral method with periodic boundary conditions can make it possible to increase the extent of the computational domain because of the superb resolution of the periodic pseudospectral approximation. The drawback of periodic boundary conditions is that signals that propagate past the boundary persist on the other side of the domain, affecting the solution through wraparound. It is possible to use an absorbing layer near the boundary to minimize these effects, but it is not always possible to completely eliminate them.

The sine-Gordon equation turns up in differential geometry and relativistic field theory. This example integrates the equation, starting with a localized initial condition that spreads out. The periodic pseudospectral method is used for the integration. Since no absorbing layer has been instituted near the boundaries, it is most appropriate to stop the integration once wraparound becomes significant. This condition is easily detected with event location using the `"EventLocator"` method.

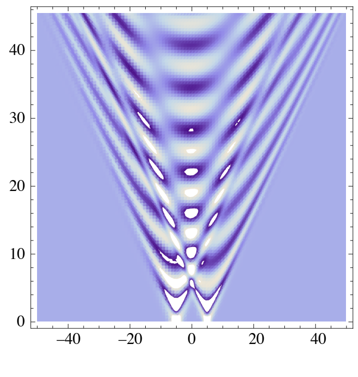
The integration is stopped when the size of the solution at the periodic wraparound point crosses a threshold of 0.01, beyond which the form of the wave would be affected by periodicity.

```
In[93]:= Timing[sgsol = First[NDSolve[{∂t,t u[t, x] == ∂x,x u[t, x] - Sin[u[t, x]],
              u[0, x] == ⅇ^-(x-5)^2 + ⅇ^-(x+5)^2/2, u^(1,0)[0, x] == 0, u[t, -50] == u[t, 50]},
            u, {t, 0, 1000}, {x, -50, 50}, Method → {"MethodOfLines",
              "SpatialDiscretization" → {"TensorProductGrid",
                "DifferenceOrder" -> "Pseudospectral"},
              Method → {"EventLocator", "Event" :→ Abs[u[t, -50]] - 0.01,
                "EventLocationMethod" -> "StepBegin"}}]]]

Out[93]= {0.301953, {u → InterpolatingFunction[{{0., 45.5002}, {-50., 50.}}, <>]}}
```

This extracts the ending time from the `InterpolatingFunction` object and makes a plot of the computed solution. You can see that the integration has been stopped just as the first waves begin to reach the boundary.

```
In[94]:= end = InterpolatingFunctionDomain[u /. sgsol][[1, -1]];
         DensityPlot[u[t, x] /. sgsol, {x, -50, 50},
          {t, 0, end}, Mesh → False, PlotPoints → 100]
```

Out[95]=



The "DiscretizedMonitorVariables" option affects the way the event is interpreted for PDEs; with the setting True, $u[t, x]$ is replaced by a vector of discretized values. This is much more efficient because it avoids explicitly constructing the InterpolatingFunction to evaluate the event.

```
In[96]:= Timing[sgsol = First[NDSolve[{∂t,t u[t, x] == ∂x,x u[t, x] - Sin[u[t, x]],
             u[0, x] == e^-(x-5)^2 + e^-(x+5)^2/2, u^(1,0)[0, x] == 0, u[t, -50] == u[t, 50]},
           u, {t, 0, 1000}, {x, -50, 50}, Method → {"MethodOfLines",
             "DiscretizedMonitorVariables" → True,
             "SpatialDiscretization" →
              {"TensorProductGrid", "DifferenceOrder" -> "Pseudospectral"},
             Method → {"EventLocator", "Event" :→ Abs[First[u[t, x]]] - 0.01,
               "EventLocationMethod" -> "StepBegin"}}]]]
Out[96]= {0.172973, {u → InterpolatingFunction[{{0., 45.5002}, {-50., 50.}}, <>]}}
```

Performance Comparison

The following example constructs a table making a comparison for two different integration methods.

This defines a function that returns the time it takes to compute a solution of a mildly damped pendulum equation up to the point at which the bob has momentarily been at rest 1000 times.

```
In[97]:= EventLocatorTiming[locmethod_, odemethod_] := Block[{Second = 1, y, t, p = 0},
            First[
             Timing[NDSolve[{y''[t] + 1/1000 y'[t] + Sin[y[t]] == 0, y[0] == 3, y'[0] == 0},
               y, {t, ∞}, Method → {"EventLocator", "Event" → y'[t],
                 "EventAction" :→ If[p++ ≥ 1000, Throw[end = t, "StopIntegration"]],
                 "EventLocationMethod" → locmethod, "Method" → odemethod},
               MaxSteps → ∞]]]
           ];
```

This uses the function to make a table comparing the different location methods for two different ODE integration methods.

```
In[98]:= elmethods = {"StepBegin", "StepEnd", "LinearInterpolation",
            {"Brent", "SolutionApproximation" -> "CubicHermiteInterpolation"}, Automatic};
         odemethods = {Automatic, "ExplicitRungeKutta"};
         TableForm[Outer[EventLocatorTiming, elmethods, odemethods, 1],
          TableHeadings → {elmethods, odemethods}]
```

Out[100]//TableForm=

	Automatic	ExplicitRungeKutta
StepBegin	0.234964	0.204969
StepEnd	0.218967	0.205968
LinearInterpolation	0.221967	0.212967
{Brent, SolutionApproximation → CubicHermiteInterpolation}	0.310953	0.314952
Automatic	0.352947	0.354946

While simple step begin/end and linear interpolation location are essentially the same low cost, the better location methods are more expensive. The default location method is particularly expensive for the explicit Runge-Kutta method because it does not yet support a continuous output formula—it therefore needs to repeatedly invoke the method with different step sizes during the local minimization.

It is worth noting that, often, a significant part of the extra time for computing events arises from the need to evaluate the event functions at each time step to check for the possibility of a sign change.

```
In[101]:= TableForm[
   {Map[
     Block[{Second = 1, y, t, p = 0},
       First[Timing[NDSolve[{y''[t] + 1/1000 y'[t] + Sin[y[t]] == 0,
           y[0] == 3, y'[0] == 0}, y, {t, end}, Method → #, MaxSteps → ∞]]]] &,
     odemethods
    ]},
   TableHeadings → {None, odemethods}]
```

```
Out[101]//TableForm= Automatic  ExplicitRungeKutta
                     0.105984   0.141979
```

An optimization is performed for event functions involving only the independent variable. Such events are detected automatically at initialization time. For example, this has the advantage that interpolation of the solution of the dependent variables is not carried out at each step of the local optimization search—it is deferred until the value of the independent variable has been found.

Limitations

One limitation of the event locator method is that since the event function is only checked for sign changes over a step interval, if the event function has multiple roots in a step interval, all or some of the events may be missed. This typically only happens when the solution to the ODE varies much more slowly than the event function. When you suspect that this may have occurred, the simplest solution is to decrease the maximum step size the method can take by using the `MaxStepSize` option to `NDSolve`. More sophisticated approaches can be taken, but the best approach depends on what is being computed. An example follows that demonstrates the problem and shows two approaches for fixing it.

This should compute the number of positive integers less than e^5 (there are 148). However, most are missed because the method is taking large time steps because the solution $x[t]$ is so simple.

```
In[102]:= Block[{n = 0}, NDSolve[{y'[t] == y[t], y[-1] == e^-1}, y, {t, 5},
            Method → {"EventLocator", "Event" → Sin[π y[t]], "EventAction" :→ n++}]; n]

Out[102]= 18
```

This restricts the maximum step size so that all the events are found.

```
In[103]:= Block[{n = 0}, NDSolve[{y'[t] == y[t], y[-1] == e^-1}, y, {t, 5},
            Method → {"EventLocator", "Event" → Sin[π y[t]], "EventAction" :→ n++},
            MaxStepSize → 0.001]; n]

Out[103]= 148
```

It is quite apparent from the nature of the example problem that if the endpoint is increased, it is likely that a smaller maximum step size may be required. Taking very small steps everywhere is quite inefficient. It is possible to introduce an adaptive time step restriction by setting up a variable that varies on the same time scale as the event function.

This introduces an additional function to integrate that is the event function. With this modification and allowing the method to take as many steps as needed, it is possible to find the correct value up to $t = 10$ in a reasonable amount of time.

```
In[104]:= Block[{n = 0}, NDSolve[
            {y'[t] == y[t], y[-1] == e^-1, z'[t] == D[Sin[π y[t]], t], z[-1] == Sin[π e^-1]},
            {y, z}, {t, 10}, Method → {"EventLocator", "Event" → z[t], "EventAction" :→ n++},
            MaxSteps → ∞]; n]

Out[104]= 22 026
```

Option Summary

"EventLocator" Options

option name	*default value*	
`“Direction“`	`All`	the direction of zero crossing to allow for the event; 1 means from negative to positive, -1 means from positive to negative, and `All` includes both directions
`“Event“`	`None`	an expression that defines the event; an event occurs at points where substituting the numerical values of the problem variables makes the expression equal to zero
`“EventAction“`	`Throw[Null, “StopIntegration“]`	what to do when an event occurs: problem variables are substituted with their numerical values at the event; in general, you need to use `RuleDelayed` (:→) to prevent the option from being evaluated except with numerical values
`“EventLocationMethod“`	`Automatic`	the method to use for refining the location of a given event
`“Method“`	`Automatic`	the method to use for integrating the system of ODEs

`“EventLocator“` method options.

"EventLocationMethod" Options

`“Brent“`	use `FindRoot` with `Method -> “Brent“` to locate the event; this is the default with the setting `Automatic`
`“LinearInterpolation“`	locate the event time using linear interpolation; cubic Hermite interpolation is then used to find the solution at the event time
`“StepBegin“`	the event is given by the solution at the beginning of the step
`“StepEnd“`	the event is given by the solution at the end of the step

Settings for the `“EventLocationMethod“` option.

"Brent" Options

option name	*default value*	
`“MaxIterations“`	`100`	the maximum number of iterations to use for locating an event within a step of the method
`“AccuracyGoal“`	`Automatic`	accuracy goal setting passed to `FindRoot`; if `Automatic`, the value passed to `FindRoot` is based on the local error setting for `NDSolve`
`“PrecisionGoal“`	`Automatic`	precision goal setting passed to `FindRoot`; if `Automatic`, the value passed to `FindRoot` is based on the local error setting for `NDSolve`
`“SolutionApproximation“`	`Automatic`	how to approximate the solution for evaluating the event function during the refinement process; can be `Automatic` or `“CubicHermiteInterpolation“`

Options for event location method `“Brent“`.

"Extrapolation" Method for NDSolve

Introduction

Extrapolation methods are a class of arbitrary-order methods with automatic order and step-size control. The error estimate comes from computing a solution over an interval using the same method with a varying number of steps and using extrapolation on the polynomial that fits through the computed solutions, giving a composite higher-order method [BS64]. At the same time, the polynomials give a means of error estimation.

Typically, for low precision, the extrapolation methods have not been competitive with Runge-Kutta-type methods. For high precision, however, the arbitrary order means that they can be arbitrarily faster than fixed-order methods for very precise tolerances.

The order and step-size control are based on the codes `odex.f` and `seulex.f` described in [HNW93] and [HW96].

This loads packages that contain some utility functions for plotting step sequences and some predefined problems.

```
In[3]:= Needs[“DifferentialEquations`NDSolveProblems`“];
        Needs[“DifferentialEquations`NDSolveUtilities`“];
```

"Extrapolation"

The method `“DoubleStep“` performs a single application of Richardson's extrapolation for any one-step integration method and is described within "DoubleStep Method for NDSolve".

`“Extrapolation“` generalizes the idea of Richardson's extrapolation to a sequence of refinements.

Consider a differential system

$$y'(t) = f(t,\, y(t)),\; y(t_0) = y_0. \tag{1}$$

Let $H > 0$ be a basic step size; choose a monotonically increasing sequence of positive integers

$$n_1 < n_2 < n_3 < \cdots < n_k$$

and define the corresponding step sizes

$$h_1 > h_2 > h_3 > \cdots > h_k$$

by

$$h_i = \frac{H}{n_i},\; i = 1,\, 2,\, \ldots,\, k.$$

Choose a numerical method of order p and compute the solution of the initial value problem by carrying out n_i steps with step size h_i to obtain:

$$T_{i,1} = y_{h_i}(t_o + H),\; i = 1,\, 2,\, \ldots,\, k.$$

Extrapolation is performed using the Aitken-Neville algorithm by building up a table of values:

$$T_{i,j} = T_{i,j-1} + \frac{T_{i,j-1} - T_{i-1,j-1}}{\left(\frac{n_i}{n_{i-j+1}}\right)^w - 1},\; i = 2,\, \ldots,\, k,\; j = 2,\, \ldots,\, i, \tag{2}$$

where w is either 1 or 2 depending on whether the base method is symmetric under extrapolation.

A dependency graph of the values in (2) illustrates the relationship:

$$\begin{tikzcd}
T_{11} & & & \\
T_{21} & T_{22} \arrow[l] \arrow[ul] & & \\
T_{31} & T_{32} \arrow[l] \arrow[ul] & T_{33} \arrow[l] \arrow[ul] & \\
T_{41} & T_{42} \arrow[l] \arrow[ul] & T_{43} \arrow[l] \arrow[ul] & T_{44} \arrow[l] \arrow[ul] \\
\cdots \quad \cdots & \cdots \quad \cdots & \cdots \quad \cdots & \cdots \quad \cdots
\end{tikzcd}$$

Considering $k = 2$, $n_1 = 1$, $n_2 = 2$ is equivalent to Richardson's extrapolation.

For non-stiff problems the order of $T_{k,k}$ in (2) is $p + (k-1)w$. For stiff problems the analysis is more complicated and involves the investigation of perturbation terms that arise in singular perturbation problems [HNW93, HW96].

Extrapolation Sequences

Any extrapolation sequence can be specified in the implementation. Some common choices are as follows.

This is the Romberg sequence.

```
In[5]:= NDSolve`RombergSequenceFunction[1, 10]
Out[5]= {1, 2, 4, 8, 16, 32, 64, 128, 256, 512}
```

This is the Bulirsch sequence.

```
In[6]:= NDSolve`BulirschSequenceFunction[1, 10]
Out[6]= {1, 2, 3, 4, 6, 8, 12, 16, 24, 32}
```

This is the harmonic sequence.

```
In[7]:= NDSolve`HarmonicSequenceFunction[1, 10]
Out[7]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

A sequence that satisfies $\left(n_i / n_{i-j+1}\right)^w \geq 2$ has the effect of minimizing the roundoff errors for an order-p base integration method.

For a base method of order two, the first entries in the sequence are given by the following.

```
In[8]:=  NDSolve`OptimalRoundingSequenceFunction[1, 10, 2]

Out[8]= {1, 2, 3, 5, 8, 12, 17, 25, 36, 51}
```

Here is an example of adding a function to define the harmonic sequence where the method order is an optional pattern.

```
In[9]:=  Default[myseqfun, 3] = 1;

         myseqfun[n1_, n2_, p_.] := Range[n1, n2]
```

The sequence with lowest cost is the Harmonic sequence, but this is not without problems since rounding errors are not damped.

Rounding Error Accumulation

For high-order extrapolation an important consideration is the accumulation of rounding errors in the Aitken-Neville algorithm (2).

As an example consider Exercise 5 of Section II.9 in [HNW93].

Suppose that the entries $T_{11}, T_{21}, T_{31}, \ldots$ are disturbed with rounding errors $\epsilon, -\epsilon, \epsilon, \ldots$ and compute the propagation of these errors into the extrapolation table.

Due to the linearity of the extrapolation process (2), suppose that the $T_{i,j}$ are equal to zero and take $\epsilon = 1$.

This shows the evolution of the Aitken-Neville algorithm (2) on the initial data using the harmonic sequence and a symmetric order-two base integration method, $w = p = 2$.

```
 1.
−1.  −1.66667
 1.     2.6      3.13333
−1.  −3.57143  −5.62857  −6.2127
 1.   4.55556   9.12698  11.9376   12.6938
−1.  −5.54545  −13.6263  −21.2107  −25.3542  −26.4413
 1.   6.53846   19.1259   35.0057   47.6544    54.144   55.8229
−1.  −7.53333  −25.6256  −54.3125  −84.0852  −105.643  −116.295  −119.027
```

Hence, for an order-sixteen method approximately two decimal digits are lost due to rounding error accumulation.

This model is somewhat crude because, as you will see later, it is more likely that rounding errors are made in $T_{i+1,1}$ than in $T_{i,1}$ for $i \geq 1$.

Rounding Error Reduction

It seems worthwhile to look for approaches that can reduce the effect of rounding errors in high-order extrapolation.

Selecting a different step sequence to diminish rounding errors is one approach, although the drawback is that the number of integration steps needed to form the $T_{i,1}$ in the first column of the extrapolation table requires more work.

Some codes, such as STEP, take active measures to reduce the effect of rounding errors for stringent tolerances [SG75].

An alternative strategy, which does not appear to have received a great deal of attention in the context of extrapolation, is to modify the base-integration method in order to reduce the magnitude of the rounding errors in floating-point operations. This approach, based on ideas that dated back to [G51], and used to good effect for the two-body problem in [F96b] (for background see also [K65], [M65a], [M65b], [V79]), is explained next.

Base Methods

The following methods are the most common choices for base integrators in extrapolation.

- `"ExplicitEuler"`
- `"ExplicitMidpoint"`
- `"ExplicitModifiedMidpoint"` (Gragg smoothing step (1))
- `"LinearlyImplicitEuler"`
- `"LinearlyImplicitMidpoint"` (Bader-Deuflhard formulation without smoothing step (1))
- `"LinearlyImplicitModifiedMidpoint"` (Bader-Deuflhard formulation with smoothing step (1))

For efficiency, these have been built into `NDSolve` and can be called via the `Method` option as individual methods.

The implementation of these methods has a special interpretation for multiple substeps within `"DoubleStep"` and `"Extrapolation"`.

The `NDSolve`. framework for one step methods uses a formulation that returns the increment or update to the solution. This is advantageous for geometric numerical integration where numerical errors are not damped over long time integrations. It also allows the application of efficient correction strategies such as compensated summation. This formulation is also useful in the context of extrapolation.

The methods are now described together with the increment reformulation that is used to reduce rounding error accumulation.

Multiple Euler Steps

Given t_0, y_0 and H, consider a succession of $n = n_k$ integration steps with step size $h = H/n$ carried out using Euler's method:

$$\begin{array}{ccc} y_1 & = & y_0 + h\, f(t_0,\, y_0) \\ y_2 & = & y_1 + h\, f(t_1,\, y_1) \\ y_3 & = & y_2 + h\, f(t_2,\, y_2) \\ \vdots & \vdots & \vdots \\ y_n & = & y_{n-1} + h\, f(t_{n-1},\, y_{n-1}) \end{array} \tag{1}$$

where $t_i = t_0 + i\, h$.

Correspondence with Explicit Runge-Kutta Methods

It is well-known that, for certain base integration schemes, the entries $T_{i,j}$ in the extrapolation table produced from (2) correspond to explicit Runge-Kutta methods (see Exercise 1, Section II.9 in [HNW93]).

For example, (1) is equivalent to an n-stage explicit Runge-Kutta method:

$$\begin{array}{ccc} k_i & = & f\left(t_0 + c_i\, H,\, y_0 + H \sum_{j=1}^{n} a_{i,j}\, k_j\right), \quad i = 1,\, \ldots,\, n, \\ y_n & = & y_0 + H \sum_{i=1}^{n} b_i\, k_i \end{array} \tag{1}$$

where the coefficients are represented by the Butcher table:

$$\begin{array}{c|cccc} 0 & & & & \\ 1/n & 1/n & & & \\ \vdots & \vdots & \ddots & & \\ (n-1)/n & 1/n & \cdots & 1/n & \\ \hline & 1/n & \cdots & 1/n & 1/n \end{array} \tag{2}$$

Reformulation

Let $\Delta y_n = y_{n+1} - y_n$. Then the integration (1) can be rewritten to reflect the correspondence with an explicit Runge-Kutta method (1, 2) as:

$$\begin{array}{ccc} \Delta y_0 & = & h\,f(t_0,\, y_0) \\ \Delta y_1 & = & h\,f(t_1,\, y_0 + \Delta y_0) \\ \Delta y_2 & = & h\,f(t_2,\, y_0 + (\Delta y_0 + \Delta y_1)) \\ \vdots & \vdots & \vdots \\ \Delta y_{n-1} & = & h\,f\left(t_{n-1},\, y_0 + \left(\Delta y_0 + \Delta y_1 + \cdots + \Delta \mathrm{y}_{n-2}\right)\right) \end{array} \tag{1}$$

where terms in the right-hand side of (1) are now considered as departures from the *same value* y_0.

The Δy_i in (1) correspond to the $h\,k_i$ in (1).

Let $\Sigma\Delta\, y_n = \sum_{i=0}^{n-1} \Delta\, y_i$; then the required result can be recovered as:

$$y_n = y_0 + \Sigma\Delta\, y_n \tag{2}$$

Mathematically the formulations (1) and (1, 2) are equivalent. For $n > 1$, however, the computations in (1) have the advantage of accumulating a sum of smaller $O(h)$ quantities, or increments, which reduces rounding error accumulation in finite-precision floating-point arithmetic.

Multiple Explicit Midpoint Steps

Expansions in even powers of h are extremely important for an efficient implementation of Richardson's extrapolation and an elegant proof is given in [S70].

Consider a succession of integration steps $n = 2\,n_k$ with step size $h = H/n$ carried out using one Euler step followed by multiple explicit midpoint steps:

$$\begin{array}{ccc} y_1 & = & y_0 + h\,f(t_0,\, y_0) \\ y_2 & = & y_0 + 2\,h\,f(t_1,\, y_1) \\ y_3 & = & y_1 + 2\,h\,f(t_2,\, y_2) \\ \vdots & \vdots & \vdots \\ y_n & = & y_{n-2} + 2\,h\,f(t_{n-1},\, y_{n-1}) \end{array} \tag{1}$$

If (1) is computed with $2 n_k - 1$ midpoint steps, then the method has a symmetric error expansion ([G65], [S70]).

Reformulation

Reformulation of (1) can be accomplished in terms of increments as:

$$\begin{array}{ccc} \Delta y_0 & = & h f(t_0, y_0) \\ \Delta y_1 & = & 2 h f(t_1, y_0 + \Delta y_0) - \Delta y_0 \\ \Delta y_2 & = & 2 h f(t_2, y_0 + (\Delta y_0 + \Delta y_1)) - \Delta y_1 \\ \vdots & \vdots & \vdots \\ \Delta y_{n-1} & = & 2 h f(t_{n-1}, y_0 + (\Delta y_0 + \Delta y_1 + \cdots + \Delta y_{n-2})) - \Delta y_{n-2} \end{array} \tag{1}$$

Gragg's Smoothing Step

The *smoothing step* of Gragg has its historical origins in the weak stability of the explicit midpoint rule:

$$S y_h(n) = 1/4 (y_{n-1} + 2 y_n + y_{n+1}) \tag{1}$$

In order to make use of (1), the formulation (1) is computed with $2 n_k$ steps. This has the advantage of increasing the stability domain and evaluating the function at the end of the basic step [HNW93].

Notice that because of the construction, a sum of increments is available at the end of the algorithm together with two consecutive increments. This leads to the following formulation:

$$S \Delta y_h(n) = S y_h(n) - y_0 = \Sigma\Delta y_n + 1/4 (\Delta y_n - \Delta y_{n-1}). \tag{2}$$

Moreover (2) has an advantage over (1) in finite-precision arithmetic because the values y_i, which typically have a larger magnitude than the increments Δy_i, do not contribute to the computation.

Gragg's smoothing step is not of great importance if the method is followed by extrapolation, and Shampine proposes an alternative smoothing procedure that is slightly more efficient [SB83].

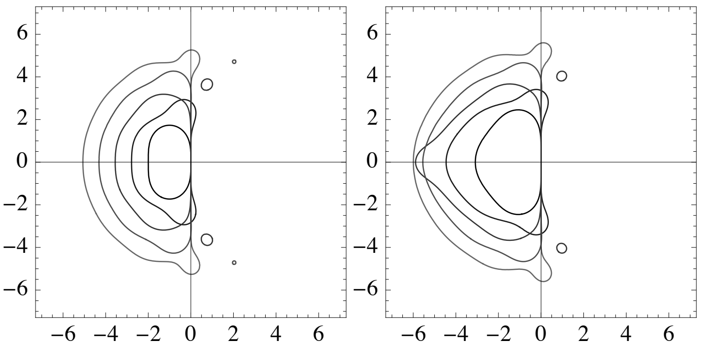
The method `"ExplicitMidpoint"` uses $2 n_k - 1$ steps and `"ExplicitModifiedMidpoint"` uses $2 n_k$ steps followed by the smoothing step (2).

Stability Regions

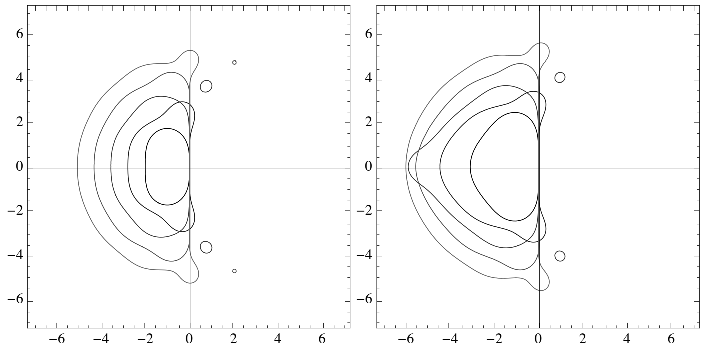
The following figures illustrate the effect of the smoothing step on the linear stability domain (carried out using the package `FunctionApproximations.m`).

Linear stability regions for $T_{i,i}$, $i = 1, \ldots, 5$ for the explicit midpoint rule (left) and the explicit midpoint rule with smoothing (right).

In[11]:=



Out[11]=



Since the precise stability boundary can be complicated to compute for an arbitrary base method, a simpler approximation is used. For an extrapolation method of order p, the intersection with the negative real axis is considered to be the point at which:

$$\left| \sum_{i=1}^{p} \frac{z^i}{i!} \right| = 1$$

The stabillity region is approximated as a disk with this radius and origin (0,0) for the negative half-plane.

Implicit Differential Equations

A generalization of the differential system (1) arises in many situations such as the spatial discretization of parabolic partial differential equations:

$$M\, y'(t) = f(t,\, y(t)),\ y(t_0) = y_0. \tag{1}$$

where M is a constant matrix that is often referred to as the *mass matrix*.

Base methods in extrapolation that involve the solution of linear systems of equations can easily be modified to solve problems of the form (1).

Multiple Linearly Implicit Euler Steps

Increments arise naturally in the description of many semi-implicit and implicit methods. Consider a succession of integration steps carried out using the linearly implicit Euler method for the system (1) with $n = n_k$ and $h = H/n$.

$$\begin{array}{ccc}
(M - h\, J)\, \Delta\, y_0 & = & h\, f(t_0,\, y_0) \\
y_1 & = & y_0 + \Delta\, y_0 \\
(M - h\, J)\, \Delta\, y_1 & = & h\, f(t_1,\, y_1) \\
y_2 & = & y_1 + \Delta\, y_1 \\
(M - h\, J)\, \Delta\, y_2 & = & h\, f(t_2,\, y_2) \\
y_3 & = & y_2 + \Delta\, y_2 \\
\vdots & \vdots & \vdots \\
(M - h\, J)\, \Delta\, y_{n-1} & = & h\, f(t_{n-1},\, y_{n-1})
\end{array} \tag{1}$$

Here M denotes the mass matrix and J denotes the Jacobian of f:

$$J = \frac{\partial f}{\partial y}(t_0,\, y_0).$$

The solution of the equations for the increments in (1) is accomplished using a single LU decomposition of the matrix $M - h\, J$ followed by the solution of triangular linear systems for each right-hand side.

The desired result is obtained from (1) as:

$$y_n = y_{n-1} + \Delta\, y_{n-1}.$$

Reformulation

Reformulation in terms of increments as departures from y_0 can be accomplished as follows:

$$\begin{array}{ccc} (M - h\,J)\,\Delta\,y_0 & = & h\,f(t_0,\,y_0) \\ (M - h\,J)\,\Delta\,y_1 & = & h\,f(t_1,\,y_0 + \Delta\,y_0) \\ (M - h\,J)\,\Delta\,y_2 & = & h\,f(t_2,\,y_0 + (\Delta\,y_0 + \Delta\,y_1)) \\ \vdots & \vdots & \vdots \\ (M - h\,J)\,\Delta\,y_{n-1} & = & h\,f(t_{n-1},\,y_0 + (\Delta\,y_0 + \Delta\,y_1 + \cdots + \Delta\,y_{n-2})) \end{array} \tag{1}$$

The result for y_n using (1) is obtained from (2).

Notice that (1) and (1) are equivalent when $J = 0$, $M = I$.

Multiple Linearly Implicit Midpoint Steps

Consider one step of the linearly implicit Euler method followed by multiple linearly implicit midpoint steps with $n = 2\,n_k$ and $h = H/n$, using the formulation of Bader and Deuflhard [BD83]:

$$\begin{array}{ccc} (M - h\,J)\,\Delta\,y_0 & = & h\,f(t_0,\,y_0) \\ y_1 & = & y_0 + \Delta\,y_0 \\ (M - h\,J)\,(\Delta\,y_1 - \Delta\,y_0) & = & 2\,(h\,f(t_1,\,y_1) - \Delta\,y_0) \\ y_2 & = & y_1 + \Delta\,y_1 \\ (M - h\,J)\,(\Delta\,y_2 - \Delta\,y_1) & = & 2\,(h\,f(t_2,\,y_2) - \Delta\,y_1) \\ y_3 & = & y_2 + \Delta\,y_2 \\ \vdots & \vdots & \vdots \\ (M - h\,J)\,(\Delta\,y_{n-1} - \Delta\,y_{n-2}) & = & 2\,(h\,f(t_{n-1},\,y_{n-1}) - \Delta\,y_{n-2}) \end{array} \tag{1}$$

If (1) is computed for $2\,n_k - 1$ linearly implicit midpoint steps, then the method has a symmetric error expansion [BD83].

Reformulation

Reformulation of (1) in terms of increments can be accomplished as follows:

$$\begin{array}{ccc} (M - h\,J)\,\Delta\,y_0 & = & h\,f(t_0,\,y_0) \\ (M - h\,J)\,(\Delta\,y_1 - \Delta\,y_0) & = & 2\,(h\,f(t_1,\,y_0 + \Delta\,y_0) - \Delta\,y_0) \\ (M - h\,J)\,(\Delta\,y_2 - \Delta\,y_1) & = & 2\,(h\,f(t_2,\,y_0 + (\Delta\,y_0 + \Delta\,y_1)) - \Delta\,y_1) \\ \vdots & \vdots & \vdots \\ (M - h\,J)\,(\Delta\,y_{n-1} - \Delta\,y_{n-2}) & = & 2\,(h\,f(t_{n-1},\,y_0 + (\Delta\,y_0 + \Delta\,y_1 + \cdots + \Delta\,y_{n-2})) - \Delta\,y_{n-2}) \end{array} \tag{1}$$

Smoothing Step

An appropriate smoothing step for the linearly implicit midpoint rule is [BD83]:

$$S\,y_h(n) = \frac{1}{2}(y_{n-1} + y_{n+1}). \qquad (1)$$

Bader's smoothing step (1) rewritten in terms of increments becomes:

$$S\,\Delta\,y_h(n) = S\,y_h(n) - y_0 = \Sigma\Delta\,y_n + \frac{1}{2}(\Delta\,y_n - \Delta\,y_{n-1}). \qquad (2)$$

The required quantities are obtained when (1) is run with $2\,n_k$ steps.

The smoothing step for the linearly implicit midpoint rule has a different role from Gragg's smoothing for the explicit midpoint rule (see [BD83] and [SB83]). Since there is no weakly stable term to eliminate, the aim is to improve the asymptotic stability.

The method `"LinearlyImplicitMidpoint"` uses $2\,n_k - 1$ steps and `"LinearlyImplicitModifiedMidpoint"` uses $2\,n_k$ steps followed by the smoothing step (2).

Polynomial Extrapolation in Terms of Increments

You have seen how to modify $T_{i,1}$, the entries in the first column of the extrapolation table, in terms of increments.

However, for certain base integration methods, *each* of the $T_{i,j}$ corresponds to an explicit Runge-Kutta method.

Therefore, it appears that the correspondence has not yet been fully exploited and further refinement is possible.

Since the Aitken-Neville algorithm (2) involves linear differences, the entire extrapolation process can be carried out using increments.

This leads to the following modification of the Aitken-Neville algorithm:

$$\Delta\,T_{i,j} = \Delta\,T_{i,j-1} + \frac{\Delta\,T_{i,j-1} - \Delta\,T_{i-1,j-1}}{\left(\frac{n_i}{n_{i-j+1}}\right)^p - 1},\; i = 2, \ldots, k,\; j = 2, \ldots, i. \qquad (1)$$

The quantities $\Delta T_{i,j} = T_{i,j} - y_0$ in (1) can be computed iteratively, starting from the initial quantities $T_{i,1}$ that are obtained from the modified base integration schemes without adding the contribution from y_0.

The final desired value $T_{k,k}$ can be recovered as $\Delta T_{k,k} + y_0$.

The advantage is that the extrapolation table is built up using smaller quantities, and so the effect of rounding errors from subtractive cancellation is reduced.

Implementation Issues

There are a number of important implementation issues that should be considered, some of which are mentioned here.

Jacobian Reuse

The Jacobian is evaluated only once for all entries $T_{i,1}$ at each time step by storing it together with the associated time that it is evaluated. This also has the advantage that the Jacobian does not need to be recomputed for rejected steps.

Dense Linear Algebra

For dense systems, the LAPACK routines xyyTRF can be used for the LU decomposition and the routines xyyTRS for solving the resulting triangular systems [LAPACK99].

Adaptive Order and Work Estimation

In order to adaptively change the order of the extrapolation throughout the integration, it is important to have a measure of the amount of work required by the base scheme and extrapolation sequence.

A measure of the relative cost of function evaluations is advantageous.

The dimension of the system, preferably with a weighting according to structure, needs to be incorporated for linearly implicit schemes in order to take account of the expense of solving each linear system.

Stability Check

Extrapolation methods use a large basic step size that can give rise to some difficulties.

"Neither code can solve the van der Pol equation problem in a straightforward way because of overflow..." [S87].

Two forms of stability check are used for the linearly implicit base schemes (for further discussion, see [HW96]).

One check is performed during the extrapolation process. Let $err_j = \left\| T_{j,j-1} - T_{j,j} \right\|$.

> If $err_j \geq err_{j-1}$ for some $j \geq 3$, then recompute the step with $H = H/2$.

In order to interrupt computations in the computation of $T_{1,1}$, Deuflhard suggests checking if the Newton iteration applied to a fully implicit scheme would converge.

For the implicit Euler method this leads to consideration of:

$$\begin{aligned} (M - h\,J)\,\Delta_0 &= h\,f(t_0,\,y_0) \\ (M - h\,J)\,\Delta_1 &= h\,f(t_0,\,y_0 + \Delta_0) - \Delta_0 \end{aligned} \tag{1}$$

Notice that (1) differs from (1) only in the second equation. It requires finding the solution for a different right-hand side but no extra function evaluation.

For the implicit midpoint method, $\Delta_0 = \Delta\,y_0$ and $\Delta_1 = 1/2\,(\Delta\,y_1 - \Delta\,y_0)$, which simply requires a few basic arithmetic operations.

> If $\|\Delta_1\| \geq \|\Delta_0\|$ then the implicit iteration diverges, so recompute the step with $H = H/2$.

Increments are a more accurate formulation for the implementation of both forms of stability check.

Examples

Work-Error Comparison

For comparing different extrapolation schemes, consider an example from [HW96].

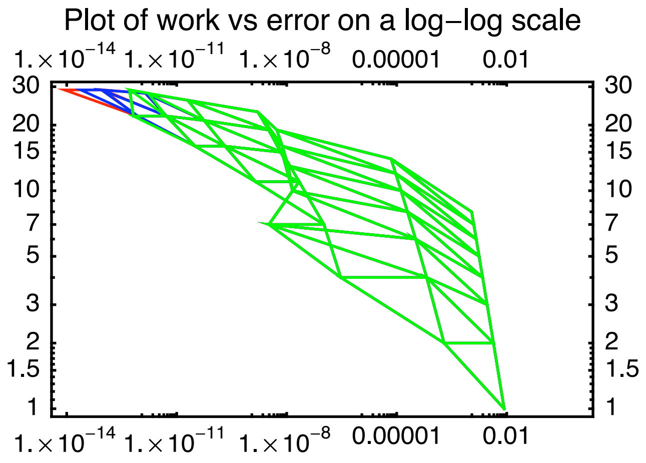
```
In[12]:= t0 = π / 6;
         h0 = 1 / 10;
         y0 = {2 / √3};
         eqs = {y'[t] == (-y[t] Sin[t] + 2 Tan[t]) y[t], y[t0] == y0};
         exactsol = y[t] /. First[DSolve[eqs, y[t], t]] /. t → t0 + h0;
         idata =  {{eqs, y[t], t}, h0, exactsol};
```

The exact solution is given by $y(t) = 1/\cos(t)$.

Increment Formulation

This example involves an eighth-order extrapolation of "`ExplicitEuler`" with the harmonic sequence. Approximately two digits of accuracy are gained by using the increment-based formulation throughout the extrapolation process.

- The results for the standard formulation (1) are depicted in green.
- The results for the increment formulation (1) followed by standard extrapolation (2) are depicted in blue.
- The results for the increment formulation (1) with extrapolation carried out on the increments using (1) are depicted in red.



Approximately two decimal digits of accuracy are gained by using the increment-based formulation throughout the extrapolation process.

This compares the relative error in the integration data that forms the initial column of the extrapolation table for the previous example.

Reference values were computed using software arithmetic with 32 decimal digits and converted to the nearest IEEE double-precision floating-point numbers, where an ULP signifies a Unit in the Last Place or Unit in the Last Position.

	T_{11}	T_{21}	T_{31}	T_{41}	T_{51}	T_{61}	T_{71}	T_{81}
Standard formulation	0	−1 ULP	0	1 ULP	0	1.5 ULPs	0	1 ULP
Increment formulation applied to the base method	0	0	0	0	1 ULP	0	0	1 ULP

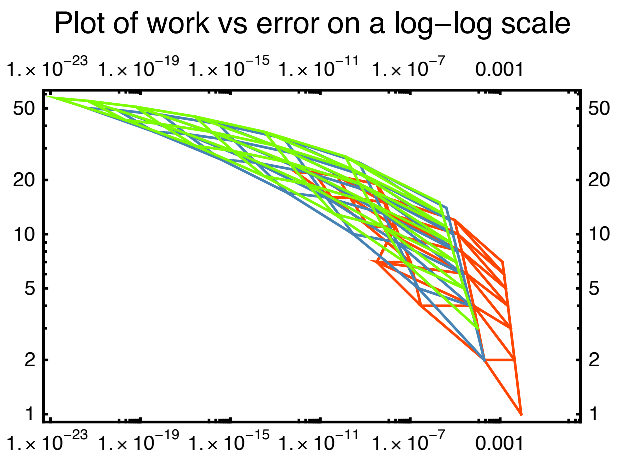
Notice that the rounding-error model that was used to motivate the study of rounding-error growth is limited because in practice, errors in $T_{i,1}$ can exceed 1 ULP.

The increment formulation used throughout the extrapolation process produces rounding errors in $T_{i,1}$ that are smaller than 1 ULP.

Method Comparison

This compares the work required for extrapolation based on `"ExplicitEuler"` (red), the `"ExplicitMidpoint"` (blue), and `"ExplicitModifiedMidpoint"` (green).

All computations are carried out using software arithmetic with 32 decimal digits.



Order Selection

Select a problem to solve.

```
In[32]:= system = GetNDSolveProblem["Pleiades"];
```

Define a monitor function to store the order and the time of evaluation.

```
In[33]:= OrderMonitor[t_, method_NDSolve`Extrapolation] :=
           Sow[{t, method["DifferenceOrder"]}];
```

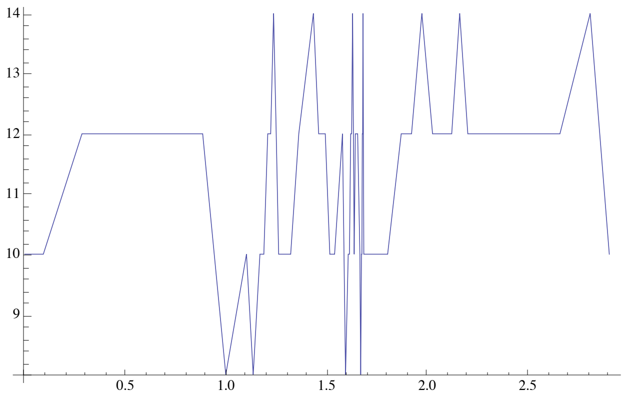
Use the monitor function to collect data as the integration proceeds.

```
In[34]:= data =
            Reap[
              NDSolve[system,
                Method → {"Extrapolation", Method -> "ExplicitModifiedMidpoint"},
                "MethodMonitor" :> OrderMonitor[T, NDSolve`Self]]
             ][[
             -1,
             1]];
```

Display how the order varies during the integration.

```
In[35]:= ListLinePlot[data]
```

Out[35]=



Method Comparison

Select the problem to solve.

```
In[67]:= system = GetNDSolveProblem["Arenstorf"];
```

A reference solution is computed with a method that switches between a pair of "Extrapolation" methods, depending on whether the problem appears to be stiff.

```
In[68]:= sol = NDSolve[system, Method → "StiffnessSwitching", WorkingPrecision → 32];

         refsol = First[FinalSolutions[system, sol]];
```

Define a list of methods to compare.

```
In[70]:= methods = {{"ExplicitRungeKutta", "StiffnessTest" → False}, {"Extrapolation",
             Method -> "ExplicitModifiedMidpoint", "StiffnessTest" → False}};
```

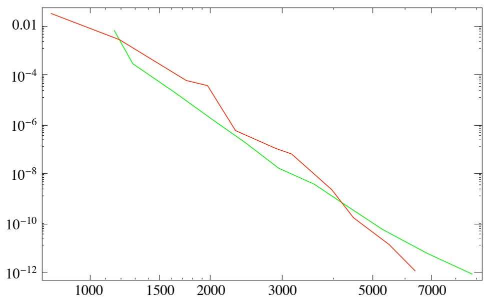
The data comparing accuracy and work is computed using `CompareMethods` for a range of tolerances.

```
In[71]:= data = Table[Map[Rest, CompareMethods[system, refsol,
              methods, AccuracyGoal → tol, PrecisionGoal → tol]], {tol, 4, 14}];
```

The work-error comparison data for the methods is displayed in the following logarithmic plot, where the global error is displayed on the vertical axis and the number of function evaluations on the horizontal axis. Eventually the higher order of the extrapolation methods means that they are more efficient. Note also that the increment formulation continues to give good results even at very stringent tolerances.

```
In[73]:= ListLogLogPlot[Transpose[data], Joined → True,
           Axes → False, Frame → True, PlotStyle → {{Green}, {Red}}]
```

Out[72]=



Stiff Systems

One of the simplest nonlinear equations describing a circuit is van der Pol's equation.

```
In[18]:= system = GetNDSolveProblem["VanderPol"];
         vars = system["DependentVariables"];
         time = system["TimeData"];
```

This solves the equations using `"Extrapolation"` with the `"ExplicitModifiedMidpoint"` base method with the default double-harmonic sequence 2, 4, 6, …. The stiffness detection device terminates the integration and an alternative method is suggested.

```
In[21]:= vdpsol = Flatten[vars /. NDSolve[system,
             Method → {"Extrapolation", Method → "ExplicitModifiedMidpoint"}]]

            NDSolve::ndstf :
              At T == 0.022920104414210326`, system appears to be stiff. Methods Automatic, BDF or
                     StiffnessSwitching may be more appropriate. ≫

Out[21]= {InterpolatingFunction[{{0., 0.0229201}}, <>][T],
          InterpolatingFunction[{{0., 0.0229201}}, <>][T]}
```

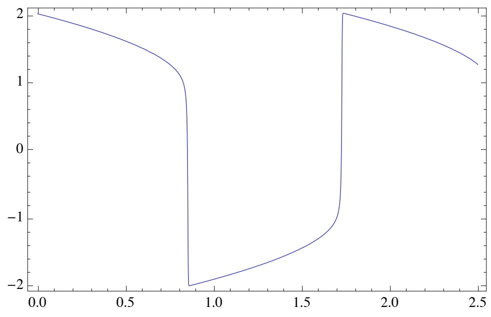
This solves the equations using “Extrapolation“ with the “LinearlyImplicitEuler“ base method with the default sub-harmonic sequence 2, 3, 4, ….

```
In[22]:= vdpsol = Flatten[vars /.
            NDSolve[system, Method → {“Extrapolation“, Method → “LinearlyImplicitEuler“}]]
Out[22]= {InterpolatingFunction[{{0., 2.5}}, <>][T], InterpolatingFunction[{{0., 2.5}}, <>][T]}
```

Notice that the Jacobian matrix is computed automatically (user-specifiable by using either numerical differences or symbolic derivatives) and appropriate linear algebra routines are selected and invoked at run time.

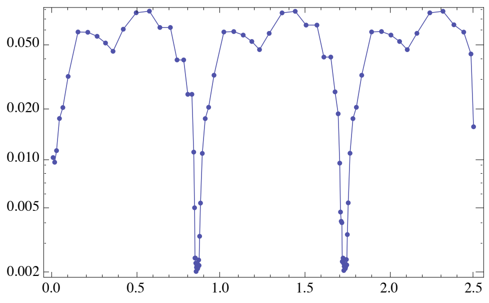
This plots the first solution component over time.

```
In[23]:= Plot[Evaluate[First[vdpsol]], Evaluate[time], Frame → True, Axes → False]
Out[23]=
```



This plots the step sizes taken in computing the solution.

```
In[24]:= StepDataPlot[vdpsol]
Out[24]=
```



High-Precision Comparison

Select the Lorenz equations.

```
In[25]:= system = GetNDSolveProblem[“Lorenz“];
```

This invokes a bigfloat, or software floating-point number, embedded explicit Runge-Kutta method of order 9(8) [V78].

```
In[26]:= Timing[
          erksol = NDSolve[system, Method → {"ExplicitRungeKutta", "DifferenceOrder" → 9},
             WorkingPrecision → 32];
         ]
Out[26]= {3.3105, Null}
```

This invokes the "Adams" method using a bigfloat version of LSODA. The maximum order of these methods is twelve.

```
In[27]:= Timing[
          adamssol = NDSolve[system, Method → "Adams", WorkingPrecision → 32];
         ]
Out[27]= {1.81172, Null}
```

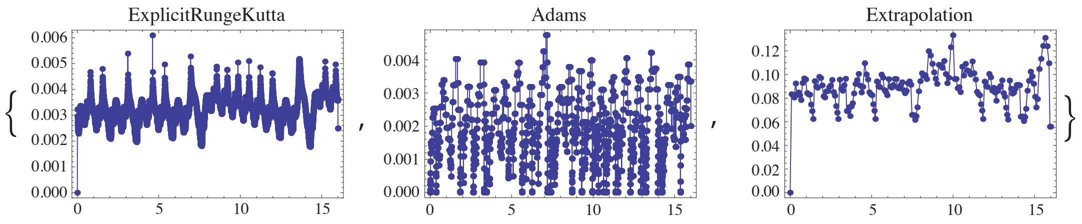
This invokes the "Extrapolation" method with "ExplicitModifiedMidpoint" as the base integration scheme.

```
In[28]:= Timing[
          extrapsol = NDSolve[system,
             Method → {"Extrapolation", Method -> "ExplicitModifiedMidpoint"},
             WorkingPrecision → 32];
         ]
Out[28]= {0.622906, Null}
```

Here are the step sizes taken by the various methods. The high order used in extrapolation means that much larger step sizes can be taken.

```
In[29]:= methods = {"ExplicitRungeKutta", "Adams", "Extrapolation"};
         solutions = {erksol, adamssol, extrapsol};
         MapThread[StepDataPlot[#2, PlotLabel → #1] &, {methods , solutions}]
```

Out[31]=



Mass Matrix - fem2ex

Consider the partial differential equation:

$$\frac{\partial u}{\partial t} = \exp(t)\,\frac{\partial^2 u}{\partial x^2},\; u(0,\,x) = \sin(x),\; u(t,\,0) = u(t,\,\pi) = 0. \qquad (1)$$

Given an integer n define $h = \pi/(n+1)$ and approximate at $x_k = k\,h$ with $k = 0, \ldots, n+1$ using the Galerkin discretization:

$$u(t,\ x_k) \approx \sum_{k=1}^{n} c_k(t)\,\phi_k(x) \tag{2}$$

where $\phi_k(x)$ is a piecewise linear function that is 1 at x_k and 0 at $x_j \neq x_k$.

The discretization (2) applied to (1) gives rise to a system of ordinary differential equations with constant mass matrix formulation as in (1). The ODE system is the fem2ex problem in [SR97] and is also found in the IMSL library.

The problem is set up to use sparse arrays for matrices which is not necessary for the small dimension being considered, but will scale well if the number of discretization points is increased. A vector-valued variable is used for the initial conditions. The system will be solved over the interval $[0, \pi]$.

```
In[35]:= n = 9;
         h = N[π / (n + 1)];
         amat = SparseArray[
            {{i_, i_} → 2 h / 3, {i_, j_} /; Abs[i - j] == 1 → h / 6}, {n + 2, n + 2}, 0.];
         rmat = SparseArray[{{i_, i_} → -2 / h, {i_, j_} /; Abs[i - j] == 1 → 1 / h},
            {n + 2, n + 2}, 0.];
         vars = {y[t]};
         eqs = {amat.y'[t] == rmat.(Exp[t] y[t])};
         ics = {y[0] == Table[Sin[k h], {k, 0, n + 1}]};
         system = {eqs, ics};
         time = {t, 0, π};
```

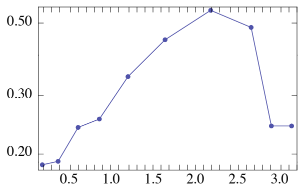
Solve the ODE system using using “Extrapolation“ with the “LinearlyImplicitEuler“ base method. The “SolveDelayed“ option is used to specify that the system is in mass matrix form.

```
In[44]:= sollim = NDSolve[system, vars, time,
            Method -> {“Extrapolation“, Method → “LinearlyImplicitEuler“},
            “SolveDelayed“ → “MassMatrix“, MaxStepFraction → 1];
```

This plot shows the relatively large step sizes that are taken by the method.

```
In[45]:= StepDataPlot[sollim]
```

Out[45]=



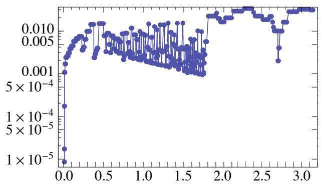
The default method for this type of problem is “IDA“ which is a general purpose differential algebraic equation solver [HT99]. Being much more general in scope, this method somewhat overkill for this example but serves for comparison purposes.

```
In[46]:= soldae = NDSolve[system, vars, time, MaxStepFraction → 1];
```

The following plot clearly shows that a much larger number of steps are taken by the DAE solver.

```
In[47]:= StepDataPlot[soldae]
```

Out[47]=



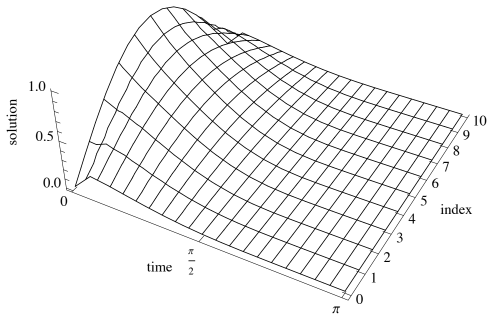
Define a function that can be used to plot the solutions on a grid.

```
In[48]:= PlotSolutionsOn3DGrid[{ndsol_}, opts___?OptionQ] :=
           Module[{if, m, n, sols, tvals, xvals},
            tvals = First[Head[ndsol]["Coordinates"]];
            sols = Transpose[ndsol /. t → tvals];
            m = Length[tvals];
            n = Length[sols];
            xvals = Range[0, n - 1];
            data =
             Table[{{Part[tvals, j], Part[xvals, i]}, Part[sols, i, j]}, {j, m}, {i, n}];
            data = Apply[Join, data];
            if = Interpolation[data];
            Plot3D[Evaluate[if[t, x]], Evaluate[{t, First[tvals], Last[tvals]}], Evaluate[
              {x, First[xvals], Last[xvals]}], PlotRange → All, Boxed → False, opts]
           ];
```

Display the solutions on a grid.

```
In[49]:= femsol = PlotSolutionsOn3DGrid[vars /. First[sollim],
           Ticks → {Table[i π, {i, 0, 1, 1 / 2}], Range[0, n + 1], Automatic},
           AxesLabel → {"time   ", "index",
             RawBoxes[RotationBox["solution\n", BoxRotation → Pi / 2]]},
           Mesh → {19, 9}, MaxRecursion → 0, PlotStyle → None]
```

Out[49]=



Fine-Tuning

"StepSizeSafetyFactors"

As with most methods, there is a balance between taking too small a step and trying to take too big a step that will be frequently rejected. The option `"StepSizeSafetyFactors"` -> $\{s_1, s_2\}$ constrains the choice of step size as follows. The step size chosen by the method for order p satisfies:

$$h_{n+1} = h_n\, s_1 \left(s_2 \frac{Tol}{\|err_n\|}\right)^{\frac{1}{p+1}}. \tag{1}$$

This includes both an order-dependent factor and an order-independent factor.

"StepSizeRatioBounds"

The option `"StepSizeRatioBounds"` -> $\{sr_{min}, sr_{max}\}$ specifies bounds on the next step size to take such that:

$$sr_{min} \le \left|\frac{h_{n+1}}{h_n}\right| \le sr_{max}.$$

"OrderSafetyFactors"

An important aspect in `"Extrapolation"` is the choice of order.

Each extrapolation step k has an associated work estimate $\mathcal{A}_k$.

The work estimate for explicit base methods is based on the number of function evaluations and the step sequence used.

The work estimate for linearly implicit base methods also includes an estimate of the cost of evaluating the Jacobian, the cost of an LU decomposition, and the cost of backsolving the linear equations.

Estimates for the work per unit step are formed from the work estimate $\mathcal{A}_k$ and the expected new step size to take for a method of order k (computed from (1)): $\mathcal{W}_k = \mathcal{A}_k / h_{n+1}^k$.

Comparing consecutive estimates, $\mathcal{W}_k$ allows a decision about when a different order method will be more efficient.

The option `"OrderSafetyFactors" -> {`f_1`, `f_2`}` specifies safety factors to be included in the comparison of estimates $\mathcal{W}_k$.

An order decrease is made when $\mathcal{W}_{k-1} < f_1 \mathcal{W}_k$.

An order increase is made when $\mathcal{W}_{k+1} < f_2 \mathcal{W}_k$.

There are some additional restrictions, such as when the maximal order increase per step is one (two for symmetric methods), and when an increase in order is prevented immediately after a rejected step.

For a nonstiff base method the default values are `{4 / 5, 9 / 10}` whereas for a stiff method they are `{7 / 10, 9 / 10}`.

Option Summary

option name	default value	
`"ExtrapolationSequence"`	`Automatic`	specify the sequence to use in extrapolation
`"MaxDifferenceOrder"`	`Automatic`	specify the maximum order to use
`Method`	`"ExplicitModifiedMidpoint"`	specify the base integration method to use
`"MinDifferenceOrder"`	`Automatic`	specify the minimum order to use
`"OrderSafetyFactors"`	`Automatic`	specify the safety factors to use in the estimates for adaptive order selection
`"StartingDifferenceOrder"`	`Automatic`	specify the initial order to use
`"StepSizeRatioBounds"`	`Automatic`	specify the bounds on a relative change in the new step size h_{n+1} from the current step size h_n as low $\leq h_{n+1} / h_n \leq$ high
`"StepSizeSafetyFactors"`	`Automatic`	specify the safety factors to incorporate into the error estimate used for adaptive step sizes
`"StiffnessTest"`	`Automatic`	specify whether to use the stiffness detection capability

Options of the method `"Extrapolation"`.

The default setting of `Automatic` for the option `"ExtrapolationSequence"` selects a sequence based on the stiffness and symmetry of the base method.

The default setting of `Automatic` for the option `"MaxDifferenceOrder"` bounds the maximum order by two times the decimal working precision.

The default setting of `Automatic` for the option `"MinDifferenceOrder"` selects the minimum number of two extrapolations starting from the order of the base method. This also depends on whether the base method is symmetric.

The default setting of `Automatic` for the option `"OrderSafetyFactors"` uses the values `{7 / 10, 9 / 10}` for a stiff base method and `{4 / 5, 9 / 10}` for a nonstiff base method.

The default setting of `Automatic` for the option `"StartingDifferenceOrder"` depends on the setting of `"MinDifferenceOrder"` p_{min}. It is set to $p_{min}+1$ or $p_{min}+2$ depending on whether the base method is symmetric.

The default setting of `Automatic` for the option `"StepSizeRatioBounds"` uses the values `{1 / 10, 4}` for a stiff base method and `{1 / 50, 4}` for a nonstiff base method.

The default setting of `Automatic` for the option `"StepSizeSafetyFactors"` uses the values `{9 / 10, 4 / 5}` for a stiff base method and `{9 / 10, 13 / 20}` for a nonstiff base method.

The default setting of `Automatic` for the option `"StiffnessTest"` indicates that the stiffness test is activated if a nonstiff base method is used.

option name	*default value*	
`"StabilityCheck"`	`True`	specify whether to carry out a stability check on consecutive implicit solutions (see e.g. (1))

Option of the method `"LinearlyImplicitEuler"`, `"LinearlyImplicitMidpoint"`, and `"LinearlyImplicitModifiedMidpoint"`.

"FixedStep" Method for NDSolve

Introduction

It is often useful to carry out a numerical integration using fixed step sizes.

For example, certain methods such as `"DoubleStep"` and `"Extrapolation"` carry out a sequence of fixed-step integrations before combining the solutions to obtain a more accurate method with an error estimate that allows adaptive step sizes to be taken.

The method `"FixedStep"` allows any one-step integration method to be invoked using fixed step sizes.

This loads a package with some example problems and a package with some utility functions.

```
In[3]:= Needs["DifferentialEquations`NDSolveProblems`"];
        Needs["DifferentialEquations`NDSolveUtilities`"];
```

Examples

Define an example problem.

```
In[5]:= system = GetNDSolveProblem["BrusselatorODE"]
```

Out[5]= NDSolveProblem$\left[\left\{\left\{(Y_1)'[T] = 1 - 4\,Y_1[T] + Y_1[T]^2\,Y_2[T],\ (Y_2)'[T] = 3\,Y_1[T] - Y_1[T]^2\,Y_2[T]\right\},\right.\right.$ $\left.\left.\left\{Y_1[0] = \frac{3}{2},\ Y_2[0] = 3\right\},\ \{Y_1[T],\ Y_2[T]\},\ \{T,\ 0,\ 20\},\ \{\},\ \{\},\ \{\}\right\}\right]$

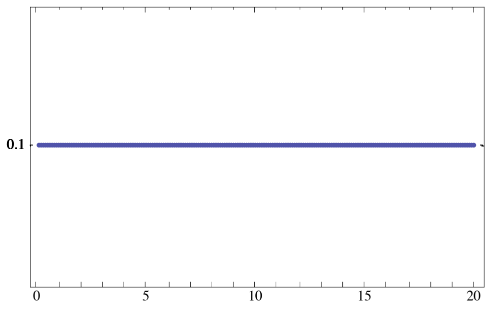
This integrates a differential system using the method "ExplicitEuler" with a fixed step size of 1/10.

```
In[6]:= NDSolve[{y''[t] == -y[t], y[0] == 1, y'[0] == 0}, y, {t, 0, 1},
         StartingStepSize → 1 / 10, Method → {"FixedStep", Method → "ExplicitEuler"}]

Out[6]= {{y → InterpolatingFunction[{{0., 1.}}, <>]}}
```

Actually the "ExplicitEuler" method has no adaptive step size control. Therefore, the integration is already carried out using fixed step sizes so the specification of "FixedStep" is unnecessary.

```
In[7]:= sol = NDSolve[system, StartingStepSize → 1 / 10, Method → "ExplicitEuler"];
        StepDataPlot[sol, PlotRange → {0, 0.2}]
```

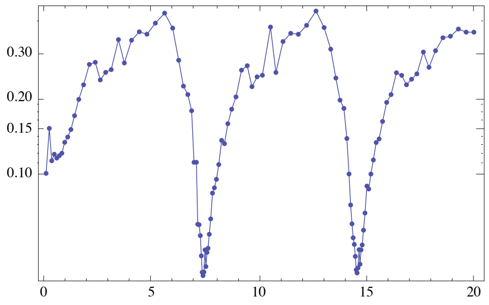
Out[8]=



Here are the step sizes taken by the method "ExplicitRungeKutta" for this problem.

```
In[9]:= sol = NDSolve[system, StartingStepSize → 1 / 10, Method → "ExplicitRungeKutta"];
        StepDataPlot[sol]
```

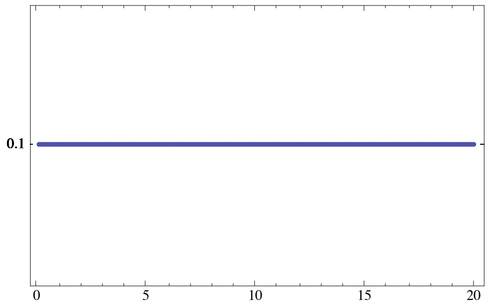
Out[10]=



This specifies that fixed step sizes should be used for the method "ExplicitRungeKutta".

```
In[11]:= sol = NDSolve[system, StartingStepSize → 1 / 10,
            Method → {"FixedStep", Method → "ExplicitRungeKutta"}];
         StepDataPlot[sol, PlotRange → {0, 0.2}]
```

Out[12]=

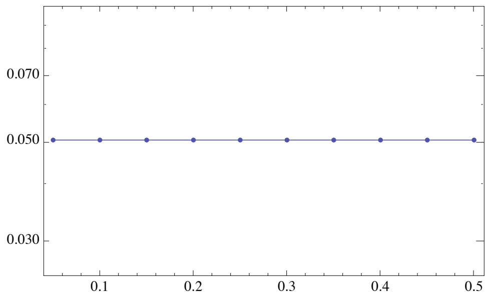


The option MaxStepFraction provides an absolute bound on the step size that depends on the integration interval.

Since the default value of MaxStepFraction is 1/10, the step size in this example is bounded by one-tenth of the integration interval, which leads to using a constant step size of 1/20.

```
In[13]:= time = {T, 0, 1 / 2};
         sol = NDSolve[system, time, StartingStepSize → 1 / 10,
            Method → {"FixedStep", Method → "ExplicitRungeKutta"}];
         StepDataPlot[sol, PlotRange → {0, 0.2}]
```

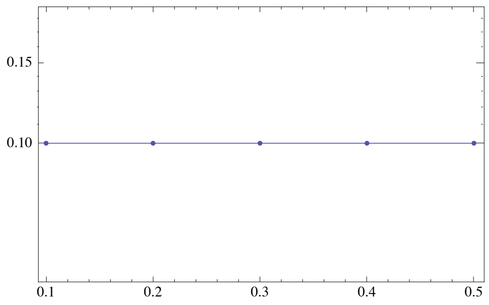
Out[15]=



By setting the value of `MaxStepFraction` to a different value, the dependence of the step size on the integration interval can be relaxed or removed entirely.

```
In[16]:= sol = NDSolve[system, time, StartingStepSize → 1 / 10, MaxStepFraction → Infinity,
            Method → {"FixedStep", Method → "ExplicitRungeKutta"}];
         StepDataPlot[sol, PlotRange → {0, 0.2}]
```

Out[17]=



Option Summary

option name	default value	
`Method`	`None`	specify the method to use with fixed step sizes

Option of the method "`FixedStep`".

"OrthogonalProjection" Method for NDSolve

Introduction

Consider the matrix differential equation:

$$y'(t) = f(t, y(t)),\ t > 0,$$

where the initial value $y_0 = y(0) \in \mathbb{R}^{m \times p}$ is given. Assume that $y_0{}^T y_0 = I$, that the solution has the property of preserving orthonormality, $y(t)^T y(t) = I$, and that it has full rank for all $t \geq 0$.

From a numerical perspective, a key issue is how to numerically integrate an orthogonal matrix differential system in such a way that the numerical solution remains orthogonal. There are several strategies that are possible. One approach, suggested in [DRV94], is to use an implicit Runge-Kutta method (such as the Gauss scheme). Some alternative strategies are described in [DV99] and [DL01].

The approach taken here is to use any reasonable numerical integration method and then postprocess using a projective procedure at the end of each integration step.

An important feature of this implementation is that the basic integration method can be any built-in numerical method, or even a user-defined procedure. In the following examples an explicit Runge-Kutta method is used for the basic time stepping. However, if greater accuracy is required an extrapolation method could easily be used, for example, by simply setting the appropriate `Method` option.

Projection Step

At the end of each numerical integration step you need to transform the approximate solution matrix of the differential system to obtain an orthogonal matrix. This can be carried out in several ways (see for example [DRV94] and [H97]):

- Newton or Schulz iteration
- QR decomposition
- Singular value decomposition

The Newton and Schulz methods are quadratically convergent, and the number of iterations may vary depending on the error tolerances used in the numerical integration. One or two iterations are usually sufficient for convergence to the orthonormal polar factor (see the following) in IEEE double-precision arithmetic.

QR decomposition is cheaper than singular value decomposition (roughly by a factor of two), but it does not give the closest possible projection.

Definition (Thin singular value decomposition [GVL96]): Given a matrix $A \in \mathbb{R}^{m \times p}$ with $m \geq p$ there exist two matrices $U \in \mathbb{R}^{m \times p}$ and $V \in \mathbb{R}^{p \times p}$ such that $U^T A V$ is the diagonal matrix of singular values of A, $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{p \times p}$, where $\sigma_1 \geq \cdots \geq \sigma_p \geq 0$. U has orthonormal columns and V is orthogonal.

Definition (Polar decomposition): Given a matrix A and its singular value decomposition $U \Sigma V^T$, the polar decomposition of A is given by the product of two matrices Z and P where $Z = U V^T$ and $P = V \Sigma V^T$. Z has orthonormal columns and P is symmetric positive semidefinite.

The orthonormal polar factor Z of A is the matrix that solves:

$$\min_{Z \in \mathbb{R}^{m \times p}} \left\{ \| A - Z \| : Z^T Z = I \right\}$$

for the 2 and Frobenius norms [H96].

Schulz Iteration

The approach chosen is based on the Schulz iteration, which works directly for $m \geq p$. In contrast, Newton iteration for $m > p$ needs to be preceded by QR decomposition.

Comparison with direct computation based on the singular value decomposition is also given.

The Schulz iteration is given by:

$$Y_{i+1} = Y_i + Y_i \left(I - Y_i^T \, Y_i \right) \big/ 2, \; Y_0 = A. \tag{1}$$

The Schulz iteration has an arithmetic operation count per iteration of $2\,m^2\,p + 2\,m\,p^2$ floating-point operations, but is rich in matrix multiplication [H97].

In a practical implementation, GEMM-based level 3 BLAS of LAPACK [LAPACK99] can be used in conjunction with architecture-specific optimizations via the Automatically Tuned Linear Algebra Software [ATLAS00]. Such considerations mean that the arithmetic operation count of the Schulz iteration is not necessarily an accurate reflection of the observed computational cost. A useful bound on the departure from orthonormality of A is in [H89]: $\| A^T\,A - I \|_F$. Comparison with the Schulz iteration gives the stopping criterion $\| A^T\,A - I \|_F < \tau$ for some tolerance τ.

Standard Formulation

Assume that an initial value y_n for the current solution of the ODE is given, together with a solution $y_{n+1} = y_n + \Delta\,y_n$ from a one-step numerical integration method. Assume that an absolute tolerance τ for controlling the Schulz iteration is also prescribed.

The following algorithm can be used for implementation.

Step 1. Set $Y_0 = y_{n+1}$ and $i = 0$.

Step 2. Compute $E = I - Y_i^T\,Y_i$.

Step 3. Compute $Y_{i+1} = Y_i + Y_i\,E/2$.

Step 4. If $\| E \|_F \leq \tau$ or $i = i_{max}$, then return Y_{i+1}.

Step 5. Set $i = i + 1$ and go to step 2.

Increment Formulation

NDSolve uses compensated summation to reduce the effect of rounding errors made by repeatedly adding the contribution of small quantities Δy_n to y_n at each integration step [H96]. Therefore, the increment Δy_n is returned by the base integrator.

An appropriate orthogonal correction ΔY_i for the projective iteration can be determined using the following algorithm.

Step 1. Set $\Delta Y_0 = 0$ and $i = 0$.

Step 2. Set $Y_i = \Delta Y_i + y_{n+1}$.

Step 3. Compute $E = I - Y_i^T Y_i$.

Step 4. Compute $\Delta Y_{i+1} = \Delta Y_i + Y_i E/2$.

Step 5. If $\| E \|_F \leq \tau$ or $i = i_{max}$, then return $\Delta Y_{i+1} + \Delta y_n$.

Step 6. Set $i = i + 1$ and go to step 2.

This modified algorithm is used in "OrthogonalProjection" and shows an advantage of using an iterative process over a direct process, since it is not obvious how an orthogonal correction can be derived for direct methods.

Examples

Orthogonal Error Measurement

A function to compute the Frobenius norm $\| A \|_F$ of a matrix A can be defined in terms of the Norm function as follows.

```
In[1]:= FrobeniusNorm[a_?MatrixQ] := Norm[a, Frobenius];
```

An upper bound on the departure from orthonormality of A can then be measured using this function [H97].

```
In[2]:= OrthogonalError[a_?MatrixQ] :=
          FrobeniusNorm[Transpose[a].a - IdentityMatrix[Last[Dimensions[a]]]];
```

This defines the utility function for visualizing the orthogonal error during a numerical integration.

```
In[4]:= (* Utility function for extracting a list of values of the
         independent variable at which the integration method has sampled *)

        TimeData[{v_?VectorQ, ___?VectorQ}] := TimeData[v];

        TimeData[{if : (InterpolatingFunction[__])[_], ___}] :=
          Part[if, 0, 3, 1];
```

```
In[6]:= (* Utility function for plotting the
         orthogonal error in a numerical integration *)

        OrthogonalErrorPlot[sol_] :=
          Module[{errdata, samples, soldata},
           (* Form a list of times at which the method is invoked *)
           samples = TimeData[sol];
           (* Form a list of solutions at the integration times *)
           soldata = Map[(sol /. t → #) &, samples];
           (* Form a list of the orthogonal errors *)
           errdata = Map[OrthogonalError, soldata];
           ListLinePlot[Transpose[{samples, errdata}],
            Frame → True, PlotLabel → "Orthogonal error ||Y^T Y - I||_F vs time"
           ]
          ];
```

Square Systems

This example concerns the solution of a matrix differential system on the orthogonal group $O_3(\mathbb{R})$ (see [Z98]).

The matrix differential system is given by

$$\begin{aligned} Y' &= F(Y)\,Y \\ &= \left(A + \left(I - Y\,Y^T\right)\right) Y \end{aligned}$$

with

$$A = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & 1 \\ -1 & -1 & 0 \end{pmatrix}$$

and

$$Y_0 = I_3.$$

The solution evolves as:

$$Y(t) = \exp[t\,A].$$

The eigenvalues of $Y(t)$ are $\lambda_1 = 1$, $\lambda_2 = \exp\left(t\, i \sqrt{3}\right)$, $\lambda_3 = \exp\left(-t\, i \sqrt{3}\right)$. Thus as t approaches $\pi \Big/ \sqrt{3}$, two of the eigenvalues of $Y(t)$ approach -1. The numerical integration is carried out on the interval $[0, 2]$.

```
In[7]:= n = 3;

        A = ( 0  -1  1
              1   0  1
             -1  -1  0 );

        Y = Table[y[i, j][t], {i, n}, {j, n}];

        F = A + ( IdentityMatrix[n] - Transpose[Y].Y);
```

```
In[8]:= (* Vector differential system *)

        system = Thread[Flatten[D[Y, t]] == Flatten[F.Y]];

        (* Vector initial conditions *)

        ics = Thread[Flatten[(Y /. t → 0)] == Flatten[IdentityMatrix[Length[Y]]]];

        eqs = {system, ics};

        vars = Flatten[Y];

        time = {t, 0, 2};
```

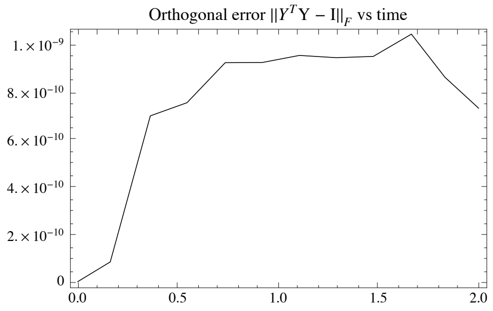
This computes the solution using an explicit Runge-Kutta method. The appropriate initial step size and method order are selected automatically, and the step size may vary throughout the integration interval, which is chosen in order to satisfy local relative and absolute error tolerances. Alternatively, the order of the method could be specified by using a `Method` option.

```
In[16]:= solerk = NDSolve[eqs, vars, time, Method → "ExplicitRungeKutta"];
```

This computes the orthogonal error, or absolute deviation from the orthogonal manifold, as the integration progresses. The error is of the order of the local accuracy of the numerical method.

```
In[17]:= solerk = Y /. First[solerk];

         OrthogonalErrorPlot[solerk]
```

Out[18]=



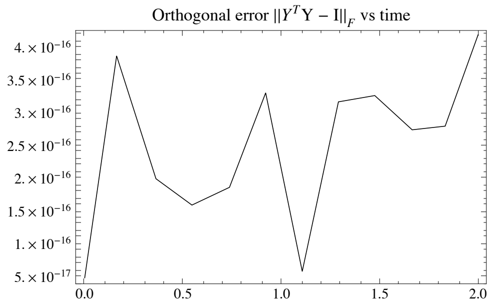
This computes the solution using an orthogonal projection method with an explicit Runge-Kutta method used for the basic integration step. The initial step size and method order are the same as earlier, but the step size sequence in the integration may differ.

```
In[19]:= solop = NDSolve[eqs, vars, time, Method → {"OrthogonalProjection",
              Method → "ExplicitRungeKutta", Dimensions → Dimensions[Y]}];
```

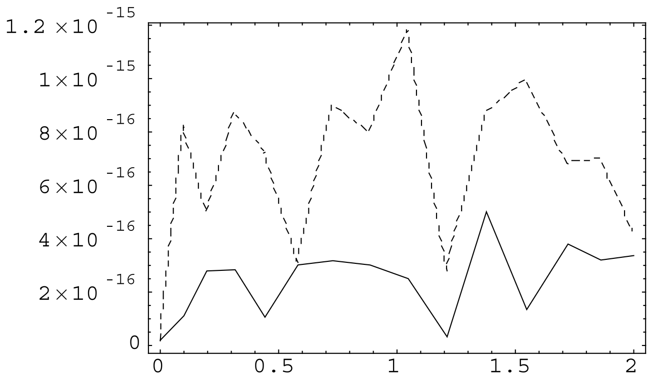
Using the orthogonal projection method, the orthogonal error is reduced to approximately the level of roundoff in IEEE double-precision arithmetic.

```
In[20]:= solop = Y /. First[solop];

         OrthogonalErrorPlot[solop]
```

Out[21]=



The Schulz iteration, using the incremental formulation, generally yields smaller errors than the direct singular value decomposition.



Rectangular Systems

In the following example it is shown how the implementation of the orthogonal projection method also works for rectangular matrix differential systems. Formally stated, the interest is in solving ordinary differential equations on the *Stiefel manifold*, the set of $n \times p$ orthogonal matrices with $p < n$.

Definition The *Stiefel manifold* of $n \times p$ orthogonal matrices is the set $V_{n,p}(\mathbb{R}) = \left\{ Y \in \mathbb{R}^{n \times p} \mid Y^T Y = I_p \right\}$, $1 \leq p < n$, where I_p is the $p \times p$ identity matrix.

Solutions that evolve on the Stiefel manifold find numerous applications such as eigenvalue problems in numerical linear algebra, computation of Lyapunov exponents for dynamical systems and signal processing.

Consider an example adapted from [DL01]:

$$q'(t) = A\, q(t),\ t > 0,\ q(0) = q_0$$

where $q_0 = 1 \Big/ \sqrt{n}\ [1, \ldots, 1]^T$, $A = \operatorname{diag}[a_1, \ldots, a_n] \in \mathbb{R}^{n \times n}$, with $a_i = (-1)^i \alpha,\ i = 1, \ldots, n$ and $\alpha > 0$.

The exact solution is given by:

$$q(t) = \frac{1}{\sqrt{n}} \begin{pmatrix} \exp(a_1 t) \\ \vdots \\ \exp(a_n t) \end{pmatrix}.$$

Normalizing $q(t)$ as:

$$Y(t) = \frac{q(t)}{\| q(t) \|} \in \mathbb{R}^{n \times 1}$$

it follows that $Y(t)$ satisfies the following weak skew-symmetric system on $V_{n,1}(\mathbb{R})$:

$$\begin{aligned} Y' &= F(Y)\, Y \\ &= \left(I_n - Y\, Y^T\right) A\, Y \end{aligned}$$

In the following example, the system is solved on the interval $[0, 5]$ with $\alpha = 9/10$ and dimension $n = 2$.

```
In[22]:= p = 1;

         n = 2;

         α = 9/10;

         ics = 1/√n Table[1, {n}];

         avec = Table[(-1)^i α, {i, n}];

         A = DiagonalMatrix[avec];

         Y = Table[y[i, 1][t], {i, n}, {j, p}];

         F = (IdentityMatrix[Length[Y]] - Y.Transpose[Y]).A;

         system = Thread[Flatten[D[Y, t]] == Flatten[F.Y]];

         ics = Thread[Flatten[(Y /. t → 0)] == ics];

         eqs = {system, ics};

         vars = Flatten[Y];

         tfinal = 5.;

         time = {t, 0, tfinal};
```

This computes the exact solution which can be evaluated throughout the integration interval.

```
In[36]:= solexact = Transpose[{(#/Norm[#, 2])}] & @ (Exp[avec t]/√n);
```

This computes the solution using an explicit Runge-Kutta method.

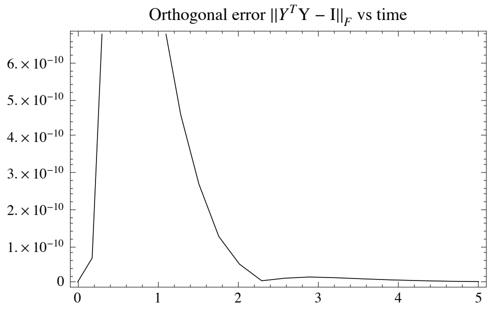
```
In[37]:= solerk = NDSolve[eqs, vars, time, Method → "ExplicitRungeKutta"];

         solerk = Y /. First[solerk];
```

This computes the componentwise absolute global error at the end of the integration interval.

```
In[39]:= (solexact - solerk) /. t → tfinal

Out[39]= {{-2.03407×10^-11}, {2.96319×10^-13}}
```

This computes the orthogonal error—a measure of the deviation from the Stiefel manifold.

```
In[40]:= OrthogonalErrorPlot[solerk]
```

Out[40]=


This computes the solution using an orthogonal projection method with an explicit Runge-Kutta method as the basic numerical integration scheme.

```
In[41]:= solop = NDSolve[eqs, vars, time, Method → {"OrthogonalProjection",
              Method → "ExplicitRungeKutta", Dimensions → Dimensions[Y]}];

         solop = Y /. First[solop];
```

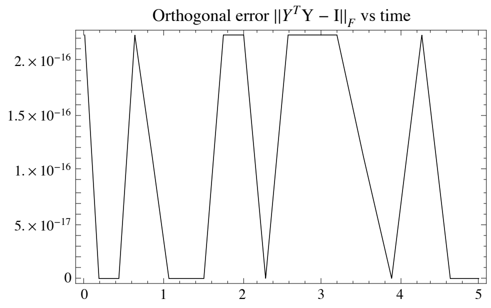
The componentwise absolute global error at the end of the integration interval is roughly the same as before since the absolute and relative tolerances used in the numerical integration are the same.

```
In[43]:= (solexact - solop) /. t → tfinal
```

Out[43]= $\{\{-2.03407\times 10^{-11}\}, \{2.55351\times 10^{-15}\}\}$

Using the orthogonal projection method, however, the deviation from the Stiefel manifold is reduced to the level of roundoff.

```
In[44]:= OrthogonalErrorPlot[solop]
```

Out[44]=


Implementation

The implementation of the method “`OrthogonalProjection`“ has three basic components:

- Initialization. Set up the base method to use in the integration, determining any method coefficients and setting up any workspaces that should be used. This is done once, before any actual integration is carried out, and the resulting `MethodData` object is validated so that it does not need to be checked at each integration step. At this stage the system dimensions and initial conditions are checked for consistency.
- Invoke the base numerical integration method at each step.
- Perform an orthogonal projection. This performs various tests such as checking that the basic integration proceeded correctly and that the Schulz iteration converges.

Options can be used to modify the stopping criteria for the Schulz iteration. One option provided by the code is “`IterationSafetyFactor`“ which allows control over the tolerance τ of the iteration. The factor is combined with a Unit in the Last Place, determined according to the working precision used in the integration ($\text{ULP} \approx 2.22045 \times 10^{-16}$ for IEEE double precision).

The Frobenius norm used for the stopping criterion can be computed efficiently using the LAPACK LANGE functions [LAPACK99].

The option `MaxIterations` controls the maximum number of iterations that should be carried out.

Option Summary

option name	*default value*	
`Dimensions`	`{}`	specify the dimensions of the matrix differential system
`"IterationSafetyFactor"`	$\frac{1}{10}$	specify the safety factor to use in the termination criterion for the Schulz iteration (1)
`MaxIterations`	`Automatic`	specify the maximum number of iterations to use in the Schulz iteration (1)
`Method`	`"StiffnessSwitching"`	specify the method to use for the numerical integration

Options of the method “`OrthogonalProjection`“.

"Projection" Method for NDSolve

Introduction

When a differential system has a certain structure, it is advantageous if a numerical integration method preserves the structure. In certain situations it is useful to solve differential equations in which solutions are constrained. Projection methods work by taking a time step with a numerical integration method and then projecting the approximate solution onto the manifold on which the true solution evolves.

`NDSolve` includes a differential algebraic solver which may be appropriate and is described in more detail within "Numerical Solution of Differential-Algebraic Equations".

Sometimes the form of the equations may not be reduced to the form required by a DAE solver. Furthermore so-called index reduction techniques can destroy certain structural properties, such as symplecticity, that the differential system may possess (see [HW96] and [HLW02]). An example that illustrates this can be found in the documentation for DAEs.

In such cases it is often possible to solve a differential system and then use a projective procedure to ensure that the constraints are conserved. This is the idea behind the method “`Projection`“.

If the differential system is ρ-reversible then a *symmetric* projection process can be advantageous (see [H00]). Symmetric projection is generally more costly than projection and has not yet been implemented in `NDSolve`.

Invariants

Consider a differential equation

$$\dot{y} = f(y),\ y(t_0) = y_0, \qquad (1)$$

where y may be a vector or a matrix.

Definition: A nonconstant function $I(y)$ is called an *invariant* of (1) if $I'(y)\,f(y) = 0$ for all y.

This implies that every solution $y(t)$ of (1) satisfies $I(y(t)) = I\,(y_0) = \text{Constant}$.

Synonymous with invariant, the terms *first integral*, *conserved quantity*, or *constant of the motion* are also common.

Manifolds

Given an $(n-m)$-dimensional submanifold of $\mathbb{R}^n$ with $g : \mathbb{R}^n \mapsto \mathbb{R}^m$:

$$\mathcal{M} = \{y; g(y) = 0\}. \tag{1}$$

Given a differential equation (1) then $y_0 \in \mathcal{M}$ implies $y(t) \in \mathcal{M}$ for all t. This is a weaker assumption than invariance and $g(y)$ is called a weak invariant (see [HLW02]).

Projection Algorithm

Let $\tilde{y}_{n+1}$ denote the solution from a one-step numerical integrator. Considering a constrained minimization problem leads to the following system (see [AP91], [HW96] and [HLW02]):

$$\begin{array}{rcl} y_{n+1} & = & \tilde{y}_{n+1} + g'(y_{n+1})^T \lambda \\ 0 & = & g(y_{n+1}). \end{array} \tag{1}$$

To save work $g(y_{n+1})$ is approximated as $g\left(\tilde{y}_{n+1}\right)$. Substituting the first relation into the second relation in (1) leads to the following simplified Newton scheme for λ:

$$\begin{array}{rcl} \Delta\lambda_i & = & -\left(g'\left(\tilde{y}_{n+1}\right) g'\left(\tilde{y}_{n+1}\right)^T\right)^{-1} g\left(\tilde{y}_{n+1} + g'\left(\tilde{y}_{n+1}\right)^T \lambda_i\right), \\ \lambda_{i+1} & = & \lambda_i + \Delta\lambda_i \end{array} \tag{2}$$

with $\lambda_0 = 0$.

The first increment $\Delta\lambda_0$ is of size $O\left(h_n^{p+1}\right)$ so that (2) usually converges quickly.

The added expense of using a higher-order integration method can be offset by fewer Newton iterations in the projective step.

For the termination criterion in the method `"Projection"`, the option `"IterationSafetyFactor"` is combined with one Unit in the Last Place in the working precision used by `NDSolve`.

Examples

Load some utility packages.

```
In[3]:= Needs["DifferentialEquations`NDSolveProblems`"];
        Needs["DifferentialEquations`NDSolveUtilities`"];
```

Linear Invariants

Define a stiff system modeling a chemical reaction.

```
In[5]:= system = GetNDSolveProblem["Robertson"];
        vars = system["DependentVariables"];
```

This system has a linear invariant.

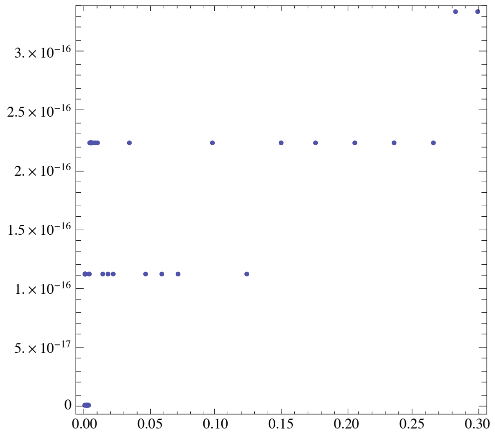
```
In[7]:= invariant = system["Invariants"]
```

Out[7]= $\{Y_1[T] + Y_2[T] + Y_3[T]\}$

Linear invariants are generally conserved by numerical integrators (see [S86]), including the default `NDSolve` method, as can be observed in a plot of the error in the invariant.

```
In[8]:= sol = NDSolve[system];

        InvariantErrorPlot[invariant, vars, T, sol]
```

Out[9]=



Therefore in this example there is no need to use the method `"Projection"`.

Certain numerical methods preserve quadratic invariants exactly (see for example [C87]). The implicit midpoint rule, or one-stage Gauss implicit Runge-Kutta method, is one such method.

Harmonic Oscillator

Define the harmonic oscillator.

```
In[10]:= system = GetNDSolveProblem["HarmonicOscillator"];
         vars = system["DependentVariables"];
```

The harmonic oscillator has the following invariant.

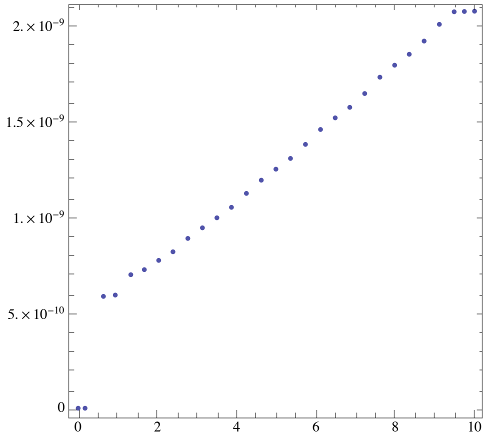
```
In[12]:= invariant = system["Invariants"]
```

Out[12]= $\left\{\frac{1}{2}\left(Y_1[T]^2 + Y_2[T]^2\right)\right\}$

Solve the system using the method "`ExplicitRungeKutta`". The error in the invariant grows roughly linearly, which is typical behavior for a dissipative method applied to a Hamiltonian system.

```
In[13]:= erksol = NDSolve[system, Method → "ExplicitRungeKutta"];

InvariantErrorPlot[invariant, vars, T, erksol]
```

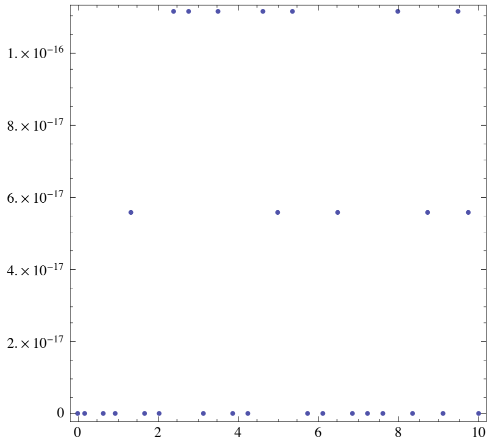
Out[14]=



This also solves the system using the method "`ExplicitRungeKutta`" but it projects the solution at the end of each step. A plot of the error in the invariant shows that it is conserved up to roundoff.

```
In[15]:= projerksol = NDSolve[system, Method →
             {"Projection", Method → "ExplicitRungeKutta", "Invariants" → invariant}];

InvariantErrorPlot[invariant, vars, T, projerksol]
```

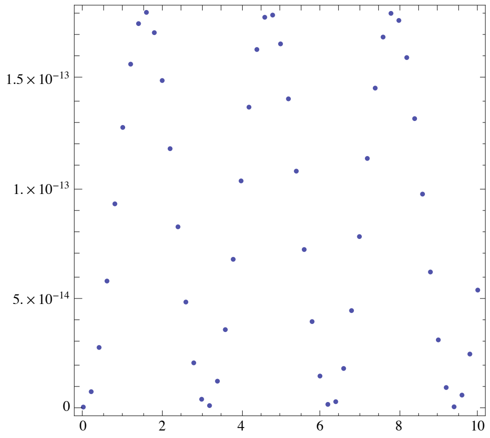
Out[16]=



Since the system is Hamiltonian (the invariant is the Hamiltonian), a symplectic integrator performs well on this problem, giving a small bounded error.

```
In[17]:= projerksol = NDSolve[system,
            Method → {"SymplecticPartitionedRungeKutta", "DifferenceOrder" → 8,
              "PositionVariables" → {Y1[T]}}, StartingStepSize → 1 / 5];

         InvariantErrorPlot[invariant, vars, T, projerksol]
```

Out[18]=



Perturbed Kepler Problem

This loads a Hamiltonian system known as the perturbed Kepler problem, sets the integration interval and the step size to take, as well as defining the position variables in the Hamiltonian formalism.

```
In[19]:= system = GetNDSolveProblem["PerturbedKepler"];
         time = system["TimeData"];
         step = 3 / 100;
         pvars = Take[system["DependentVariables"], 2]

Out[22]= {Y1[T], Y2[T]}
```

The system has two invariants, which are defined as H and L.

```
In[23]:= {H, L} = system["Invariants"]
```

Out[23]= $$\left\{-\frac{1}{400\left(Y_1[T]^2+Y_2[T]^2\right)^{3/2}}-\frac{1}{\sqrt{Y_1[T]^2+Y_2[T]^2}}+\frac{1}{2}\left(Y_3[T]^2+Y_4[T]^2\right),\ -Y_2[T]\,Y_3[T]+Y_1[T]\,Y_4[T]\right\}$$

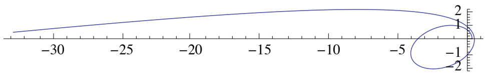
An experiment now illustrates the importance of using all the available invariants in the projective process (see [HLW02]). Consider the solutions obtained using:

- The method "ExplicitEuler"
- The method "Projection" with "ExplicitEuler", projecting onto the invariant L

- The method "Projection" with "ExplicitEuler", projecting onto the invariant H
- The method "Projection" with "ExplicitEuler", projecting onto both the invariants H and L

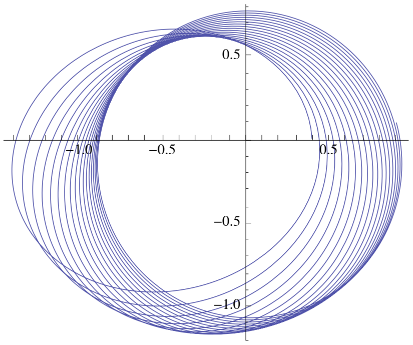
```
In[24]:= sol = NDSolve[system, Method → "ExplicitEuler", StartingStepSize → step];

         ParametricPlot[Evaluate[pvars /. First[sol]], Evaluate[time]]
```

Out[25]=



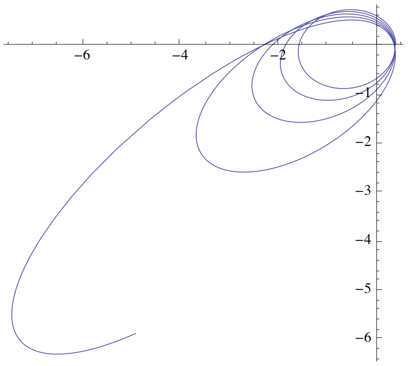
```
In[26]:= sol = NDSolve[system, Method → {"Projection", Method -> "ExplicitEuler",
              "Invariants" → {H}}, StartingStepSize → step];

         ParametricPlot[Evaluate[pvars /. First[sol]], Evaluate[time]]
```

Out[27]=



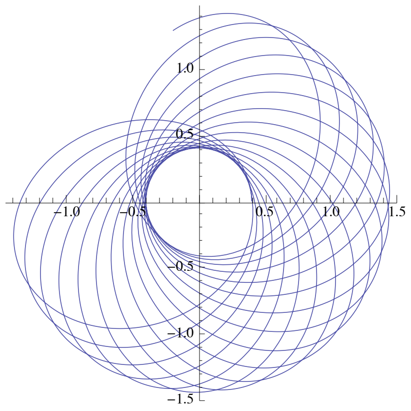
```
In[28]:= sol = NDSolve[system, Method → {"Projection", Method -> "ExplicitEuler",
              "Invariants" → {L}}, StartingStepSize → step];

         ParametricPlot[Evaluate[pvars /. First[sol]], Evaluate[time]]
```

Out[29]=



```
In[30]:= sol = NDSolve[system, Method → {"Projection", Method -> "ExplicitEuler",
              "Invariants" → {H, L}}, StartingStepSize → step];

         ParametricPlot[Evaluate[pvars /. First[sol]], Evaluate[time]]
```

Out[31]=



It can be observed that only the solution with projection onto both invariants gives the correct qualitative behavior—for comparison, results using an efficient symplectic solver can be found in "SymplecticPartitionedRungeKutta Method for NDSolve".

Lotka Volterra

An example of constraint projection for the Lotka-Volterra system is given within "Numerical Methods for Solving the Lotka-Volterra Equations".

Euler's Equations

An example of constraint projection for Euler's equations is given within "Rigid Body Solvers".

Option Summary

option name	*default value*	
`"Invariants"`	`None`	specify the invariants of the differential system
`"IterationSafetyFactor"`	$\frac{1}{10}$	specify the safety factor to use in the iterative solution of the invariants
`MaxIterations`	`Automatic`	specify the maximum number of iterations to use in the iterative solution of the invariants
`Method`	`"StiffnessSwitching"`	specify the method to use for integrating the differential system numerically

Options of the method "`Projection`".

"StiffnessSwitching" Method for NDSolve

Introduction

The basic idea behind the “StiffnessSwitching“ method is to provide an automatic means of switching between a nonstiff and a stiff solver.

The “StiffnessTest“ and “NonstiffTest“ options (described within "Stiffness Detection in NDSolve") provides a useful means of detecting when a problem appears to be stiff.

The “StiffnessSwitching“ method traps any failure code generated by “StiffnessTest“ and switches to an alternative solver. The “StiffnessSwitching“ method also uses the method specified in the “NonstiffTest“ option to switch back from a stiff to a nonstiff method.

“Extrapolation“ provides a powerful technique for computing highly accurate solutions using dynamic order and step size selection (see "Extrapolation Method for NDSolve" for more details) and is therefore used as the default choice in “StiffnessSwitching“.

Examples

This loads some useful packages.

```
In[3]:= Needs[“DifferentialEquations`NDSolveProblems`“];
        Needs[“DifferentialEquations`NDSolveUtilities`“];
```

This selects a stiff problem and specifies a longer integration time interval than the default specified by NDSolveProblem.

```
In[5]:= system = GetNDSolveProblem[“VanderPol“];
        time = {T, 0, 10};
```

The default “Extrapolation“ base method is not appropriate for stiff problems and gives up quite quickly.

```
In[7]:= NDSolve[system, time, Method → “Extrapolation“]

    NDSolve::ndstf :
      At T == 0.022920104414210326`, system appears to be stiff. Methods Automatic, BDF or
            StiffnessSwitching may be more appropriate. ≫

Out[7]= {{Y1[T] → InterpolatingFunction[{{0., 0.0229201}}, <>][T],
          Y2[T] → InterpolatingFunction[{{0., 0.0229201}}, <>][T]}}
```

Instead of giving up, the “StiffnessSwitching“ method continues the integration with a stiff solver.

```
In[8]:= NDSolve[system, time, Method → “StiffnessSwitching“]

Out[8]= {{Y1[T] → InterpolatingFunction[{{0., 10.}}, <>][T],
          Y2[T] → InterpolatingFunction[{{0., 10.}}, <>][T]}}
```

The `“StiffnessSwitching”` method uses a pair of extrapolation methods as the default. The nonstiff solver uses the `“ExplicitModifiedMidpoint”` base method, and the stiff solver uses the `“LinearlyImplicitEuler”` base method.

For small values of the `AccuracyGoal` and `PrecisionGoal` tolerances, it is sometimes preferable to use an explicit Runge-Kutta method for the nonstiff solver.

The `“ExplicitRungeKutta”` method eventually gives up when the problem is considered to be stiff.

```
In[9]:= NDSolve[system, time, Method → “ExplicitRungeKutta”,
         AccuracyGoal → 5, PrecisionGoal → 4]

           NDSolve::ndstf :
             At T == 0.028229404169279455`, system appears to be stiff. Methods Automatic, BDF or
                     StiffnessSwitching may be more appropriate. ≫

Out[9]= {{Y1[T] → InterpolatingFunction[{{0., 0.0282294}}, <>][T],
          Y2[T] → InterpolatingFunction[{{0., 0.0282294}}, <>][T]}}
```

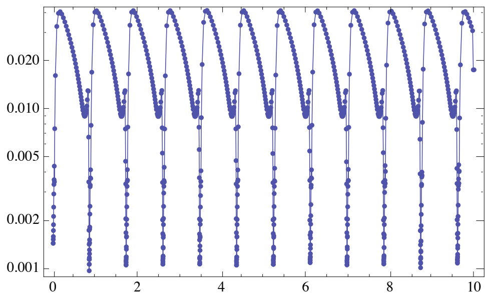
This sets the `“ExplicitRungeKutta”` method as a submethod of `“StiffnessSwitching”`.

```
In[10]:= sol = NDSolve[system, time,
            Method → {StiffnessSwitching, Method → {ExplicitRungeKutta, Automatic}},
            AccuracyGoal → 5, PrecisionGoal → 4]
Out[10]= {{Y1[T] → InterpolatingFunction[{{0., 10.}}, <>][T],
           Y2[T] → InterpolatingFunction[{{0., 10.}}, <>][T]}}
```

A switch to the stiff solver occurs at $T \approx 0.0282294$, and a plot of the step sizes used shows that the stiff solver takes much larger steps.

```
In[11]:= StepDataPlot[sol]
```

Out[11]=



Option Summary

option name	*default value*	
`Method`	`{Automatic, Automatic}`	specify the methods to use for the nonstiff and stiff solvers respectively
`“NonstiffTest“`	`Automatic`	specify the method to use for deciding whther to switch to a nonstiff solver

Options of the method `“StiffnessSwitching“`.

Extensions

NDSolve Method Plug-in Framework

Introduction

The control mechanisms set up for `NDSolve` enable you to define your own numerical integration algorithms and use them as specifications for the `Method` option of `NDSolve`.

`NDSolve` accesses its numerical algorithms and the information it needs from them in an object-oriented manner. At each step of a numerical integration, `NDSolve` keeps the method in a form so that it can keep private data as needed.

AlgorithmIdentifier [*data*]	an algorithm object that contains any *data* that a particular numerical ODE integration algorithm may need to use; the data is effectively private to the algorithm; *AlgorithmIdentifier* should be a *Mathematica* symbol, and the algorithm is accessed from `NDSolve` by using the option `Method` -> *AlgorithmIdentifier*

The structure for method data used in `NDSolve`.

`NDSolve` does not access the data associated with an algorithm directly, so you can keep the information needed in any form that is convenient or efficient to use. The algorithm and information that might be saved in its private data are accessed only through method functions of the algorithm object.

AlgorithmObject["Step"[*rhs*,*t*,*h*,*y*,*yp*]]	attempt to take a single time step of size *h* from time *t* to time *t* + *h* using the numerical algorithm, where *y* and *yp* are the approximate solution vector and its time derivative, respectively, at time *t*; the function should generally return a list {*newh*, Δ *y*} where *newh* is the best size for the next step determined by the algorithm and Δ *y* is the *increment* such that the approximate solution at time *t* + *h* is given by *y* + Δ *y*; if the time step is too large, the function should only return the value {*hnew*} where *hnew* should be small enough for an acceptable step (see later for complete descriptions of possible return values)
AlgorithmObject["DifferenceOrder"]	return the current asymptotic difference order of the algorithm
AlgorithmObject["StepMode"]	return the step mode for the algorithm object where the step mode should either be `Automatic` or `Fixed`; `Automatic` means that the algorithm has a means to estimate error and determines an appropriate size *newh* for the next time step; `Fixed` means that the algorithm will be called from a time step controller and is not expected to do any error estimation

Required method functions for algorithms used from `NDSolve`.

These method functions must be defined for the algorithm to work with `NDSolve`. The `"Step"` method function should always return a list, but the length of the list depends on whether the step was successful or not. Also, some methods may need to compute the function value *rhs*[*t* + *h*, *y* + Δ *y*] at the step end, so to avoid recomputation, you can add that to the list.

“Step“[*rhs*, *t*, *h*, *y*, *yp*] *method output*	*interpretation*
{*newh*,Δ *y*}	successful step with computed solution increment Δ *y* and recommended next step *newh*
{*newh*,Δ *y*,*yph*}	successful step with computed solution increment Δ *y* and recommended next step *newh* and time derivatives computed at the step endpoint, *yph* = *rhs*[*t* + *h*, *y* + Δ *y*]
{*newh*,Δ *y*,*yph*,*newobj*}	successful step with computed solution increment Δ *y* and recommended next step *newh* and time derivatives computed at the step endpoint, *yph* = *rhs*[*t* + *h*, *y* + Δ *y*]; any changes in the object data are returned in the new instance of the method object, *newobj*
{*newh*,Δ *y*,None,*newobj*}	successful step with computed solution increment Δ *y* and recommended next step *newh*; any changes in the object data are returned in the new instance of the method object, *newobj*
{*newh*}	rejected step with recommended next step *newh* such that |*newh*| < |*h*|
{*newh*,$Failed,None,*newobj*}	rejected step with recommended next step *newh* such that |*newh*| < |*h*|; any changes in the object data are returned in the new instance of the method object, *newobj*

Interpretation of “Step“ method output.

Classical Runge-Kutta

Here is an example of how to set up and access a simple numerical algorithm.

This defines a method function to take a single step toward integrating an ODE using the classical fourth-order Runge-Kutta method. Since the method is so simple, it is not necessary to save any private data.

```
In[1]:= CRK4[]["Step"[rhs_, t_, h_, y_, yp_]] := Module[{k0, k1, k2, k3},
          k0 = h yp;
          k1 = h rhs[t + h / 2, y + k0 / 2];
          k2 = h rhs[t + h / 2, y + k1 / 2];
          k3 = h rhs[t + h, y + k2];
          {h, (k0 + 2 k1 + 2 k2 + k3) / 6}]
```

This defines a method function so that NDSolve can obtain the proper difference order to use for the method. The ___ template is used because the difference order for the method is always 4.

```
In[2]:= CRK4[___]["DifferenceOrder"] := 4
```

This defines a method function for the step mode so that `NDSolve` will know how to control time steps. This algorithm method does not have any step control, so you define the step mode to be `Fixed`.

```
In[3]:= CRK4[___]["StepMode"] := Fixed
```

This integrates the simple harmonic oscillator equation with fixed step size.

```
In[4]:= fixed =
         NDSolve[{x''[t] + x[t] == 0, x[0] == 1, x'[0] == 0}, x, {t, 0, 2 π}, Method → CRK4]
Out[4]= {{x → InterpolatingFunction[{{0., 6.28319}}, <>]}}
```

Generally using a fixed step size is less efficient than allowing the step size to vary with the local difficulty of the integration. Modern explicit Runge-Kutta methods (accessed in `NDSolve` with `Method -> "ExplicitRungeKutta"`) have a so-called embedded error estimator that makes it possible to very efficiently determine appropriate step sizes. An alternative is to use built-in step controller methods that use extrapolation. The method `"DoubleStep"` uses an extrapolation based on integrating a time step with a single step of size h and two steps of size $h/2$. The method `"Extrapolation"` does a more sophisticated extrapolation and modifies the degree of extrapolation automatically as the integration is performed, but is generally used with base methods of difference orders 1 and 2.

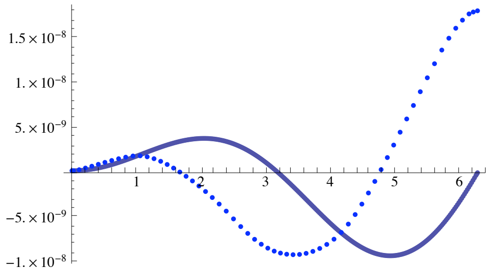
This integrates the simple harmonic oscillator using the classical fourth-order Runge-Kutta method with steps controlled by using the `"DoubleStep"` method.

```
In[5]:= dstep = NDSolve[{x''[t] + x[t] == 0, x[0] == 1, x'[0] == 0},
          x, {t, 0, 2 π}, Method → {"DoubleStep", Method → CRK4}]
Out[5]= {{x → InterpolatingFunction[{{0., 6.28319}}, <>]}}
```

This makes a plot comparing the error in the computed solutions at the step ends. The error for the "DoubleStep" method is shown in blue.

```
In[6]:= ploterror[{sol_}, opts___] := Module[{
            points = x@"Coordinates"[1] /. sol,
            values = x@"ValuesOnGrid" /. sol},
          ListPlot[Transpose[{points, values - Cos[points]}], opts]
         ];
        Show[{
          ploterror[fixed],
          ploterror[dstep, PlotStyle → RGBColor[0, 0, 1]]
         }]
```

Out[7]=



The fixed step size ended up with smaller overall error mostly because the steps are so much smaller; it required more than three times as many steps. For a problem where the local solution structure changes more significantly, the difference can be even greater.

A facility for stiffness detection is described within "DoubleStep Method for NDSolve".

For more sophisticated methods, it may be necessary or more efficient to set up some data for the method to use. When `NDSolve` uses a particular numerical algorithm for the first time, it calls an initialization function. You can define rules for the initialization that will set up appropriate data for your method.

`InitializeMethod`[*Algorithm Identifier*, *stepmode*, *state*, *Algorithm Options*]	
	the expression that `NDSolve` evaluates for initialization when it first uses an algorithm for a particular integration where *stepmode* is either `Automatic` or `Fixed` depending on whether your method is expected to be called within the framework of a step controller or another method or not; *state* is the `NDSolveState` object used by `NDSolve`, and *Algorithm Options* is a list that contains any options given specifically with the specification to use the particular algorithm, for example, {*opts*} in `Method -> {`*Algorithm Identifier*, *opts*`}`

Algorithm Identifier /: `InitializeMethod[`*Algorithm Identifier*`,`*stepmode*`_,`*rhs*`_NumericalFunction,`*state*`_NDSolveState,{`*opts*`___?OptionQ}]:=`*initialization*	
	definition of the initialization so that the rule is associated with the algorithm, and *initialization* should return an algorithm object in the form *Algorithm Identifier*[*data*]

Initializing a method from `NDSolve`.

As a system symbol, `InitializeMethod` is protected, so to attach rules to it, you would need to unprotect it first. It is better to keep the rules associated with your method. A tidy way to do this is to make the initialization definition using `TagSet` as shown earlier.

As an example, suppose you want to redefine the Runge-Kutta method shown earlier so that instead of using the exact coefficients 2, 1/2, and 1/6, numerical values with the appropriate precision are used instead to make the computation slightly faster.

This defines a method function to take a single step toward integrating an ODE using the classical fourth-order Runge-Kutta method using saved numerical values for the required coefficients.

```
In[15]:= CRK4[{two_, half_, sixth_}]["Step"[rhs_, t_, h_, y_, yp_]] :=
          Module[{k0, k1, k2, k3},
           k0 = h yp;
           k1 = h rhs[t + half h, y + half k0];
           k2 = h rhs[t + half h, y + half k1];
           k3 = h rhs[t + h, y + k2];
           {h, sixth (k0 + two (k1 + k2) + k3)}]
```

This defines a rule that initializes the algorithm object with the data to be used later.

```
In[16]:= CRK4 /: NDSolve`InitializeMethod[CRK4,
           stepmode_, rhs_, state_, opts___] := Module[{prec},
           prec = state@"WorkingPrecision";
           CRK4[N[{2, 1 / 2, 1 / 6}, prec]]]
```

Saving the numerical values of the numbers gives between 5 and 10 percent speedup for a longer integration using `"DoubleStep"`.

Adams Methods

In terms of the `NDSolve` framework, it is not really any more difficult to write an algorithm that controls steps automatically. However, the requirements for estimating error and determining an appropriate step size usually make this much more difficult from both the mathematical and programming standpoints. The following example is a partial adaptation of the Fortran DEABM code of Shampine and Watts to fit into the `NDSolve` framework. The algorithm adaptively chooses both step size and order based on criteria described in [SG75].

The first stage is to define the coefficients. The integration method uses variable step-size coefficients. Given a sequence of step sizes $\{h_{n-k+1}, h_{n-k+2}, \ldots, h_n\}$, where h_n is the current step to take, the coefficients for the method with Adams-Bashforth predictor of order k and Adams-Moulton corrector of order $k+1$, $g_j(n)$ such that

$$y_{n+1} = p_{n+1} + h_n\, g_k(n)\, \Phi_k(n+1)$$

$$p_{n+1} = y_n + h_n \sum_{j=0}^{k-1} g_j(n)\, \Phi_k^*(n),$$

where the $\Phi_j(n)$ are the divided differences.

$$\Phi_j(n) == \prod_{i=0}^{j-1} (t_n - t_{n-i})\, \delta^k f\left[t_n, \ldots, t_{n-j}\right]$$

$$(\Phi_j)^*(n) = \beta_j(n)\, \Phi_j(n) \text{ with } \beta_j(n) = \prod_{i=0}^{j-1} \frac{t_{n+1} - t_{n-i}}{t_n - t_{-i+n-1}}.$$

This defines a function that computes the coefficients Φ_j and β_j, along with σ_j, that are used in error estimation. The formulas are from [HNW93] and use essentially the same notation.

```
In[17]:= AdamsBMCoefficients[hlist_List] := Module[{k, h, Δh, brat, β, α, σ, c},
           k = Length[hlist];
           h = Last[hlist];
           Δh = Drop[FoldList[Plus, 0, Reverse[hlist]], 1];
                       Drop[Δh, -1]
           brat = ──────────────────;
                   Drop[Δh, 1] - h
           β = FoldList[Times, 1, brat];
                h
           α = ──;
                Δh
           σ = FoldList[Times, 1, α Range[Length[α]]];
                            1
           c[0] = Table[─, {q, 1, k}];
                            q
                                1
           c[1] = Table[─────────, {q, 1, k}];
                          q (q + 1)
                                                     Drop[c[j - 1], 1] h
           Do[c[j] = Drop[c[j - 1], -1] - ─────────────────────, {j, 2, k}];
                                                          Δh⟦j⟧
           {(First[c[#1]] &) /@ Range[0, k], β, σ}]
```

hlist is the list of step sizes $\{h_{n-k}, h_{n-k+1}, \ldots, h_n\}$ from past steps. The constant-coefficient Adams coefficients can be computed once, and much more easily. Since the constant step size Adams-Moulton coefficients are used in error prediction for changing the method order, it makes sense to define them once with rules that save the values.

This defines a function that computes and saves the values of the constant step size Adams-Moulton coefficients.

```
In[18]:= Moulton[0] = 1;
         Moulton[m_] := Moulton[m] = -Sum[Moulton[k] / (1 + m - k), {k, 0, m - 1}]
```

The next stage is to set up a data structure that will keep the necessary information between steps and define how that data should be initialized. The key information that needs to be saved is the list of past step sizes, *hlist*, and the divided differences, Φ. Since the method does the error estimation, it needs to get the correct norm to use from the `NDSolve`StateData` object. Some other data such as precision is saved for optimization and convenience.

This defines a rule for initializing the `AdamsBM` method from `NDSolve`.

```
In[20]:= AdamsBM /:
           NDSolve`InitializeMethod[AdamsBM, {Automatic, DenseQ_},
             rhs_, ndstate_, opts___] := Module[{prec, norm, hlist, Φ, mord},
             mord = MaxDifferenceOrder /. Flatten[{opts, Options[AdamsBM]}];
             If[mord ≠ ∞ && ! (IntegerQ[mord] && mord > 0), Return[$Failed]];
             prec = ndstate["WorkingPrecision"];
             norm = ndstate["Norm"];
             hlist = {};
             Φ = {ndstate["SolutionDerivativeVector"]["Active"]]};
             AdamsBM[{{hlist, Φ, N[0, prec] Φ〚1〛}, {norm, prec, mord, 0, True}}]];
```

hlist is initialized to $\{\}$ since at initialization time there have been no steps. Φ is initialized to the derivative of the solution vector at the initial condition since the 0th divided difference is just the function value. Note that Φ is a matrix. The third element, which is initialized to a zero vector, is used for determining the best order for the next step. It is effectively an additional divided difference. The use of the other parts of the data is clarified in the definition of the stepping function.

The initialization is also set up to get the value of an option that can be used to limit the maximum order of the method to use. In the code DEABM, this is limited to 12, because this is a practical limit for machine-precision calculations. However, in *Mathematica*, computations can be done in higher precision where higher-order methods may be of significant advantage, so there is no good reason for an absolute limit of this sort. Thus, you set the default of the option to be ∞.

This sets the default for the `MaxDifferenceOrder` option of the `AdamsBM` method.

```
In[21]:= Options[AdamsBM] = {MaxDifferenceOrder → ∞};
```

Before proceeding to the more complicated `"Step"` method functions, it makes sense to define the simple `"StepMode"` and `"DifferenceOrder"` method functions.

This defines the step mode for the `AdamsBM` method to always be `Automatic`. This means that it cannot be called from step controller methods that request fixed step sizes of possibly varying sizes.

```
In[22]:= AdamsBM[___]["StepMode"] = Automatic;
```

This defines the difference order for the `AdamsBM` method. This varies with the number of past values saved.

```
In[23]:= AdamsBM[data_]["DifferenceOrder"] := Length[data[[1, 2]]];
```

Finally, here is the definition of the `"Step"` function. The actual process of taking a step is only a few lines. The rest of the code handles the automatic step size and order selection following very closely the DEABM code of Shampine and Watts.

This defines the `"Step"` method function for `AdamsBM` that returns step data according to the templates described earlier.

```
In[24]:= AdamsBM[data_]["Step"[rhs_, t_, h_, y_, yp_]] :=
           Module[{prec, norm, hlist, Φ, Φ1, ns, starting, k, zero,
             g, β, σ, p, f, Δy, normh, ev, err, PE, knew, hnew, temp},
            {{hlist, Φ, Φ1}, {norm, prec, mord, ns, starting}} = data;
            (* Norm scaling will be based on current solution y. *)
            normh = (Abs[h] temp[#1, y] &) /. {temp → norm};
            k = Length[Φ];
            zero = N[0, prec];
            (* Keep track of number of steps at this stepsize h. *)
            If[Length[hlist] > 0 && Last[hlist] == h, ns ++, ns = 1];
            hlist = Join[hlist, {h}];
            {g, β, σ} = AdamsBMCoefficients[hlist];
            (* Convert Φ to Φ* *)
            Φ = Φ Reverse[β];
            (* PE: Predict and evaluate *)
            p = Reverse[Drop[g, -1]].Φ;
            f = rhs[h + t, h p + y];
            (* Update divided differences *)
            Φ = FoldList[Plus, zero Φ1, Φ];
            (* Compute scaled error estimate *)
            ev = f - Last[Φ];
            err = (g⟦-2⟧ - g⟦-1⟧) normh[ev];
            (* First order check: determines if order should be lowered
             even in the case of a rejected step *)
            knew = OrderCheck[PE, k, Φ, ev, normh, σ];
            If[err > 1,
             (* Rejected step: reduce h by half,
             make sure starting mode flag is unset and reset Φ to previous values *)
```

```
         h
 hnew = —; Δy = $Failed; f = None; starting = False; Φ = data⟦1, 2⟧,
         2
 (* Sucessful step:
   CE: Correct and evaluate *)
 Δy = h (p + ev Last[g]);
 f = rhs[h + t, y + Δy]; temp = f - Last[Φ];
 (* Update the divided differences *)
 Φ = (temp + #1 &) /@ Φ;
 (* Determine best order and stepsize for the next step *)
 Φ1 = temp - Φ1;
 knew = ChooseNextOrder[starting, PE, k, knew, Φ1, normh, σ, mord, ns];
 hnew = ChooseNextStep[PE, knew, h]];
(* Truncate hlist and Φ to the appropriate length for the chosen order. *)
hlist = Take[hlist, 1 - knew];
If[Length[Φ] > knew, Φ1 = Φ⟦Length[Φ] - knew⟧; Φ = Take[Φ, -knew];];
(* Return step data along with updated method data *)
{hnew, Δy, f, AdamsBM[{{hlist, Φ, Φ1}, {norm, prec, mord, ns, starting}}]}];
```

There are a few deviations from DEABM in the code here. The most significant is that coefficients are recomputed at each step, whereas DEABM computes only those that need updating. This modification was made to keep the code simpler, but does incur a clear performance loss, particularly for small to moderately sized systems. A second significant modification is that much of the code for limiting rejected steps is left to `NDSolve`, so there are no checks in this code to see if the step size is too small or the tolerances are too large. The stiffness detection heuristic has also been left out. The order and step-size determination code has been modularized into separate functions.

This defines a function that constructs error estimates PE_j for $j == k-2$, $k-1$, and k and determines if the order should be lowered or not.

```
In[25]:= OrderCheck[PE_, k_, Φ_, ev_, normh_, σ_] := Module[{knew = k},
            PE_k = Abs[σ⟦k + 1⟧ Moulton[k] normh[ev]]; If[k > 1,
             PE_k-1 = Abs[σ⟦k⟧ Moulton[k - 1] normh[ev + Φ⟦2⟧]];
             If[k > 2,
              PE_k-2 = Abs[σ⟦k - 1⟧ Moulton[k - 2] normh[ev + Φ⟦3⟧]];
              If[Max[PE_k-1, PE_k-2] < PE_k, knew = k - 1]],
                          PE_k
             If[PE_k-1 < ————, knew = k - 1];
                           2
            ];
            knew
           ];
```

This defines a function that determines the best order to use after a successful step.

```
In[26]:= SetAttributes[ChooseNextOrder, HoldFirst];
         ChooseNextOrder[starting_, PE_, k_, knw_, Φ1_, normh_, σ_, mord_, ns_] :=
           Module[{knew = knw},
            starting = starting && knew ≥ k && k < mord;
```

```
  If[starting,
   knew = k + 1; PE[k+1] = 0,
   If[knew ≥ k && ns ≥ k + 1,
     PE[k+1] = Abs[Moulton[k + 1] normh[Φ1]];
     If[k > 1,
       If[PE[k-1] ≤ Min[PE[k], PE[k+1]],
        knew = k - 1,
        If[PE[k+1] < PE[k] && k < mord, knew = k + 1]
       ],
       If[PE[k+1] < PE[k]/2, knew = k + 1]
     ];
   ];
  ];
  knew
 ];
```

This defines a function that determines the best step size to use after a successful step of size h.

```
In[28]:= ChooseNextStep[PE_, k_, h_] :=
           If[PE[k] < 2^-(k+2),
            2 h,
            If[PE[k] < 1/2, h, h Max[1/2, Min[9/10, (1/(2 PE[k]))^(1/(k+1))]]]
           ];
```

Once these definitions are entered, you can access the method in `NDSolve` by simply using `Method -> AdamsBM`.

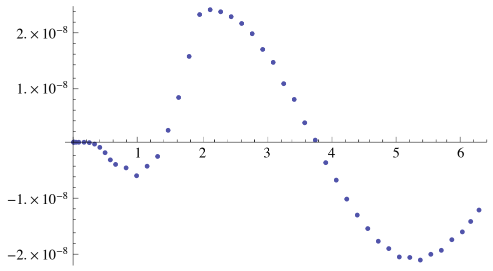
This solves the harmonic oscillator equation with the Adams method defined earlier.

```
In[29]:= asol = NDSolve[{x''[t] + x[t] == 0, x[0] == 1, x'[0] == 0},
           x, {t, 0, 2 π}, Method → AdamsBM]
Out[29]= {{x → InterpolatingFunction[{{0., 6.28319}}, <>]}}
```

This shows the error of the computed solution. It is apparent that the error is kept within reasonable bounds. Note that after the first few points, the step size has been increased.

```
In[30]:= ploterror[asol]
```

Out[30]=



Where this method has the potential to outperform some of the built-in methods is with high-precision computations with strict tolerances. This is because the built-in methods are adapted from codes with the restriction to order 12.

```
In[31]:= LorenzEquations = {
            {x'[t] == -3 (x[t] - y[t]), x[0] == 0},
            {y'[t] == -x[t] z[t] + 53 / 2 x[t] - y[t], y[0] == 1},
            {z'[t] == x[t] y[t] - z[t], z[0] == 0}};
         vars = {x[t], y[t], z[t]};
```

A lot of time is required for coefficient computation.

```
In[33]:= Timing[NDSolve[LorenzEquations, vars, {t, 0, 20}, Method → AdamsBM]]

Out[33]= {7.04 Second, {{x[t] → InterpolatingFunction[{{0., 20.}}, <>][t],
            y[t] → InterpolatingFunction[{{0., 20.}}, <>][t],
            z[t] → InterpolatingFunction[{{0., 20.}}, <>][t]}}}
```

This is not using as high an order as might be expected.

In any case, about half the time is spent generating coefficients, so to make it better, you need to figure out the coefficient update.

```
In[34]:= Timing[NDSolve[LorenzEquations, vars,
            {t, 0, 20}, Method → AdamsBM, WorkingPrecision → 32]]

Out[34]= {11.109, {{x[t] → InterpolatingFunction[{{0, 20.000000000000000000000000000000}}, <>][t],
            y[t] → InterpolatingFunction[{{0, 20.000000000000000000000000000000}}, <>][t],
            z[t] → InterpolatingFunction[{{0, 20.000000000000000000000000000000}}, <>][t]}}}
```

Numerical Solution of Partial Differential Equations

The Numerical Method of Lines

Introduction

The numerical method of lines is a technique for solving partial differential equations by discretizing in all but one dimension, and then integrating the semi-discrete problem as a system of ODEs or DAEs. A significant advantage of the method is that it allows the solution to take advantage of the sophisticated general-purpose methods and software that have been developed for numerically integrating ODEs and DAEs. For the PDEs to which the method of lines is applicable, the method typically proves to be quite efficient.

It is necessary that the PDE problem be well-posed as an initial value (Cauchy) problem in at least one dimension, since the ODE and DAE integrators used are initial value problem solvers. This rules out purely elliptic equations such as Laplace's equation, but leaves a large class of evolution equations that can be solved quite efficiently.

A simple example illustrates better than mere words the fundamental idea of the method. Consider the following problem (a simple model for seasonal variation of heat in soil).

$$u_t == \frac{1}{8} u_{xx},\ u(0,\ t) == \sin(2\pi t),\ u_x(1,\ t) == 0,\ u(x,\ 0) == 0 \tag{1}$$

This is a candidate for the method of lines since you have the initial value $u(x,\ 0) == 0$.

Problem (1) will be discretized with respect to the variable x using second-order finite differences, in particular using the approximation

$$u_{xx}(x,\ t) \simeq \frac{u(x+h,t)-2\,u(x,t)-u(x-h,t)}{h^2} \tag{2}$$

Even though finite difference discretizations are the most common, there is certainly no requirement that discretizations for the method of lines be done with finite differences; finite volume or even finite element discretizations can also be used.

To use the discretization shown, choose a uniform grid x_i, $0 \le i \le n$ with spacing $h == 1/n$ such that $x_i == i\,h$. Let $u_i[t]$ be the value of $u(x_i, t)$. For the purposes of illustrating the problem setup, a particular value of n is chosen.

This defines a particular value of n and the corresponding value of h used in the subsequent commands. This can be changed to make a finer or coarser spatial approximation.

```
In[1]:=  n = 10; hn = 1/n;
```

This defines the vector of u_i.

```
In[2]:=  U[t_] = Table[ui[t], {i, 0, n}]
Out[2]=  {u0[t], u1[t], u2[t], u3[t], u4[t], u5[t], u6[t], u7[t], u8[t], u9[t], u10[t]}
```

For $1 \le i \le 9$, you can use the centered difference formula (2) to obtain a system of ODEs. However, before doing this, it is useful to incorporate the boundary conditions first.

The Dirichlet boundary condition at $x == 0$ can easily be handled by simply defining u_0 as a function of t. An alternative option is to differentiate the boundary condition with respect to time and use the corresponding differential equation. In this example, the latter method will be used.

The Neumann boundary condition at $x == 1$ is a little more difficult. With second-order differences, one way to handle it is with reflection: imagine that you are solving the problem on the interval $0 \le x \le 2$ with the same boundary conditions at $x == 0$ and $x == 2$. Since the initial condition and boundary conditions are symmetric with respect to x, the solution should be symmetric with respect to x for all time, and so symmetry is equivalent to the Neumann boundary condition at $x = 1$. Thus, $u(1 + h, t) == u(1 - h, t)$, so $u_{n+1}[t] == u_{n-1}[t]$.

This uses `ListCorrelate` to apply the difference formula. The padding $\{u_{n-1}[t]\}$ implements the Neumann boundary condition.

```
In[3]:= eqns = Thread[D[U[t], t] == Join[{D[Sin[2 π t], t]},
            ListCorrelate[{1, -2, 1} / h_n^2, U[t], {1, 2}, {u_{n-1}[t]}] / 8]]
```

$$\Big\{u_0'[t] == 2\pi \text{Cos}[2\pi t],\ u_1'[t] == \frac{1}{8}(100\, u_0[t] - 200\, u_1[t] + 100\, u_2[t]),$$
$$u_2'[t] == \frac{1}{8}(100\, u_1[t] - 200\, u_2[t] + 100\, u_3[t]),$$
$$u_3'[t] == \frac{1}{8}(100\, u_2[t] - 200\, u_3[t] + 100\, u_4[t]),\ u_4'[t] == \frac{1}{8}(100\, u_3[t] - 200\, u_4[t] + 100\, u_5[t]),$$
$$u_5'[t] == \frac{1}{8}(100\, u_4[t] - 200\, u_5[t] + 100\, u_6[t]),\ u_6'[t] == \frac{1}{8}(100\, u_5[t] - 200\, u_6[t] + 100\, u_7[t]),$$
$$u_7'[t] == \frac{1}{8}(100\, u_6[t] - 200\, u_7[t] + 100\, u_8[t]),\ u_8'[t] == \frac{1}{8}(100\, u_7[t] - 200\, u_8[t] + 100\, u_9[t]),$$
$$u_9'[t] == \frac{1}{8}(100\, u_8[t] - 200\, u_9[t] + 100\, u_{10}[t]),\ u_{10}'[t] == \frac{1}{8}(200\, u_9[t] - 200\, u_{10}[t])\Big\}$$

This sets up the zero initial condition.

```
In[4]:= initc = Thread[U[0] == Table[0, {n + 1}]]

Out[4]= {u0[0] == 0, u1[0] == 0, u2[0] == 0, u3[0] == 0, u4[0] == 0,
         u5[0] == 0, u6[0] == 0, u7[0] == 0, u8[0] == 0, u9[0] == 0, u10[0] == 0}
```

Now the PDE has been partially discretized into an ODE initial value problem that can be solved by the ODE integrators in `NDSolve`.

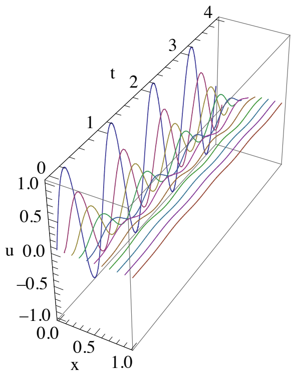
This solves the ODE initial value problem.

```
In[5]:= lines = NDSolve[{eqns, initc}, U[t], {t, 0, 4}]

Out[5]= {{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 the solutions $u(x_i, t)$ plotted as a function of x and t.

```
In[6]:= ParametricPlot3D[Evaluate[Table[{i hn, t, First[ui[t] /. lines]}, {i, 0, n}]],
         {t, 0, 4}, PlotRange → All, AxesLabel → {"x", "t", "u"}]
```

Out[6]=



The plot indicates why this technique is called the numerical "method of lines".

The solution in between lines can be found by interpolation. When `NDSolve` computes the solution for the PDE, the result is a two-dimensional `InterpolatingFunction`.

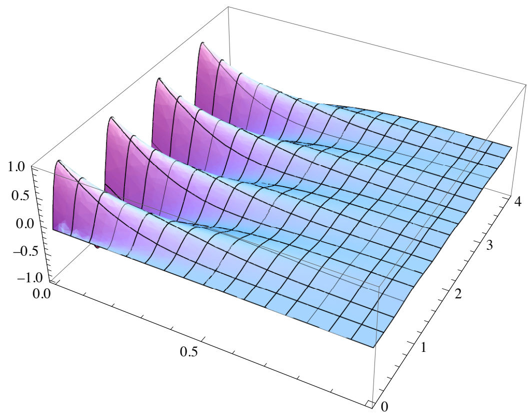
This uses `NDSolve` to compute the solution of the heat equation (1) directly.

```
In[7]:= solution = NDSolve[{D[u[x, t], t] == 1/8 D[u[x, t], x, x], u[x, 0] == 0,
           u[0, t] == Sin[2 π t], (D[u[x, t], x] /. x → 1) == 0}, u, {x, 0, 1}, {t, 0, 4}]
Out[7]= {{u → InterpolatingFunction[{{0., 1.}, {0., 4.}}, <>]}}
```

This creates a surface plot of the solution.

```
In[8]:= Plot3D[Evaluate[First[u[x, t] /. solution]],
         {x, 0, 1}, {t, 0, 4}, PlotPoints → {14, 36}, PlotRange → All]
```

Out[8]=



The setting $n == 10$ used did not give a very accurate solution. When `NDSolve` computes the solution, it uses spatial error estimates on the initial condition to determine what the grid spacing should be. The error in the temporal (or at least time-like) variable is handled by the adaptive ODE integrator.

In the example (1), the distinction between time and space was quite clear from the problem context. Even when the distinction is not explicit, this tutorial will refer to "spatial" and "temporal" variables. The "spatial" variables are those to which the discretization is done. The "temporal" variable is the one left in the ODE system to be integrated.

option name	*default value*	
`TemporalVariable`	`Automatic`	what variable to keep derivatives with respect to the derived ODE or DAE system
`Method`	`Automatic`	what method to use for integrating the ODEs or DAEs
`SpatialDiscretization`	`TensorProductGrid`	what method to use for spatial discretization
`DifferentiateBoundaryConditions`	`True`	whether to differentiate the boundary conditions with respect to the temporal variable
`ExpandFunctionSymbolically`	`False`	whether to expand the effective function symbolically or not
`DiscretizedMonitorVariables`	`False`	whether to interpret dependent variables given in monitors like `StepMonitor` or in method options for methods like `EventLocator` and `Projection` as functions of the spatial variables or vectors representing the spatially discretized values

Options for `NDSolve`MethodOfLines`.

Use of some of these options requires further knowledge of how the method of lines works and will be explained in the sections that follow.

Currently, the only method implemented for spatial discretization is the `TensorProductGrid` method, which uses discretization methods for one spatial dimension and uses an outer tensor product to derive methods for multiple spatial dimensions on rectangular regions. `TensorProductGrid` has its own set of options that you can use to control the grid selection process. The following sections give sufficient background information so that you will be able to use these options if necessary.

Spatial Derivative Approximations

Finite Differences

The essence of the concept of finite differences is embodied in the standard definition of the derivative

$$f'(x_i) == \lim_{h \to 0} \frac{f(h + x_i) - f(x_i)}{h}$$

where instead of passing to the limit as h approaches zero, the finite spacing to the next adjacent point, $x_{i+1} == x_i + h$, is used so that you get an approximation.

$$f'(x_i)_{\text{approx}} == \frac{f(x_{i+1}) - f(x_i)}{h}$$

The difference formula can also be derived from Taylor's formula,

$$f(x_{i+1}) == f(x_i) + h\, f'(x_i) + \frac{h^2}{2} f''(\xi_i);\ x_i < \xi_i < x_{i+1}$$

which is more useful since it provides an error estimate (assuming sufficient smoothness)

$$f'(x_i) == \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{h}{2} f''(\xi_i)$$

An important aspect of this formula is that ξ_i must lie between x_i and x_{i+1} so that the error is local to the interval enclosing the sampling points. It is generally true for finite difference formulas that the error is local to the stencil, or set of sample points. Typically, for convergence and other analysis, the error is expressed in asymptotic form:

$$f'(x_i) == \frac{f(x_{i+1}) - f(x_i)}{h} + O(h)$$

This formula is most commonly referred to as the first-order forward difference. The backward difference would use x_{i-1}.

Taylor's formula can easily be used to derive higher-order approximations. For example, subtracting

$$f(x_{i+1}) == f(x_i) + h\, f'(x_i) + \frac{h^2}{2} f''(x_i) + O\left(h^3\right)$$

from

$$f(x_{i-1}) == f(x_i) - h\, f'(x_i) + \frac{h^2}{2} f''(x_i) + O\left(h^3\right)$$

and solving for $f'(x_i)$ gives the second-order centered difference formula for the first derivative,

$$f'(x_i) == \frac{f(x_{i+1}) - f(x_{i-1})}{2\, h} + O\left(h^2\right)$$

If the Taylor formulas shown are expanded out one order farther and added and then combined with the formula just given, it is not difficult to derive a centered formula for the second derivative.

$$f''(x_i) == \frac{f(x_{i+1}) - 2\, f(x_i) + f(x_{i-1})}{h^2} + O\left(h^2\right)$$

Note that the while having a uniform step size h between points makes it convenient to write out the formulas, it is certainly not a requirement. For example, the approximation to the second derivative is in general

$$f''(x_i) == \frac{2\,(f(x_{i+1})\,(x_{i-1} - x_i) + f(x_{i-1})\,(x_i - x_{i+1}) + f(x_i)\,(x_{i+1} - x_{i-1}))}{(x_{i-1} - x_i)\,(x_{i-1} - x_{i+1})\,(x_i - x_{i+1})} + O(h)$$

where h corresponds to the maximum local grid spacing. Note that the asymptotic order of the three-point formula has dropped to first order; that it was second order on a uniform grid is due to fortuitous cancellations.

In general, formulas for any given derivative with asymptotic error of any chosen order can be derived from the Taylor formulas as long as a sufficient number of sample points are used. However, this method becomes cumbersome and inefficient beyond the simple examples shown. An alternate formulation is based on polynomial interpolation: since the Taylor formulas are exact (no error term) for polynomials of sufficiently low order, so are the finite difference