

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

©2008 Wolfram Research, Inc.

All rights reserved. No part of this document may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the copyright holder.

Wolfram Research is the holder of the copyright to the Wolfram *Mathematica* software system ("Software") described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, "look and feel," programming language, and compilation of command names. Use of the Software unless pursuant to the terms of a license granted by Wolfram Research or as otherwise authorized by law is an infringement of the copyright.

Wolfram Research, Inc. and Wolfram Media, Inc. ("Wolfram") make no representations, express, statutory, or implied, with respect to the Software (or any aspect thereof), including, without limitation, any implied warranties of merchantability, interoperability, or fitness for a particular purpose, all of which are expressly disclaimed. Wolfram does not warrant that the functions of the Software will meet your requirements or that the operation of the Software will be uninterrupted or error free. As such, Wolfram does not recommend the use of the software described in this document for applications in which errors or omissions could threaten life, injury or significant loss.

Mathematica, *MathLink*, and *MathSource* are registered trademarks of Wolfram Research, Inc. *J/Link*, *MathLM*, *.NET/Link*, and *webMathematica* are trademarks of Wolfram Research, Inc. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Macintosh is a registered trademark of Apple Computer, Inc. All other trademarks used herein are the property of their respective owners. *Mathematica* is not associated with Mathematica Policy Research, Inc.

Contents

Introduction	1
Overview	1
The Design of the NDSolve Framework	11
ODE Integration Methods	17
Methods	17
Controller Methods	66
Extensions	162
Partial Differential Equations	174
The Numerical Method of Lines	174
Boundary Value Problems	243
Shooting Method	243
Chasing Method	248
Boundary Value Problems with Parameters	255
Differential-Algebraic Equations	256
Introduction	256
IDA Method	264
Delay Differential Equations	274
Comparison and Contrast with ODEs	275
Propagation and Smoothing of Discontinuities	280
Storing History Data	284
The Method of Steps	285
Examples	290
Norms in NDSolve	294
ScaledVectorNorm	296
Stiffness Detection	298
Overview	298
Introduction	299
Linear Stability	301
"StiffnessTest" Method Option	304
"NonstiffTest" Method Option	305
Examples	315
Option Summary	323

Structured Systems	324
Numerical Methods for Solving the Lotka-Volterra Equations	324
Rigid Body Solvers	329
Components and Data Structures	339
Introduction	339
Example	340
Creating NDSolve`StateData Objects	341
Iterating Solutions	343
Getting Solution Functions	344
NDSolve`StateData methods	348
DifferentialEquations Utility Packages	351
InterpolatingFunctionAnatomy	351
NDSolveUtilities	356
References	358

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.

<code>NDSolve [{eqn₁, eqn₂, ...} , $u, \{t, t_{min}, t_{max}\}]$</code>	find a numerical solution for the function u with t in the range t_{min} to t_{max}
<code>NDSolve [{eqn₁, eqn₂, ...} , $\{u_1, u_2, \dots\}, \{t, t_{min}, t_{max}\}]$</code>	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 prob-

lems 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.

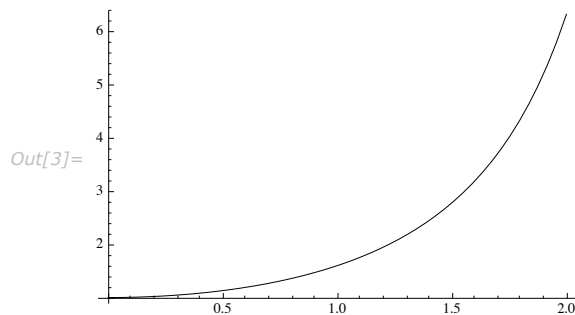
This solves an initial value problem for a second-order equation, which requires two conditions, and are given at $t = 0$.

```
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}]
```



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.}}, <>]}}
```

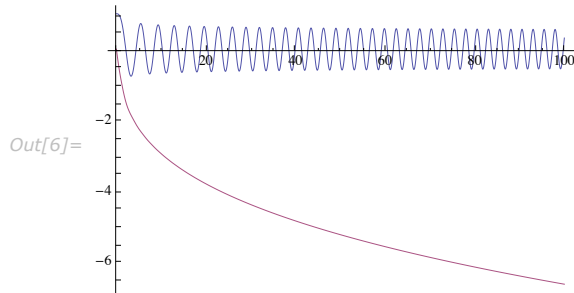
You can use `NDSolve` to solve systems of coupled differential equations as long as each variable has the appropriate number of conditions.

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]
```



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

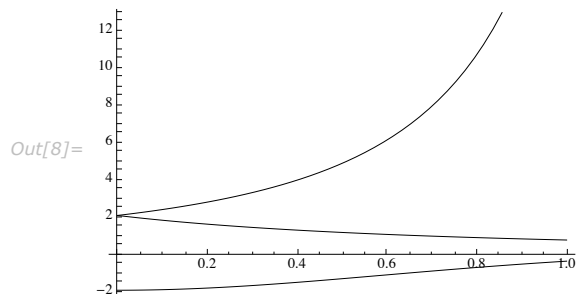
The initial conditions in this case lead to multiple solutions.

```
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}]
```



`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. >>

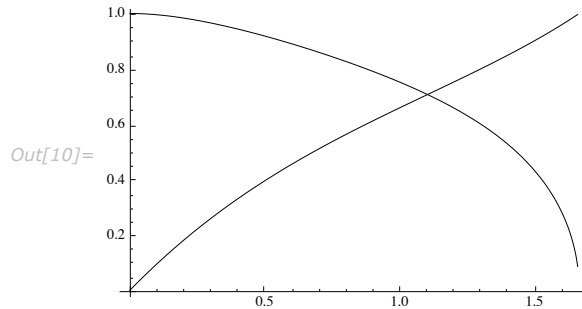
```
Out[9]= {{x -> InterpolatingFunction[{{0., 1.66565}}, <>], y -> InterpolatingFunction[{{0., 1.66565}}, <>]}}
```

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}]
```



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[{x_i'[t] == n^2 (x_{i+1}[t] - 2 x_i[t] + x_{i-1}[t]), x_i[0] == (i/n)^10}, {i, n-1}];
vars = Table[x_i[t], {i, n-1}];
NDSolve[eqns, vars, {t, 0, .25}]
```

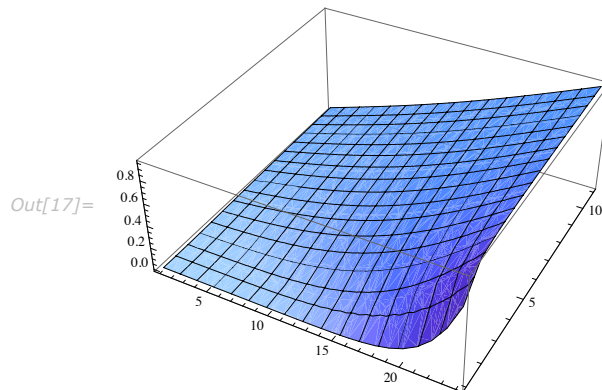
Out[16]=

```
{{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}]]
```



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[{eqn1, eqn2, ...}, u, {t, tmin, tmax}, {x, xmin, xmax}, ...]
```

solve a system of partial differential equations for a function $u[t, x \dots]$ with t in the range $t_{\min}$ to $t_{\max}$ and x in the range $x_{\min}$ to $x_{\max}$, ...

```
NDSolve[{eqn1, eqn2, ...}, {u1, u2, ...}, {t, tmin, tmax}, {x, xmin, xmax}, ...]
```

solve a system of partial differential equations for several functions u_i

Finding numerical solutions to partial differential equations.

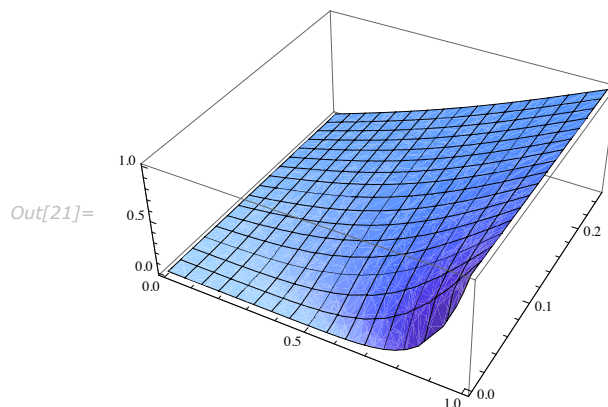
Here is a solution of the heat equation found directly by NDSolve.

```
In[20]:= NDSolve[{D[u[x, t], t] == D[u[x, t], x, x], u[x, 0] == x10,  
               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}]
```



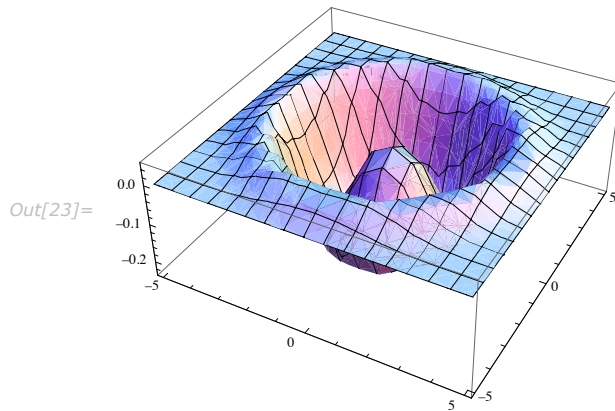
NDSolve currently uses the numerical method of lines to compute solutions to partial differential equations. The method is restricted to problems that can be posed with an initial condition in at least one independent variable. For example, the method cannot solve elliptic PDEs such as Laplace's equation 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}]
```



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

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

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, 1.000000000000000000000000000000}}, <>]}}
```

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.

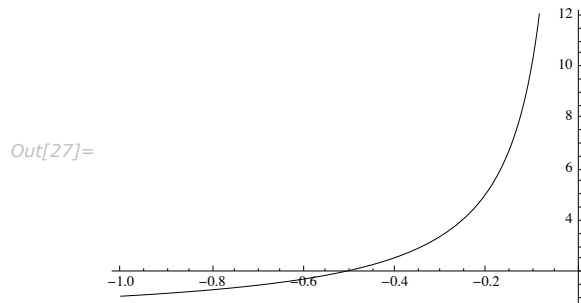
```
In[26]:= NDSolve[{y'[x] == 1/x^2, y[-1] == 1}, y[x], {x, -1, 0}]
```

NDSolve::mxst: Maximum number of 10000 steps reached at the point $x == -1.00413 \times 10^{-172}$. >>

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

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

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



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.

NDSolve → "ExplicitRungeKutta"

NDSolve → "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.

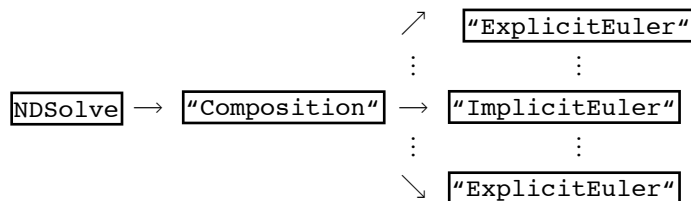
NDSolve → "Extrapolation" → "ExplicitMidpoint"

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

NDSolve → "Projection" → "ExplicitRungeKutta"

Second Revision

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

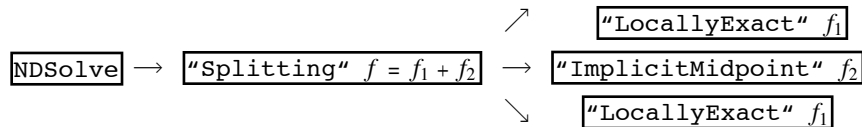


This is an example of a method composed with its adjoint.

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".



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”.

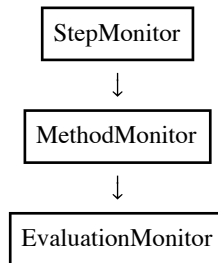
```
In[1]:= Options[NDSolve`ExplicitRungeKutta]

Out[1]= {Coefficients → EmbeddedExplicitRungeKuttaCoefficients, DifferenceOrder → Automatic,
  EmbeddedDifferenceOrder → Automatic, StepSizeControlParameters → Automatic,
  StepSizeRatioBounds →  $\left\{-\frac{1}{8}, 4\right\}$ , 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:

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

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*.

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

The midpoint method can be shown to have a local error of $O(h^3)$, 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:

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

where s is the number of stages.

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

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

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:

$$a_{i,j} = 0, \quad j \geq i, \quad j = 1, \dots, s.$$

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

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

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):

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

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

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

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

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

The quantity $\|\text{err}_n\|$ 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:

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

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[{{(Y1)'[T] == 1 - 4 Y1[T] + Y1[T]^2 Y2[T], (Y2)'[T] == 3 Y1[T] - Y1[T]^2 Y2[T]},  
  {Y1[0] ==  $\frac{3}{2}$ , Y2[0] == 3}, {Y1[T], Y2[T]}, {T, 0, 20}, {}, {}, {}]}
```

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

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

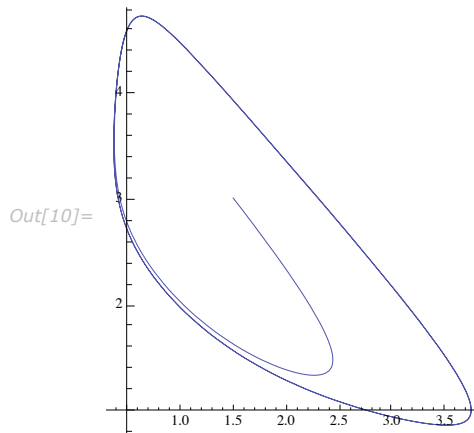
```
Out[8]= {{Y1[T] -> InterpolatingFunction[{{0., 20.}}, <>][T],  
  Y2[T] -> InterpolatingFunction[{{0., 20.}}, <>][T]}}
```

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"]]]
```



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]= {{{{1}, { $\frac{1}{2}$ ,  $\frac{1}{2}$ }}, { $\frac{1}{2}$ ,  $\frac{1}{2}$ , 0}}, {1, 1}}, {{ $-\frac{1}{2}$ ,  $\frac{2}{3}$ ,  $-\frac{1}{6}$ }}}
```

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}];

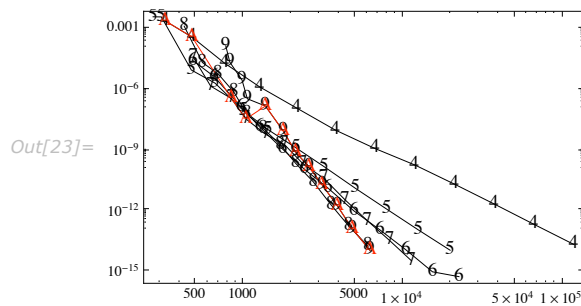
methods = Table[{"ExplicitRungeKutta", "DifferenceOrder" → Part[orders, i],
  "StiffnessTest" → False}, {i, Length[orders]}];
```

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}}]
```



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]:= Fehlbbergamat = {
  {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}};
Fehlbbergbvec = {25 / 216, 0, 1408 / 2565, 2197 / 4104, -1 / 5, 0};
Fehlbbergcvec = {1 / 4, 3 / 8, 12 / 13, 1, 1 / 2};
Fehlbbergevec = {-1 / 360, 0, 128 / 4275, 2197 / 75240, -1 / 50, -2 / 55};
FehlbbergCoefficients[4, p_] :=
  N[{Fehlbbergamat, Fehlbbergbvec, Fehlbbergcvec, Fehlbbergevec}, 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},
  {19372 / 6561, -25360 / 2187, 64448 / 6561, -212 / 729},
  {9017 / 3168, -355 / 33, 46732 / 5247, 49 / 176, -5103 / 18656},
  {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 / 57600, 0, -71 / 16695, 71 / 1920, -17253 / 339200, 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:

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

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]
```

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

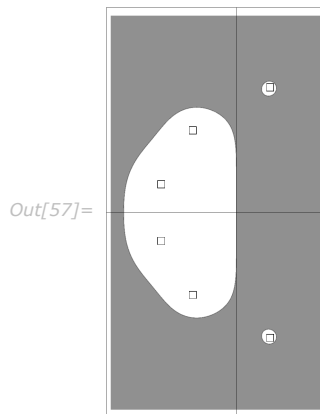
This function finds the linear stability function $R(z)$ for Runge-Kutta methods. The form depends on the coefficients and is a polynomial if the Runge-Kutta method is explicit.

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]
```



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:

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

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 hJ .

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 $c_{s-1} = c_s = 1$.

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

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.

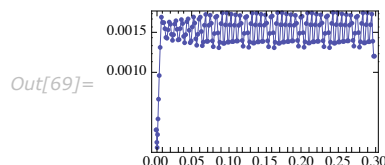
```
In[66]:= system = GetNDSolveProblem["Robertson"];
```

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]
```



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:

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

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"` $\rightarrow \{k_1, k_2\}$ 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:

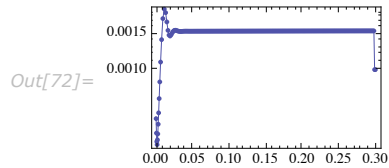
$$k_1 = 3/10, k_2 = 2/5$$

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]
```



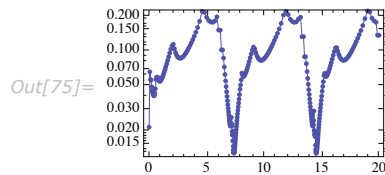
Nonstiff Problem

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.

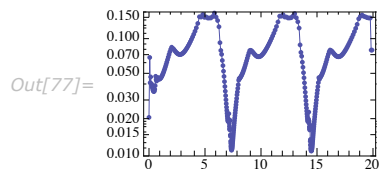
```
In[73]:= system = GetNDSolveProblem["BrusselatorODE"];
```

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



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

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



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"` $\rightarrow \{s_1, s_2\}$ specifies the safety factors to use in the step-size estimate so that (1) becomes:

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

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

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

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

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"`
- `"ImplicitRungeKuttaLobattoIIICoefficients"`

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]= {{{1/4, 1/12 (3 - 2 Sqrt[3])}, {1/12 (3 + 2 Sqrt[3]), 1/4}}, {{1/2, 1/2}, {1/6 (3 - Sqrt[3]), 1/6 (3 + Sqrt[3])}}
```

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

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

```
Out[7]= {{{{0.04000000000000000000, -0.087618018725274235050,
0.085317987638600293760, -0.055818078483298114837, 0.018118109569972056127},
{0.04000000000000000000, 0.12875675325490976116, -0.047477730403197434295,
0.026776985967747870688, -0.0082961444756796453993},
{0.04000000000000000000, 0.23310008036710237092, 0.16758507013524896344,
-0.032883343543501401775, 0.0086077606722332473607},
{0.04000000000000000000, 0.21925333267709602305, 0.33134489917971587453,
0.14621486784749350665, -0.013656113342429231907},
{0.04000000000000000000, 0.22493691761630663460, 0.30390571559725175840,
0.30105430635402060050, 0.072998864317903324306}},
{{0.04000000000000000000, 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]= {-1/(400 (Y1[T]^2 + Y2[T]^2)^(3/2)) - 1/(sqrt(Y1[T]^2 + Y2[T]^2)) + 1/2 (Y3[T]^2 + Y4[T]^2), -Y2[T] Y3[T] + Y1[T] Y4[T]}
```

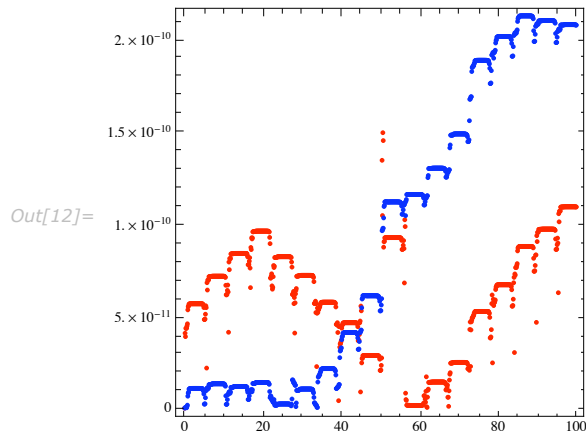
This solves the system using an implicit Runge-Kutta Gauss method. The order of the scheme is selected using the "DifferenceOrder" method option.

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

```
Out[11]= {{Y1[T] -> InterpolatingFunction[{{0., 100.}}, <>][T],
Y2[T] -> InterpolatingFunction[{{0., 100.}}, <>][T],
Y3[T] -> InterpolatingFunction[{{0., 100.}}, <>][T],
Y4[T] -> 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]
```



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.

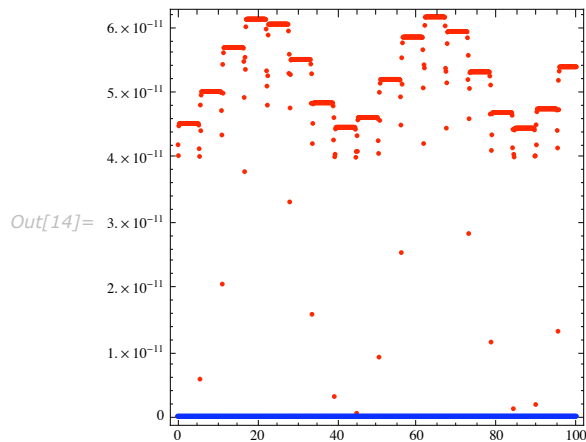
```
In[13]:= sol = NDSolve[system,
  Method -> {"FixedStep", Method -> {"ImplicitRungeKutta", "DifferenceOrder" -> 10,
    "ImplicitSolver" -> {"Newton", AccuracyGoal -> MachinePrecision,
      PrecisionGoal -> MachinePrecision,
      "IterationSafetyFactor" -> 1}}}, StartingStepSize -> 1 / 10]
```

```
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]
```



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

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

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

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

Here ∇ represents the gradient operator:

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

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.

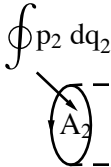
A numerical method applied to a Hamiltonian system (2) is said to be symplectic if it produces a symplectic map. That is, let $(p^*, q^*) = \psi(p, q)$ be a C^1 transformation defined in a domain Ω :

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

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.



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:

(1)

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:

$$\left| \begin{array}{cccc} 0 & \cdots & 0 & 0 \\ b_1 & 0 & \cdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ b_1 & \cdots & b_{s-1} & 0 \end{array} \right| \left| \begin{array}{cccc} B_1 & \cdots & 0 & 0 \\ B_1 & B_2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ B_1 & B_2 & \cdots & B_s \end{array} \right| \quad (1)$$

$$\left| \begin{array}{cccc} b_1 & \cdots & b_{s-1} & b_s \end{array} \right| \left| \begin{array}{cccc} B_1 & B_2 & \cdots & B_s \end{array} \right|$$

or the form:

$$\left| \begin{array}{cccc} b_1 & \cdots & 0 & 0 \\ b_1 & b_2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & \cdots & b_s \end{array} \right| \left| \begin{array}{cccc} 0 & \cdots & 0 & 0 \\ B_1 & 0 & \cdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ B_1 & \cdots & B_{s-1} & 0 \end{array} \right| \quad (2)$$

$$\left| \begin{array}{cccc} b_1 & b_2 & \cdots & b_s \end{array} \right| \left| \begin{array}{cccc} B_1 & \cdots & B_{s-1} & B_s \end{array} \right|$$

The $2d$ free parameters of (2) are sometimes represented using the shorthand notation $[b_1, \dots, b_s](B_1, \dots, B_s)$.

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

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

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 $i = 1, \dots, s$

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

$$Q_{i+1} = Q_i + h_{n+1} B_i g(P_i)$$

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

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

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:

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

The equations of motion are given by:

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

Input

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

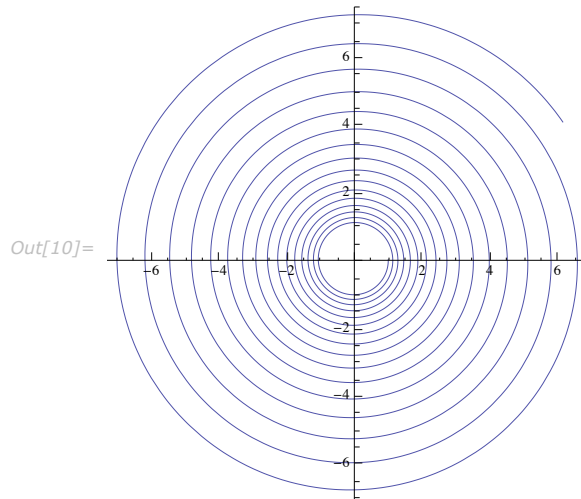
Explicit Euler Method

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];
```

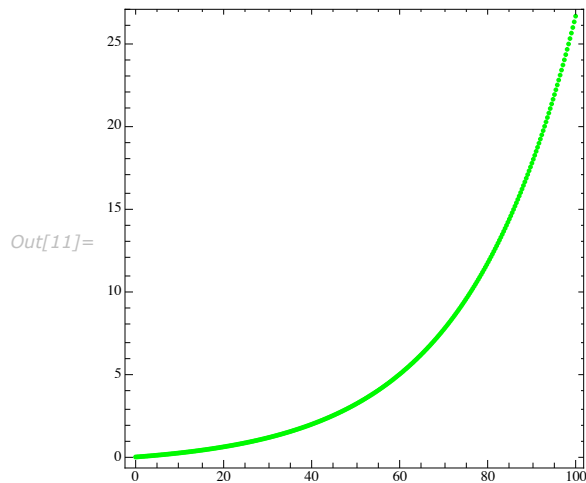
Since the method is dissipative, the trajectory spirals into or away from the fixed point at the origin.

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



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

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



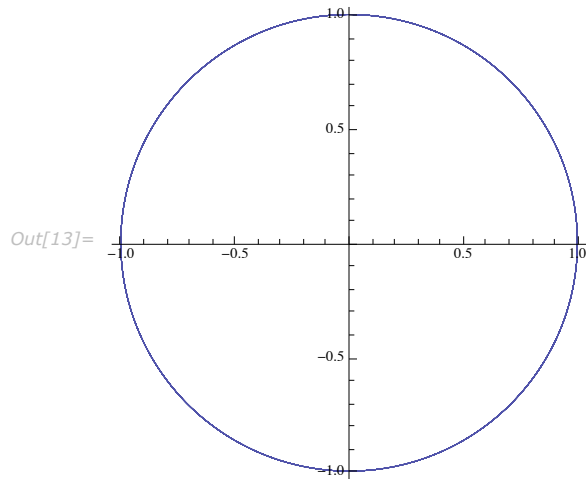
Symplectic Method

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];
```

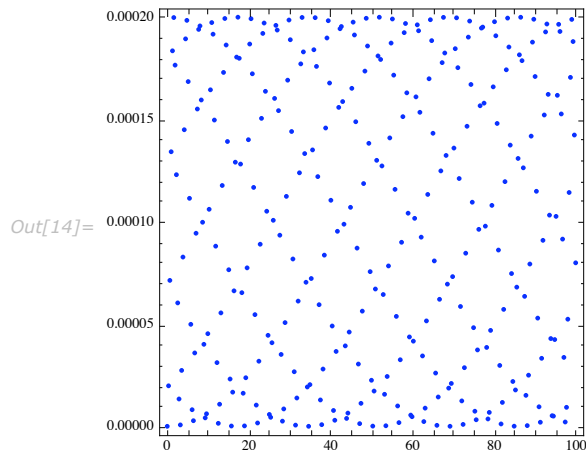
The solution is now a closed curve.

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



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]
```



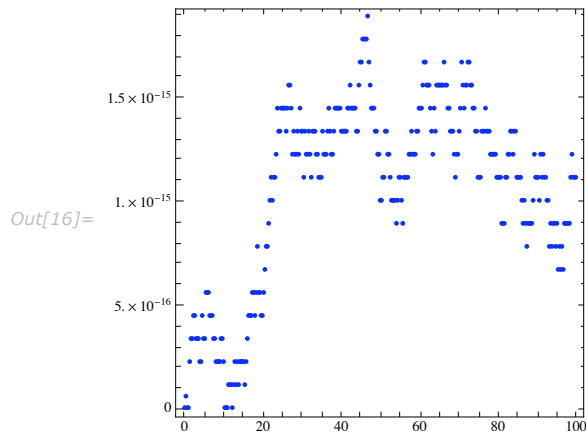
Rounding Error Reduction

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]
```

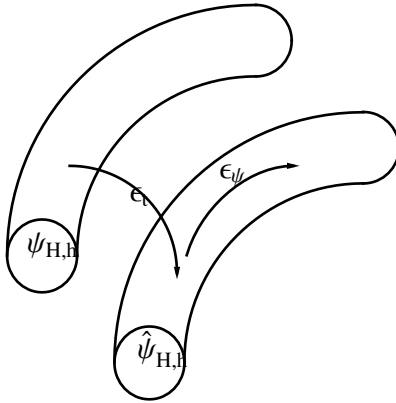


There is a curious drift in the error in the Hamiltonian that is actually a numerical artifact of floating-point arithmetic.

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 = 106;
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, ..., N
    Δyn = h f(yn) + yerr
    yn+1 = yn + Δyn
    yerr = (yn - yn+1) + Δyn
```

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 $i = 1, \dots, s$

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

$$\Delta Q_{i+1} = \Delta Q_i + h_{n+1} B_i g(p_n + \Delta P_i)$$

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:

$$k \in \sqrt{n}$$

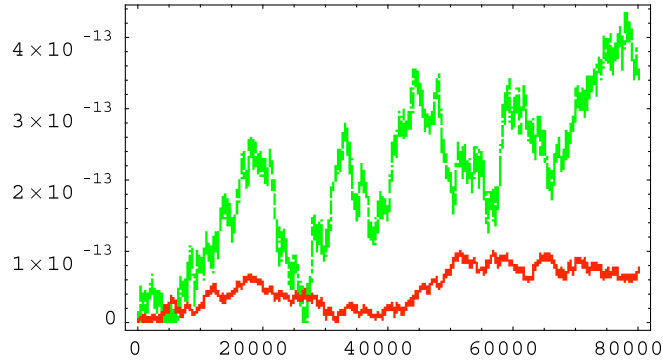
for some constant k .

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).



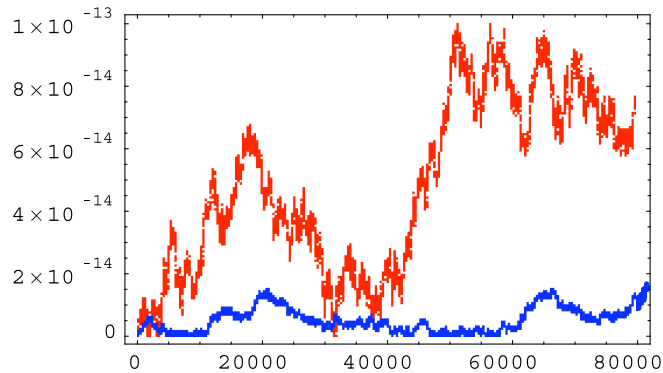
Algorithm 1 for a 15-stage method corresponds to $n = 15 \times 2 \times 10^6 = 3 \times 10^7$.

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

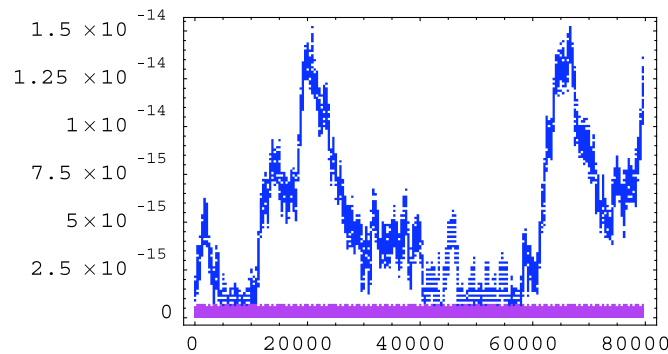
The relative error in the Hamiltonian is displayed here for the increment formulation in Algorithm 3 without compensated summation (red) and with compensated summation (blue) for the Harmonic oscillator (1).



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]).

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

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, 10 000 * 2 * Pi};
step = 2 * Pi / 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]}};

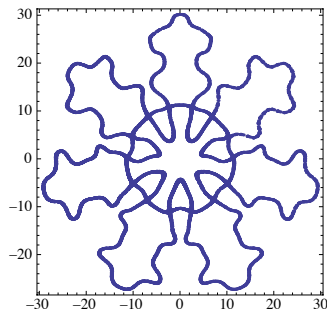
sprkdata =
  Block[{k = 1},
    Reap[
      NDSolve[{eqs, ics}, {}, time,
        Method → {"EventLocator", "Direction" → 1, "Event" → (t - 2 k Pi),
          "EventAction" → (k++; Sow[{q[t], p[t]}])}, Method → sprkmethod},
        StartingStepSize → step, MaxSteps → Infinity
      ];
    ]
  ];
```

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.

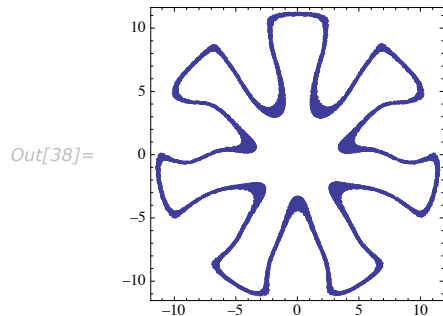
```
In[36]:= rkmethod = {"FixedStep", Method → {"ExplicitRungeKutta", "DifferenceOrder" → 4}};

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]
```



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

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

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 = {qn+1[t] → q1[t]};

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

eigenvalueRule = {ak_[t] → -pk[t] / 2, bk_[t] → 1 / 2 Exp[1 / 2 (qk+1[t] - qk[t])]};

L =  $\begin{pmatrix} a_1[t] & b_1[t] & b_3[t] \\ b_1[t] & a_2[t] & b_2[t] \\ b_3[t] & b_2[t] & a_3[t] \end{pmatrix}$  /. eigenvalueRule /. periodicRule;

eqs = {q1'[t] == D[H, p1[t]], q2'[t] == D[H, p2[t]], q3'[t] == D[H, p3[t]],
  p1'[t] == -D[H, q1[t]], p2'[t] == -D[H, q2[t]], p3'[t] == -D[H, q3[t]]};
ics = {q1[0] == 1, q2[0] == 2, q3[0] == 4, p1[0] == 0, p2[0] == 1, p3[0] == 1 / 2};
eqs = {eqs, ics};
vars = {q1[t], q2[t], q3[t], p1[t], p2[t], p3[t]};
time = {t, 0, 50};
```

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.

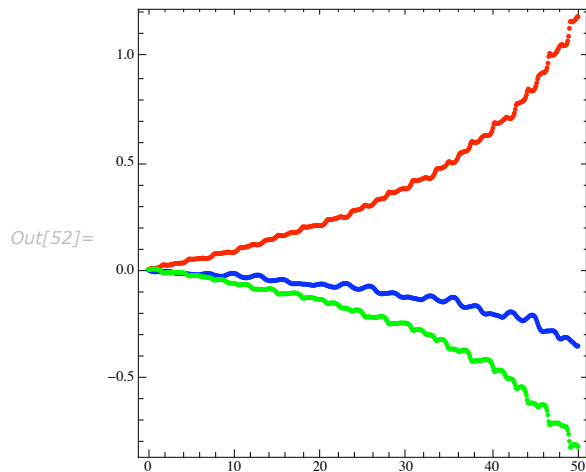
```
In[51]:= emsol =
  NDSolve[eqs, vars, time, Method → "ExplicitMidpoint", StartingStepSize → 1 / 10];
```

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}]
```

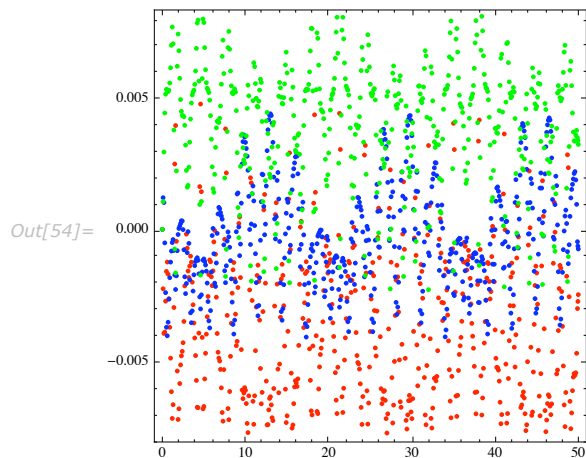


Integrate the equations for the Toda lattice using the “SymplecticPartitionedRungeKutta” method.

```
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}]
```



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.}}
```

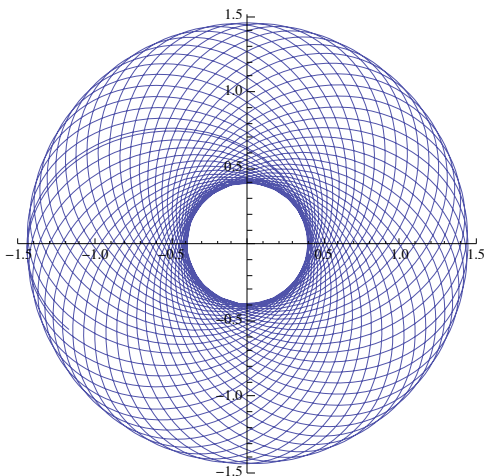
This invokes the symplectic partitioned Runge-Kutta solver using Yoshida's coefficients.

```
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]]
```



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 k^{th} 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  $k = 1, \dots, |\Pi|$ 
  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  $k = 1, \dots, |\Pi|$ 
  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:

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

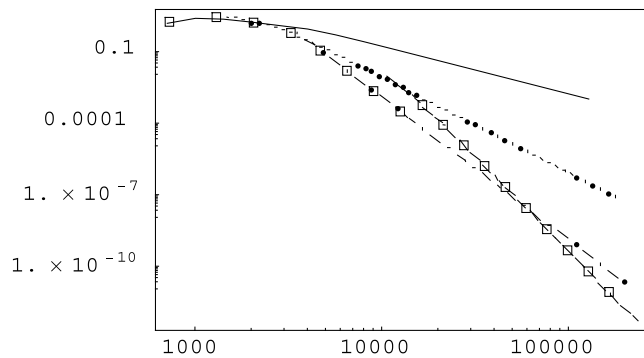
For initial conditions take

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

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.

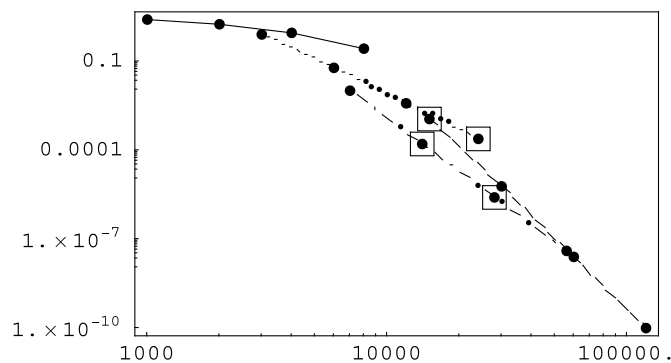


It can be observed that the algorithm does a reasonable job of staying near the optimal method, although it switches over to the 8th-order method slightly earlier than necessary.

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

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

$$\Psi_{f,h} = \Phi_{f,\gamma_s h}^{(s)} \circ \dots \circ \Phi_{f,\gamma_1 h}^{(1)}.$$

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

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

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:

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

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

$$\Psi_{f,h} = \Phi_{f_s,\gamma_s h}^{(s)} \circ \dots \circ \Phi_{f_1,\gamma_1 h}^{(1)},$$

with $f_1, \dots, f_s$ 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 $\Psi_{f,h} = \varphi_{f_2,h}^{(2)} \circ \varphi_{f_1,h}^{(1)}$ 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.

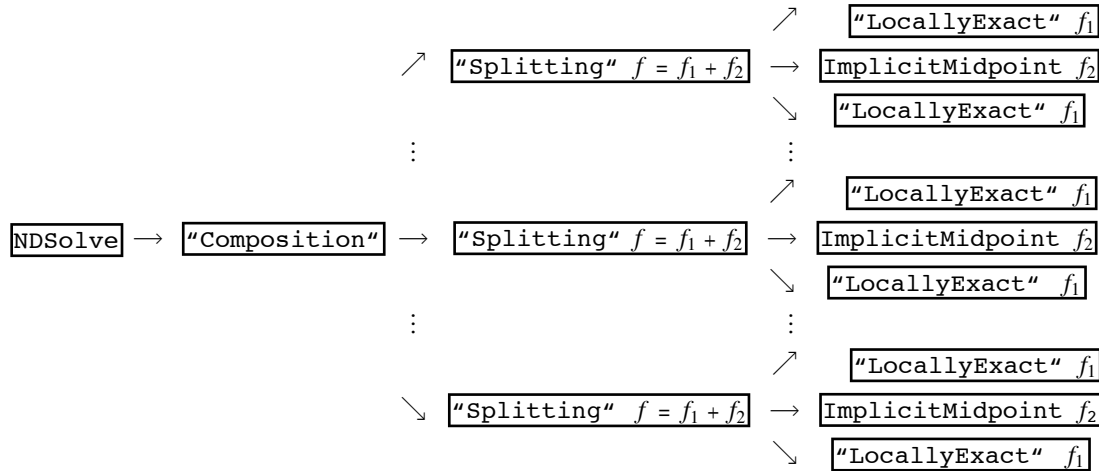
A simplified input syntax allows omitted vector fields and methods to be filled in cyclically. These must be defined unambiguously:

$\{f_1, f_2, f_1, f_2\}$ 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 $\{f_1, f_2, f_3, f_1, f_2\}$.

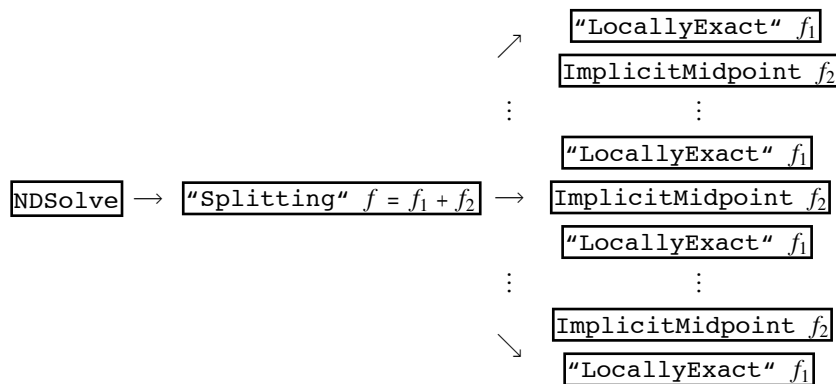
Nested Methods

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



Simplification

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



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]}, 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]= NDSolveProblem[{{Y1'[T] == -3/4 Cos[Y2[T]] + Sin[Y3[T]],
    Y2'[T] == Cos[Y3[T]] + Sin[Y1[T]], Y3'[T] == Cos[Y1[T]] + 3/4 Sin[Y2[T]]},
    {Y1[0] == 1/4, Y2[0] == 1/3, Y3[0] == 1/2}, {Y1[T], Y2[T], Y3[T]}, {T, 0, 100}, {}, {}]]

```

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]= {Y1'[T] == -3/4 Cos[Y2[T]] + Sin[Y3[T]], Y2'[T] == 0, Y3'[T] == Cos[Y1[T]] + 3/4 Sin[Y2[T]]}

```

```

In[46]:= Y2

```

```

Out[46]= {Y1'[T] == 0, Y2'[T] == Cos[Y3[T]] + Sin[Y1[T]], Y3'[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]= NDSolveProblem[{{Y1'[T] == Y2[T], Y2'[T] ==  $\frac{3 \cos[T]}{10} + Y_1[T] - Y_1[T]^3 + \frac{Y_2[T]}{4}$ },
    {Y1[0] == 0, Y2[0] == 1}, {Y1[T], Y2[T]}, {T, 0, 10}, {}, {}]}
```

This defines a splitting of the system.

```
In[50]:= Y1 = {Y1'[T] == Y2[T], Y2'[T] ==  $\frac{Y_2[T]}{4}$ };
Y2 = {Y1'[T] == 0, Y2'[T] ==  $\frac{3 \cos[T]}{10} + Y_1[T] - Y_1[T]^3$ };
```

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::spltdp:
  The differential system  $\left\{0, \frac{3 \cos[T]}{10} + Y_1[T] - Y_1[T]^3\right\}$  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.

```
In[53]:= Y1 = {Y1'[T] == Y2[T], Y2'[T] ==  $\frac{Y_2[T]}{4}$ , Z'[T] == 1};
Y2 = {Y1'[T] == 0, Y2'[T] ==  $\frac{3 \cos[Z[T]]}{10} + Y_1[T] - Y_1[T]^3$ , Z'[T] == 0};
eqs = Join[system["System"], {Z'[T] == 1}];
ics = Join[system["InitialConditions"], {Z[0] == 1}];
vars = Join[system["DependentVariables"], {Z[T]}];
time = system["TimeData"];
```

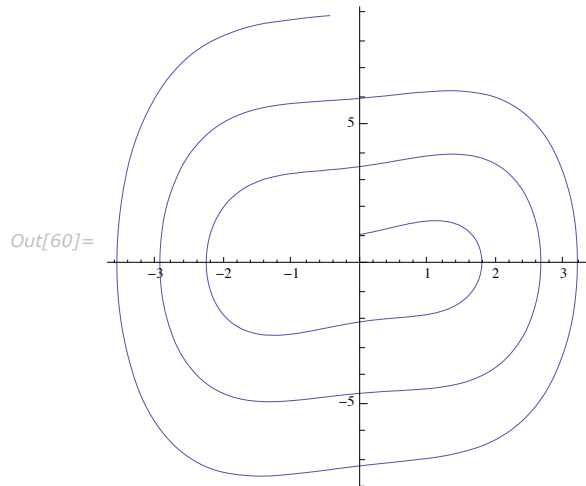
This defines a geometric splitting method that satisfies $\lambda_1 + \lambda_2 = -\delta$ for any finite time interval, where λ_1 and λ_2 are the Lyapunov exponents [MQ02].

```
In[53]:= splittingsol = NDSolve[{eqs, ics}, vars, time, StartingStepSize → 1 / 10,
  Method → {NDSolve`Splitting, "DifferenceOrder" → 2,
    "Equations" → {Y2, Y1, Y2}}, "Method" → {"LocallyExact"}]]

Out[53]= {{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]
```



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 $f_1, \dots, f_n$ such that $f = f_1 + \dots + f_n$.

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

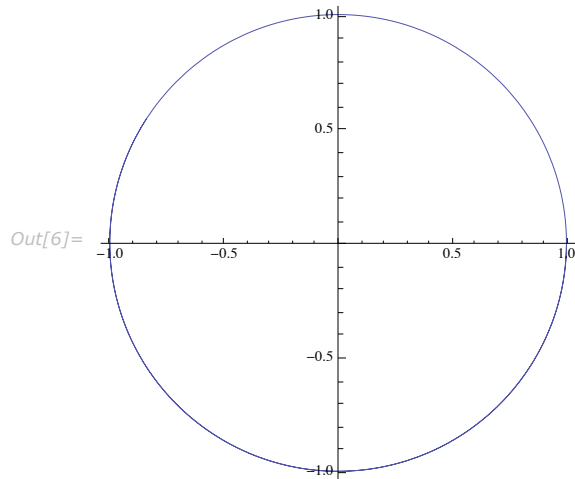
Numerically solve the equations of motion for a harmonic oscillator using the method "LocallyExact". The result is two interpolating functions that approximate the solution and the first derivative.

```
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]
```



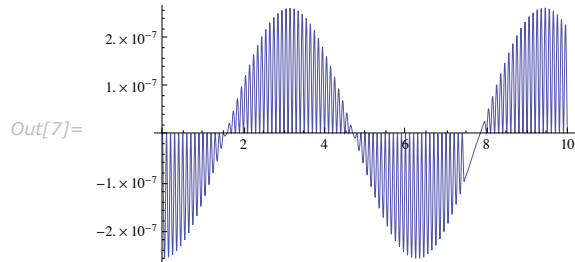
Global versus Local

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]]
```



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]= {{Y1[T] →
```

$$\frac{1}{2402} \left(1201 e^{\frac{1}{2}(-11-\sqrt{1201})T} + 9\sqrt{1201} e^{\frac{1}{2}(-11-\sqrt{1201})T} + 1201 e^{\frac{1}{2}(-11+\sqrt{1201})T} - 9\sqrt{1201} e^{\frac{1}{2}(-11+\sqrt{1201})T} \right)$$

$$10 \left(e^{\frac{1}{2}(-11-\sqrt{1201})T} - e^{\frac{1}{2}(-11+\sqrt{1201})T} \right) C[2]$$

$$C[1] - \frac{\sqrt{1201}}{28 \left(e^{\frac{1}{2}(-11-\sqrt{1201})T} - e^{\frac{1}{2}(-11+\sqrt{1201})T} \right) C[1]}$$

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

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[{simp},
Print["Before simplification ", x];
simp = FullSimplify[ExpToTrig[x]];
Print["After simplification ", simp];
simp
];
```

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

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

After simplification

$$\begin{aligned} & \left\{ \frac{1}{1201} e^{-11T/2} \left(1201 \cosh\left[\frac{\sqrt{1201}T}{2}\right] Y_1[T] + \sqrt{1201} \sinh\left[\frac{\sqrt{1201}T}{2}\right] \right. \right. \\ & \quad \left. \left. (-9 Y_1[T] + 20 Y_2[T]) \right), e^{-11T/2} \cosh\left[\frac{\sqrt{1201}T}{2}\right] Y_2[T] + \right. \\ & \quad \left. \frac{e^{-11T/2} \sinh\left[\frac{\sqrt{1201}T}{2}\right] (56 Y_1[T] + 9 Y_2[T])}{\sqrt{1201}}, e^{-8T/3} Y_3[T] \right\} \end{aligned}$$

```
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:

$$e = \frac{y_2 - y_1}{2^p - 1}. \quad (1)$$

- 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:

$$\hat{y}_2 = y_2 + e. \quad (2)$$

- 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".

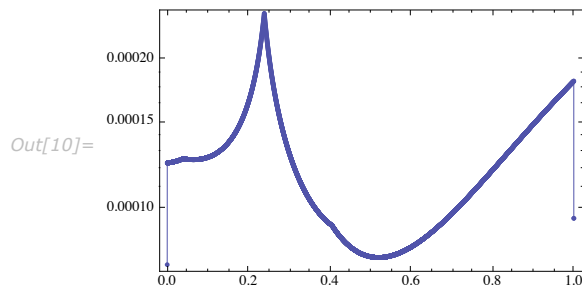
The local error estimate (1) is used to dynamically adjust the step size throughout the integration.

```
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]
```



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[ $\frac{z^i}{i!}$ , {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 #12 + #13 &, 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 $\text{low} \leq h_{n+1}/h_n \leq \text{high}$
"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.

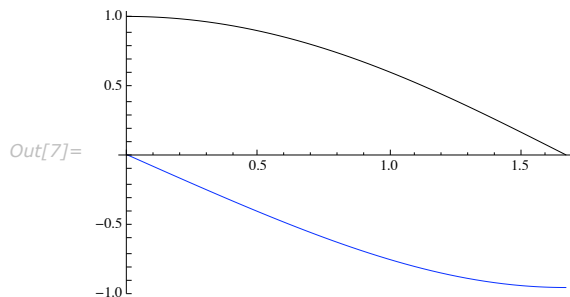
This integrates the pendulum equation up to the first point at which the solution $y[t]$ crosses the axis.

```
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}}]
```



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 ($\rightarrow$) 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., 620.015.}}, <>]}}
```

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.

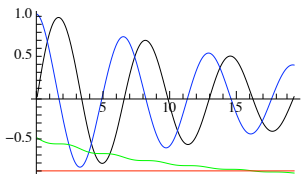
The “EventCondition” option of “EventLocator” allows you to specify additional Boolean conditions that need to be satisfied for an event to be tested. It is advantageous to use this instead of a Boolean event when possible because the root finding process can be done more efficiently.

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 threshold 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:

$$y_e = k_n y_n + k_{n+1} y_{n+1} + l_n f_n + l_{n+1} f_{n+1}$$

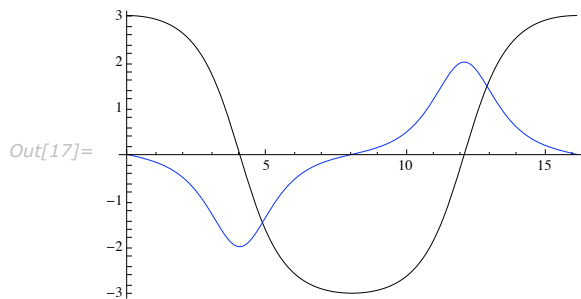
for suitably defined interpolation weights:

$$\begin{aligned} k_n &= (w_e - 1)^2 (2w_e + 1) \\ k_{n+1} &= (3 - 2w_e) w_e^2 \\ l_n &= h_n (w_e - 1)^2 w_e \\ l_{n+1} &= h_n (w_e - 1) w_e^2 \end{aligned}$$

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.

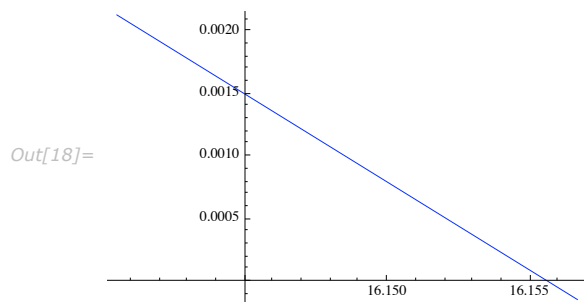
```
In[16]:= sol = First[NDSolve[{y''[t] + Sin[y[t]] == 0, y[0] == 3, y'[0] == 0},
  y, {t, 0, ∞}, Method -> {"EventLocator",
    "Event" -> y'[t],
    "EventAction" -> Throw[end == t, "StopIntegration"], "Direction" -> -1,
    "EventLocationMethod" -> "LinearInterpolation",
    Method -> "ExplicitRungeKutta"}]];
Plot[Evaluate[{y[t], y'[t]} /. sol], {t, 0, end}, PlotStyle -> {{Black}, {Blue}}]
```



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]
```



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"]}]]

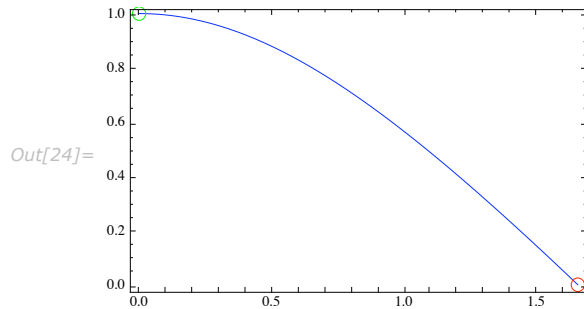
Out[21]= InterpolatingFunction[{{0., 1.65745}}, <>][t]
```

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]
```



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[ $\left(\begin{array}{l} y_1'[t] = y_2[t] \\ y_2'[t] = \mu(1 - y_1[t]^2)y_2[t] - y_1[t] \end{array}\right)$ ,
    {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"]}

Out[29]= {{(Y1)'[T] == Y3[T], (Y2)'[T] == Y4[T], (Y3)'[T] == -Y1[T] (1 + 2 Y2[T]),
  (Y4)'[T] == -Y1[T]^2 + (-1 + Y2[T]) Y2[T]}, {Y1[0] ==  $\frac{3}{25}$ , Y2[0] ==  $\frac{3}{25}$ , Y3[0] ==  $\frac{3}{25}$ , Y4[0] ==  $\frac{3}{25}$ }}
```

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.

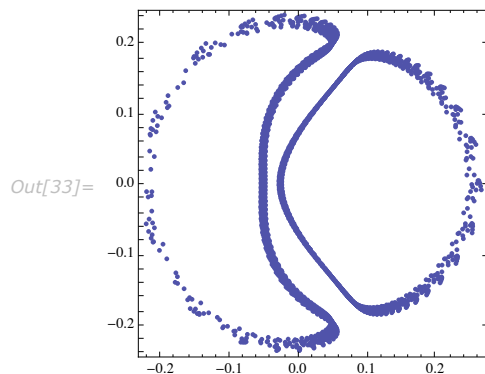
```
In[30]:= Off[NDSolve::noout];
```

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]
```



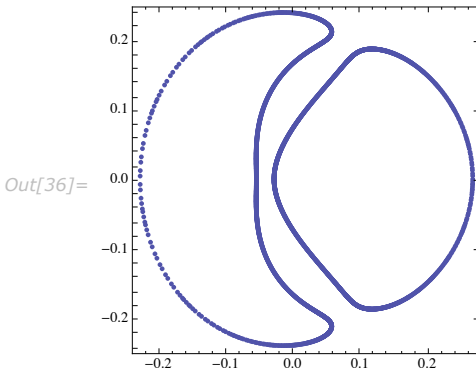
Since the Hénon-Heiles system is Hamiltonian, a symplectic method gives much better qualitative results for this example.

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]
```



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]= {{(Y1)'[T] == 3/4 Cos[Y2[T]] + Sin[Y3[T]], (Y2)'[T] == 0, (Y3)'[T] == Cos[Y1[T]] + 3/4 Sin[Y2[T]]}}
```

```
In[42]:= Y2 = eqs; Y2[{{1, 3}, 2}] = 0; Y2
```

```
Out[42]= {{(Y1)'[T] == 0, (Y2)'[T] == Cos[Y3[T]] + Sin[Y1[T]], (Y3)'[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.

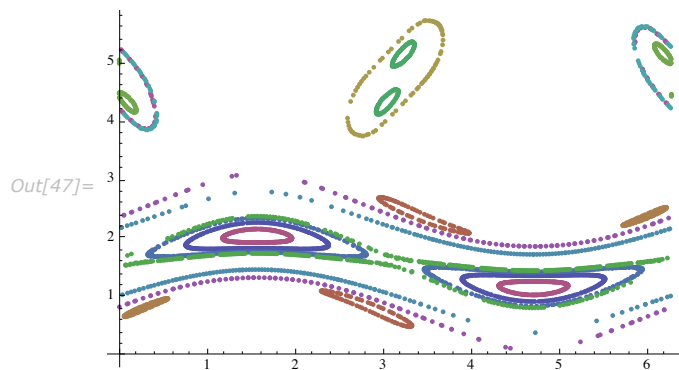
```
In[44]:= ABCSplitting = {"Splitting",
  "DifferenceOrder" → 2,
  "Equations" → {Y2, Y1, Y2},
  "Method" → {"LocallyExact", ImplicitMidpoint, "LocallyExact"}};
```

This defines a function that gives the Poincaré section for a particular initial condition.

```
In[45]:= pssect[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  $\pi$ ];
ListPlot[data, ImageSize  $\rightarrow$  Medium]
```



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, xpl, y1, ypl, 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,  $\infty$ }, Method  $\rightarrow$  {"EventLocator", "Event"  $\rightarrow$  y[t] - ramp[x[t]]},
       MaxStepSize  $\rightarrow$  0.01}];
    t1 = InterpolatingFunctionDomain[x /. sol][[1, -1]];
    {x1, xpl, y1, ypl} =
      Reflection[k, ramp][{x[t1], x'[t1], y[t1], y'[t1]} /. sol];
    Sow[{x[t] /. sol, t0  $\leq$  t  $\leq$  t1}, "X"];
    Sow[{y[t] /. sol, t0  $\leq$  t  $\leq$  t1}, "Y"];
    Sow[{x1, y1}, "Bounces"];
    {t1, x1, xpl, y1, ypl}]
```


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.

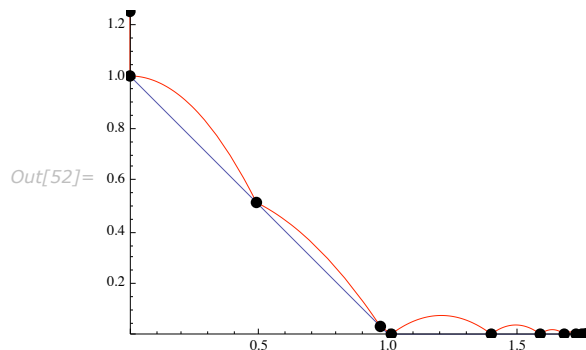
```
In[49]:= Reflection[k_, ramp_][{x_, xp_, y_, yp_}] := Module[{gramp, gp, xpnew, ypnew},
  gramp = -ramp'[x];
  If[Not[NumberQ[gramp]],
    Print["Could not compute derivative "];
    Throw[$Failed]];
  gramp = {-ramp'[x], 1};
  If[gramp.{xp, yp} == 0,
    Print["No reflection"];
    Throw[$Failed]];
  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]]]]
```

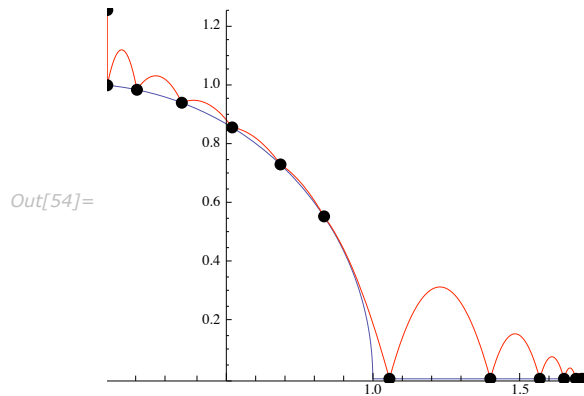
This is the example that is done in [SGT03].

```
In[51]:= ramp[x_] := If[x < 1, 1 - x, 0];
BouncingBall[.7, ramp, {0, 1.25}]
```



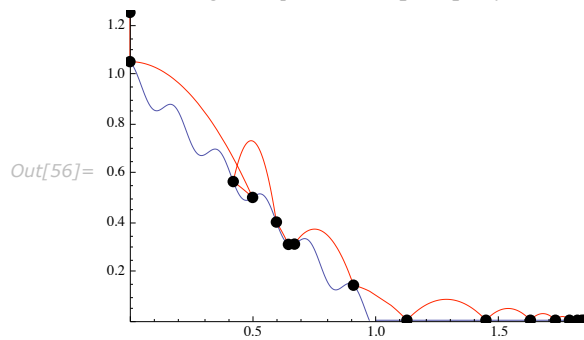
The ramp is now defined to be a quarter circle.

```
In[53]:= circle[x_] := If[x < 1, Sqrt[1 - x^2], 0];  
BouncingBall[.7, circle, {.1, 1.25}]
```



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}]
```



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]:= 
$$\mu = \frac{1}{82.45};$$


$$\mu^* = 1 - \mu;$$


$$r_1 = \sqrt{(y_1[t] + \mu)^2 + y_2[t]^2};$$


$$r_2 = \sqrt{(y_1[t] - \mu^*)^2 + y_2[t]^2};$$


$$\text{eqns} = \left\{ \left\{ y_1'[t] == y_3[t], y_1[0] == 1.2 \right\}, \left\{ y_2'[t] == y_4[t], y_2[0] == 0 \right\}, \right.$$


$$\left. \left\{ y_3'[t] == 2 y_4[t] + y_1[t] - \frac{\mu^* (y_1[t] + \mu)}{r_1^3} - \frac{\mu (y_1[t] - \mu^*)}{r_2^3}, y_3[0] == 0 \right\}, \right.$$


$$\left. \left\{ y_4'[t] == -2 y_3[t] + y_2[t] - \frac{\mu^* y_2[t]}{r_1^3} - \frac{\mu y_2[t]}{r_2^3}, \right. \right.$$


$$\left. y_4[0] == -1.04935750983031990726 \cdot 10^{20.020923474937767} \right\};$$

```

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`.

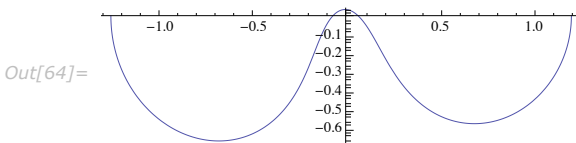
```
In[63]:= sol = First[NDSolve[eqns, {y1, y2, y3, y4}, {t, 0, \infty},
  Method -> {"EventLocator",
    "Event" -> {ddist, ddist},
    "Direction" -> {1, -1},
    "EventAction" -> {Throw[tfinal = t, "StopIntegration"], tfar = t},
    Method -> "ExplicitRungeKutta"}]]

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.

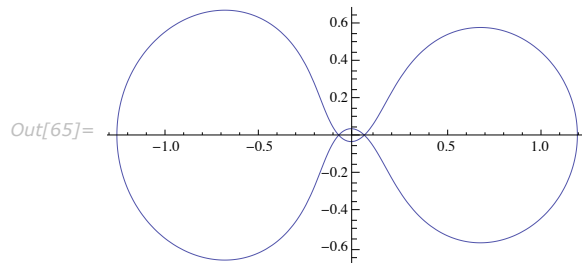
This displays one half-orbit when the spaceship is at the furthest point from the initial position.

```
In[64]:= ParametricPlot[{y1[t], y2[t]} /. sol, {t, 0, tfar}]
```



This displays one complete orbit when the spaceship returns to the initial position.

```
In[65]:= ParametricPlot[{y1[t], y2[t]} /. sol, {t, 0, tfinal}]
```



Delay Differential Equation

The following system models an infectious disease (see [HNW93], [ST00] and [ST01]).

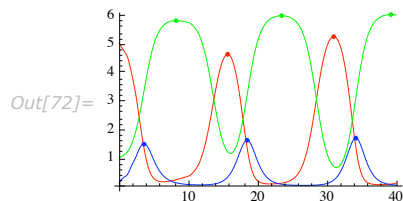
```
In[66]:= system = {y1'[t] == -y1[t] y2[t - 1] + y2[t - 10],
  y1[t /; t <= 0] == 5, y2'[t] == y1[t] y2[t - 1] - y2[t],
  y2[t /; t <= 0] == 1 / 10, y3'[t] == y2[t] - y2[t - 10], y3[t /; t <= 0] == 1};
vars = {y1[t], y2[t], y3[t]};
```

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]
```



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 :

$$y' = \begin{cases} f_I(t, y) & \text{if } g(t, y) > 0 \\ f_{II}(t, y) & \text{if } g(t, y) < 0 \end{cases}$$

In order to illustrate the difficulty in crossing a discontinuity, consider the following example [GØ84] (see also [HNW93]):

$$y' = \begin{cases} t^2 + 2y^2 & \text{if } \left(t + \frac{1}{20}\right)^2 + \left(y + \frac{3}{20}\right)^2 \leq 1 \\ 2t^2 + 3y[t]^2 - 2 & \text{if } \left(t + \frac{1}{20}\right)^2 + \left(y + \frac{3}{20}\right)^2 > 1 \end{cases}$$

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;
```

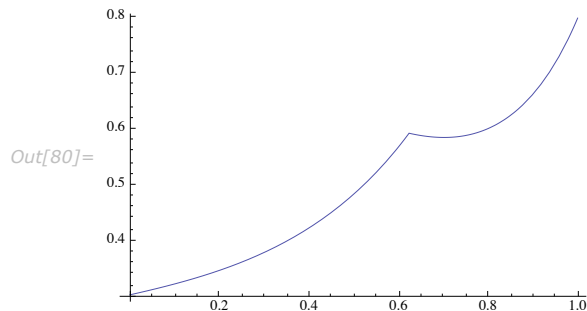
This solves the system on the interval $[0, 1]$ and collects data for the mesh points of the integration using `Reap` and `Sow`.

```
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}]
```



Despite the fact that a solution has been obtained, it is not clear whether it has been obtained efficiently.

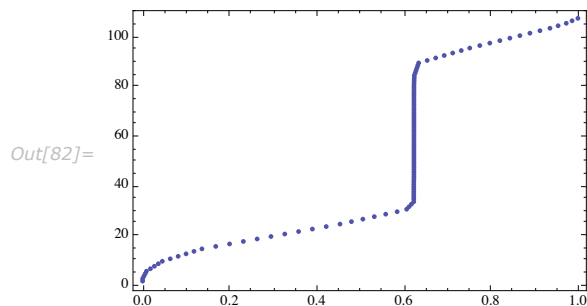
The following example shows that the crossing of the discontinuity presents difficulties for the numerical solver.

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]
```



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.

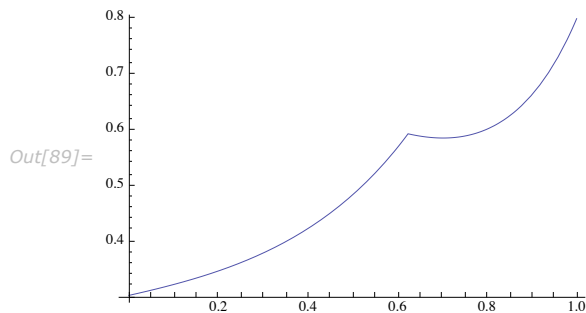
```
In[86]:= system2 = {y'[t] == 2 t^2 + 3 y[t]^2 - 2, y[t1] == ics1};

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]
```

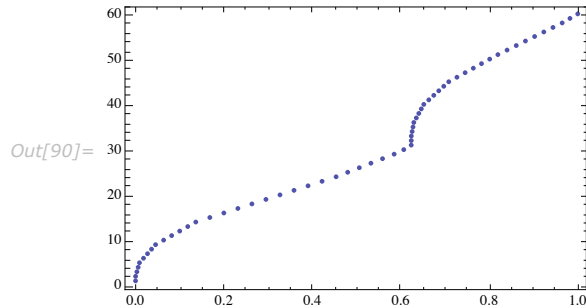
A plot of the two solutions is very similar to that obtained by solving the entire system at once.

```
In[89]:= evsol = Plot[If[t ≤ t1, sol1, sol2], {t, 0, 1}]
```



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]]
```



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]]]
```

$$\text{Out[91]} = \left\{ y[t] \rightarrow \left(t \left(3 \text{BesselJ}\left[-\frac{3}{4}, \frac{t^2}{\sqrt{2}}\right] \text{Gamma}\left[\frac{1}{4}\right] + 10 \times 2^{1/4} \text{BesselJ}\left[\frac{3}{4}, \frac{t^2}{\sqrt{2}}\right] \text{Gamma}\left[\frac{3}{4}\right] \right) \right) / \left(\sqrt{2} \left(-3 \text{BesselJ}\left[\frac{1}{4}, \frac{t^2}{\sqrt{2}}\right] \text{Gamma}\left[\frac{1}{4}\right] + 10 \times 2^{1/4} \text{BesselJ}\left[-\frac{1}{4}, \frac{t^2}{\sqrt{2}}\right] \text{Gamma}\left[\frac{3}{4}\right] \right) \right) \right\}$$

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.

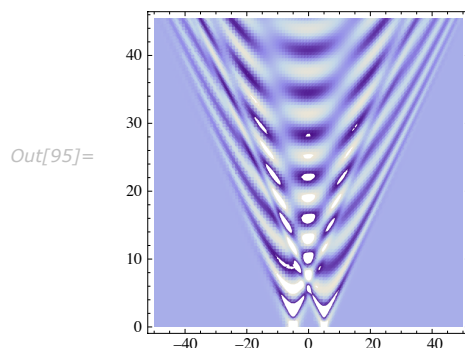
The sine-Gordon equation turns up in differential geometry and relativistic field theory. This example integrates the equation, starting with a localized initial condition that spreads out. The periodic pseudospectral method is used for the integration. Since no absorbing layer has been instituted near 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[{ $\partial_{t,t}u[t, x] == \partial_{x,x}u[t, x] - \text{Sin}[u[t, x]]$ ,
    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]
```



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[{ $\partial_{t,x} u[t, x] == \partial_{x,x} u[t, x] - \text{Sin}[u[t, x]]$ ,
  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] +  $\frac{1}{1000} y'[t] + \text{Sin}[y[t]] == 0$ , y[0] == 3, y'[0] == 0},
      y, {t,  $\infty$ }, Method -> {"EventLocator", "Event" -> y'[t],
        "EventAction" -> If[p++ >= 1000, Throw[end = t, "StopIntegration"]],
        "EventLocationMethod" -> locmethod, "Method" -> odemethod},
      MaxSteps ->  $\infty$ }]
  ];
```

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] +  $\frac{1}{1000}$  y'[t] + Sin[y[t]] == 0,
        y[0] == 3, y'[0] == 0}, y, {t, end}, Method -> #, MaxSteps ->  $\infty$ ]]]] &,
    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[pi 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[pi 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[pi y[t]], t], z[-1] == Sin[pi e^-1]},
    {y, z}, {t, 10}, Method -> {"EventLocator", "Event" -> z[t], "EventAction" -> n++},
    MaxSteps -> Infinity]; 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

$$y'(t) = f(t, y(t)), y(t_0) = y_0. \quad (1)$$

Let $H > 0$ be a basic step size; choose a monotonically increasing sequence of positive integers

$$n_1 < n_2 < n_3 < \dots < n_k$$

and define the corresponding step sizes

$$h_1 > h_2 > h_3 > \dots > h_k$$

by

$$h_i = \frac{H}{n_i}, i = 1, 2, \dots, k.$$

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:

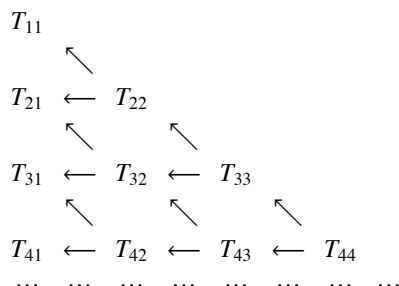
$$T_{i,1} = y_{h_i}(t_0 + H), i = 1, 2, \dots, k.$$

Extrapolation is performed using the Aitken-Neville algorithm by building up a table of values:

$$T_{i,j} = T_{i,j-1} + \frac{T_{i,j-1} - T_{i-1,j-1}}{\left(\frac{n_i}{n_{i-j+1}}\right)^w - 1}, i = 2, \dots, k, j = 2, \dots, i, \quad (2)$$

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:



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 $(n_i / n_{i-j+1})^w \geq 2$ 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 $T_{11}, T_{21}, T_{31}, \dots$ are disturbed with rounding errors $\epsilon, -\epsilon, \epsilon, \dots$ 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:

$$\begin{aligned} y_1 &= y_0 + h f(t_0, y_0) \\ y_2 &= y_1 + h f(t_1, y_1) \\ y_3 &= y_2 + h f(t_2, y_2) \\ &\vdots \\ y_n &= y_{n-1} + h f(t_{n-1}, y_{n-1}) \end{aligned} \tag{1}$$

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:

$$\begin{aligned} k_i &= f(t_0 + c_i H, y_0 + H \sum_{j=1}^n a_{i,j} k_j), \quad i = 1, \dots, n, \\ y_n &= y_0 + H \sum_{i=1}^n b_i k_i \end{aligned} \tag{1}$$

where the coefficients are represented by the Butcher table:

$$\begin{array}{c|ccc} 0 & & & \\ 1/n & 1/n & & \\ \vdots & \vdots & \ddots & \\ (n-1)/n & 1/n & \cdots & 1/n \\ \hline & 1/n & \cdots & 1/n & 1/n \end{array} \tag{2}$$

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:

$$\begin{aligned}
 \Delta y_0 &= h f(t_0, y_0) \\
 \Delta y_1 &= h f(t_1, y_0 + \Delta y_0) \\
 \Delta y_2 &= h f(t_2, y_0 + (\Delta y_0 + \Delta y_1)) \\
 \vdots &\quad \quad \quad \vdots \\
 \Delta y_{n-1} &= h f(t_{n-1}, y_0 + (\Delta y_0 + \Delta y_1 + \cdots + \Delta y_{n-2}))
 \end{aligned} \tag{1}$$

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 $n = 2n_k$ with step size $h = H/n$ carried out using one Euler step followed by multiple explicit midpoint steps:

$$\begin{aligned}
 y_1 &= y_0 + h f(t_0, y_0) \\
 y_2 &= y_0 + 2 h f(t_1, y_1) \\
 y_3 &= y_1 + 2 h f(t_2, y_2) \\
 \vdots &\quad \quad \quad \vdots \\
 y_n &= y_{n-2} + 2 h f(t_{n-1}, y_{n-1})
 \end{aligned} \tag{1}$$

If (1) is computed with $2n_k - 1$ midpoint steps, then the method has a symmetric error expansion ([G65], [S70]).

Reformulation

Reformulation of (1) can be accomplished in terms of increments as:

$$\begin{aligned}
 \Delta y_0 &= h f(t_0, y_0) \\
 \Delta y_1 &= 2 h f(t_1, y_0 + \Delta y_0) - \Delta y_0 \\
 \Delta y_2 &= 2 h f(t_2, y_0 + (\Delta y_0 + \Delta y_1)) - \Delta y_1 \\
 \vdots &\quad \quad \quad \vdots \\
 \Delta y_{n-1} &= 2 h f(t_{n-1}, y_0 + (\Delta y_0 + \Delta y_1 + \cdots + \Delta y_{n-2})) - \Delta y_{n-2}
 \end{aligned} \tag{1}$$

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 $2n_k$ 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.

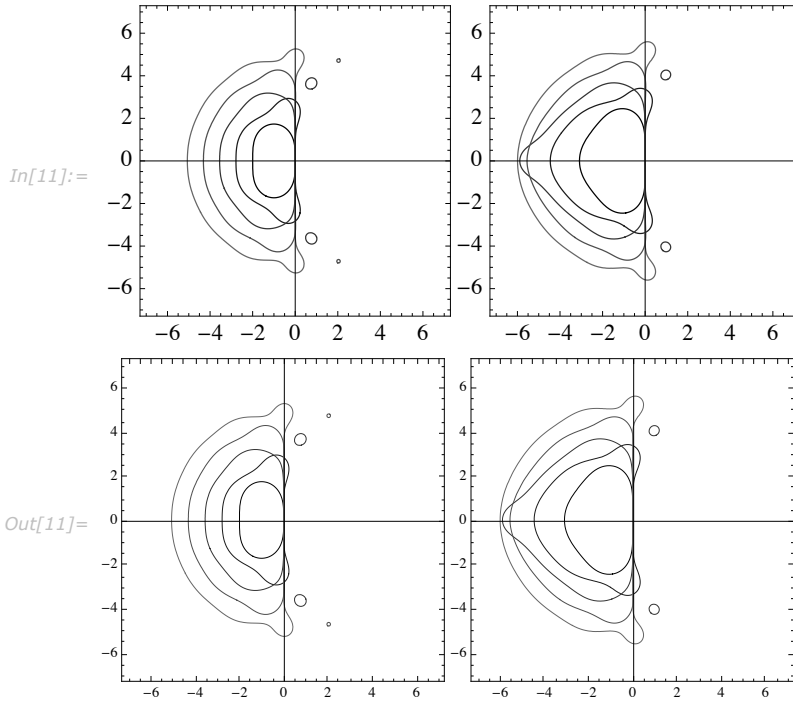
Gragg's smoothing step is not of great importance if the method is followed by extrapolation, and Shampine proposes an alternative smoothing procedure that is slightly more efficient [SB83].

The method "ExplicitMidpoint" uses $2n_k - 1$ steps and "ExplicitModifiedMidpoint" uses $2n_k$ 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 $T_{i,i}$, $i = 1, \dots, 5$ for the explicit midpoint rule (left) and the explicit midpoint rule with smoothing (right).



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 stability 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:

$$M y'(t) = f(t, y(t)), y(t_0) = y_0. \quad (1)$$

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 $h = H/n$.

$$\begin{aligned} (M - hJ) \Delta y_0 &= h f(t_0, y_0) \\ y_1 &= y_0 + \Delta y_0 \\ (M - hJ) \Delta y_1 &= h f(t_1, y_1) \\ y_2 &= y_1 + \Delta y_1 \\ (M - hJ) \Delta y_2 &= h f(t_2, y_2) \\ y_3 &= y_2 + \Delta y_2 \\ \vdots &\quad \quad \quad \vdots \\ (M - hJ) \Delta y_{n-1} &= h f(t_{n-1}, y_{n-1}) \end{aligned} \quad (1)$$

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 $M - hJ$ 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:

$$\begin{aligned}
 (M - hJ) \Delta y_0 &= h f(t_0, y_0) \\
 (M - hJ) \Delta y_1 &= h f(t_1, y_0 + \Delta y_0) \\
 (M - hJ) \Delta y_2 &= h f(t_2, y_0 + (\Delta y_0 + \Delta y_1)) \\
 \vdots &\quad \quad \quad \vdots \\
 (M - hJ) \Delta y_{n-1} &= h f(t_{n-1}, y_0 + (\Delta y_0 + \Delta y_1 + \cdots + \Delta y_{n-2}))
 \end{aligned} \tag{1}$$

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 $n = 2n_k$ and $h = H/n$, using the formulation of Bader and Deuflhard [BD83]:

$$\begin{aligned}
 (M - hJ) \Delta y_0 &= h f(t_0, y_0) \\
 y_1 &= y_0 + \Delta y_0 \\
 (M - hJ) (\Delta y_1 - \Delta y_0) &= 2 (h f(t_1, y_1) - \Delta y_0) \\
 y_2 &= y_1 + \Delta y_1 \\
 (M - hJ) (\Delta y_2 - \Delta y_1) &= 2 (h f(t_2, y_2) - \Delta y_1) \\
 y_3 &= y_2 + \Delta y_2 \\
 \vdots &\quad \quad \quad \vdots \\
 (M - hJ) (\Delta y_{n-1} - \Delta y_{n-2}) &= 2 (h f(t_{n-1}, y_{n-1}) - \Delta y_{n-2})
 \end{aligned} \tag{1}$$

If (1) is computed for $2n_k - 1$ 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:

$$\begin{aligned}
 (M - hJ) \Delta y_0 &= h f(t_0, y_0) \\
 (M - hJ) (\Delta y_1 - \Delta y_0) &= 2 (h f(t_1, y_0 + \Delta y_0) - \Delta y_0) \\
 (M - hJ) (\Delta y_2 - \Delta y_1) &= 2 (h f(t_2, y_0 + (\Delta y_0 + \Delta y_1)) - \Delta y_1) \\
 \vdots &\quad \quad \quad \vdots \\
 (M - hJ) (\Delta y_{n-1} - \Delta y_{n-2}) &= 2 (h f(t_{n-1}, y_0 + (\Delta y_0 + \Delta y_1 + \cdots + \Delta y_{n-2})) - \Delta y_{n-2})
 \end{aligned} \tag{1}$$

Smoothing Step

An appropriate smoothing step for the linearly implicit midpoint rule is [BD83]:

$$S y_h(n) = \frac{1}{2} (y_{n-1} + y_{n+1}). \quad (1)$$

Bader's smoothing step (1) rewritten in terms of increments becomes:

$$S \Delta y_h(n) = S y_h(n) - y_0 = \Sigma \Delta y_n + \frac{1}{2} (\Delta y_n - \Delta y_{n-1}). \quad (2)$$

The required quantities are obtained when (1) is run with $2n_k$ 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 $2n_k - 1$ steps and "LinearlyImplicitModifiedMidpoint" uses $2n_k$ 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:

$$\Delta T_{i,j} = \Delta T_{i,j-1} + \frac{\Delta T_{i,j-1} - \Delta T_{i-1,j-1}}{\left(\frac{n_i}{n_{i-j+1}}\right)^p - 1}, \quad i = 2, \dots, k, \quad j = 2, \dots, i. \quad (1)$$

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 $err_j = \|T_{j,j-1} - T_{j,j}\|$.

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:

$$\begin{aligned} (M - hJ) \Delta_0 &= h f(t_0, y_0) \\ (M - hJ) \Delta_1 &= h f(t_0, y_0 + \Delta_0) - \Delta_0 \end{aligned} \tag{1}$$

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].

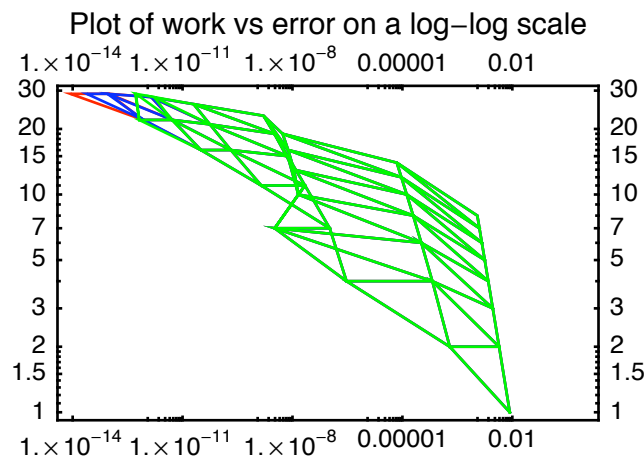
```
In[12]:= t0 =  $\pi / 6$ ;
h0 = 1 / 10;
y0 = {2 /  $\sqrt{3}$ };
eqs = {y'[t] == (-y[t] Sin[t] + 2 Tan[t]) y[t], y[t0] == y0};
exactsol = y[t] /. First[DSolve[eqs, y[t], t]] /. t -> t0 + h0;
idata = {{eqs, y[t], t}, h0, exactsol};
```

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

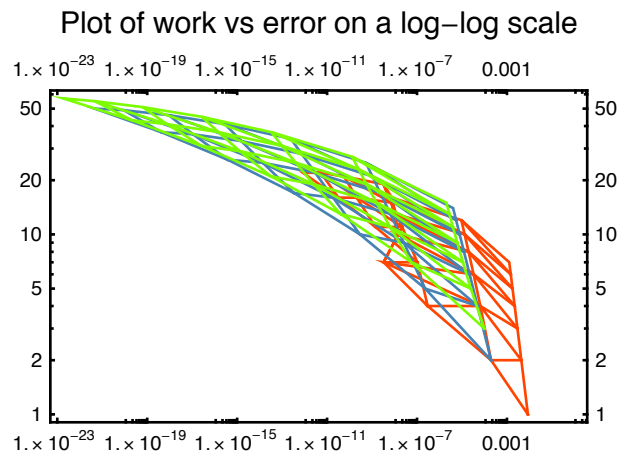
Notice that the rounding-error model that was used to motivate the study of rounding-error growth is limited because in practice, errors in $T_{i,1}$ can exceed 1 ULP.

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.

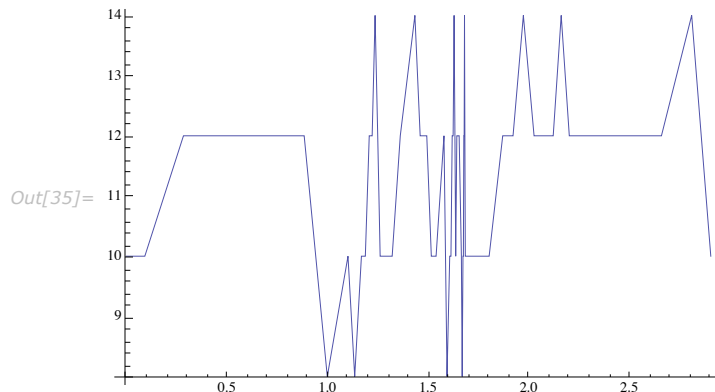
```
In[33]:= OrderMonitor[t_, method_NDSolve`Extrapolation] :=  
  Sow[{t, method["DifferenceOrder"]}];
```

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]
```



Method Comparison

Select the problem to solve.

```
In[67]:= system = GetNDSolveProblem["Arenstorff"];
```

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.

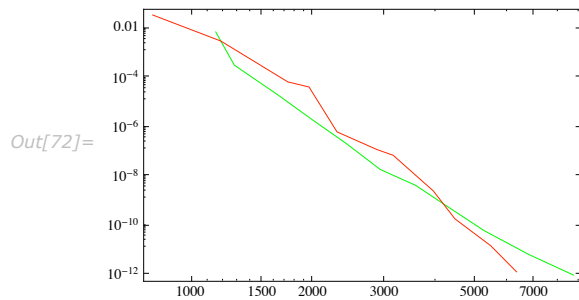
```
In[70]:= methods = {{"ExplicitRungeKutta", "StiffnessTest" → False}, {"Extrapolation",  
Method → "ExplicitModifiedMidpoint", "StiffnessTest" → False}};
```

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}}]
```



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]:= vdpso1 = 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]}
```

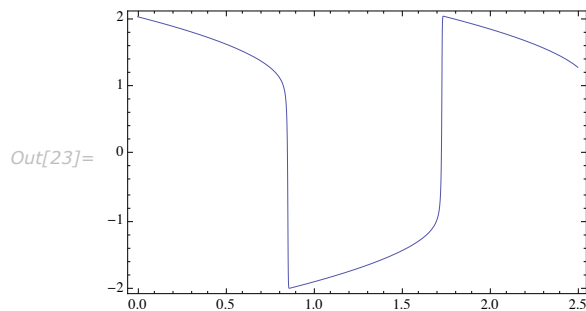
This solves the equations using “Extrapolation” with the “LinearlyImplicitEuler” base method with the default sub-harmonic sequence 2, 3, 4,

```
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.

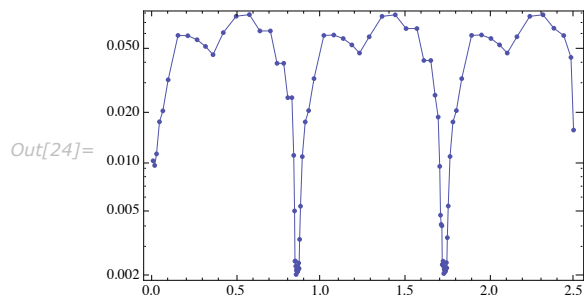
This plots the first solution component over time.

```
In[23]:= Plot[Evaluate[First[vdpsol]], Evaluate[time], Frame → True, Axes → False]
```



This plots the step sizes taken in computing the solution.

```
In[24]:= StepDataPlot[vdpsol]
```



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.

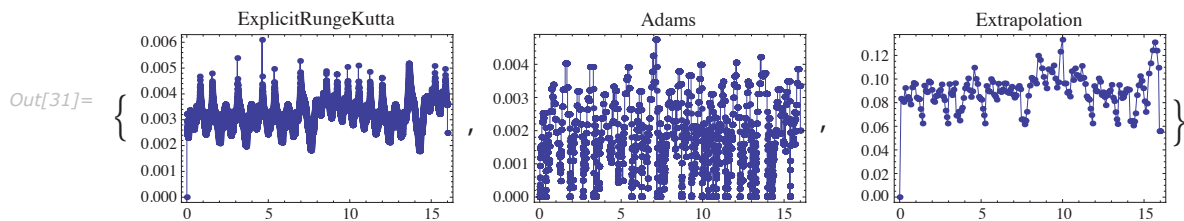
```
In[27]:= Timing[
  adamssol = NDSolve[system, Method -> "Adams", WorkingPrecision -> 32];
]
Out[27]:= {1.81172, Null}
```

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}]
```



Mass Matrix - fem2ex

Consider the partial differential equation:

$$\frac{\partial u}{\partial t} = \exp(t) \frac{\partial^2 u}{\partial x^2}, \quad u(0, x) = \sin(x), \quad u(t, 0) = u(t, \pi) = 0. \quad (1)$$

Given an integer n define $h = \pi/(n+1)$ and approximate at $x_k = k h$ with $k = 0, \dots, n+1$ using the Galerkin discretization:

$$u(t, x_k) \approx \sum_{k=1}^n c_k(t) \phi_k(x) \quad (2)$$

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 $[0, \pi]$.

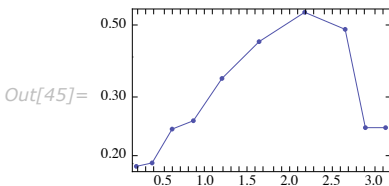
```
In[35]:= n = 9;
h = N[π / (n + 1)];
amat = SparseArray[
  {{i_, i_} → 2 h / 3, {i_, j_} /; Abs[i - j] == 1 → h / 6}, {n + 2, n + 2}, 0.];
rmat = SparseArray[{{i_, i_} → -2 / h, {i_, j_} /; Abs[i - j] == 1 → 1 / h},
  {n + 2, n + 2}, 0.];
vars = {y[t]};
eqs = {amat.y'[t] == rmat.(Exp[t] y[t])};
ics = {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]
```

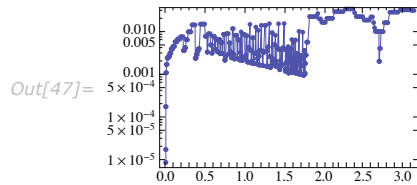


The default method for this type of problem is “IDA” which is a general purpose differential algebraic equation solver [HT99]. Being much more general in scope, this method somewhat overkill for this example but serves for comparison purposes.

```
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]`

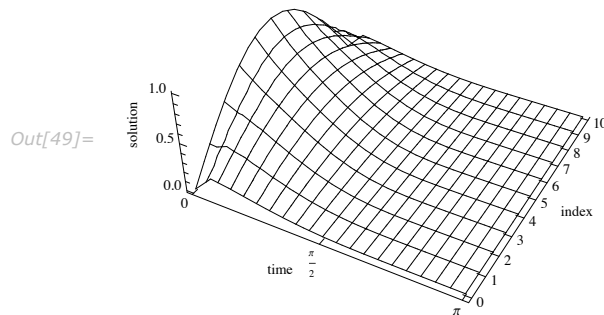


Define a function that can be used to plot the solutions on a grid.

`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\",
 RowBoxes[RotationBox[\"solution\\n\", BoxRotation -> Pi / 2]]},
Mesh -> {19, 9}, MaxRecursion -> 0, PlotStyle -> None]`



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" $\rightarrow \{s_1, s_2\}$ constrains the choice of step size as follows. The step size chosen by the method for order p satisfies:

$$h_{n+1} = h_n s_1 \left(s_2 \frac{Tol}{\|err_n\|} \right)^{\frac{1}{p+1}}. \quad (1)$$

This includes both an order-dependent factor and an order-independent factor.

"StepSizeRatioBounds"

The option "StepSizeRatioBounds" $\rightarrow \{sr_{min}, sr_{max}\}$ specifies bounds on the next step size to take such that:

$$sr_{min} \leq \left| \frac{h_{n+1}}{h_n} \right| \leq sr_{max}.$$

"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" $\rightarrow \{f_1, f_2\}$ 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[{{(Y1)'[T] == 1 - 4 Y1[T] + Y1[T]^2 Y2[T], (Y2)'[T] == 3 Y1[T] - Y1[T]^2 Y2[T]},
  {Y1[0] ==  $-\frac{3}{2}$ , Y2[0] == 3}, {Y1[T], Y2[T]}, {T, 0, 20}, {}, {}, {}]}
```

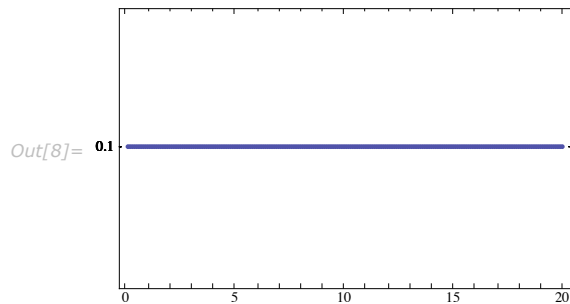
This integrates a differential system using the method "ExplicitEuler" with a fixed step size of 1/10.

```
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.}}, <>]}}
```

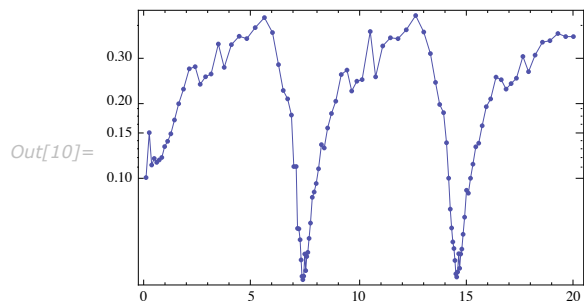
Actually the "ExplicitEuler" method has no adaptive step size control. Therefore, the integration is already carried out using fixed step sizes so the specification of "FixedStep" is unnecessary.

```
In[7]:= sol = NDSolve[system, StartingStepSize -> 1/10, Method -> "ExplicitEuler"];
StepDataPlot[sol, PlotRange -> {0, 0.2}]
```



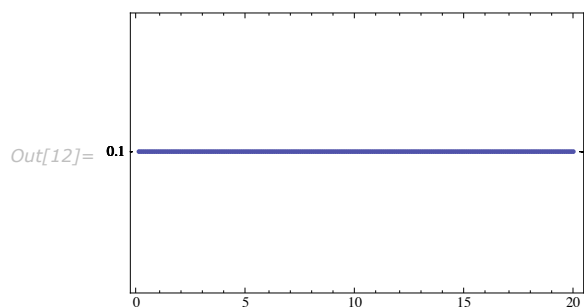
Here are the step sizes taken by the method "ExplicitRungeKutta" for this problem.

```
In[9]:= sol = NDSolve[system, StartingStepSize → 1 / 10, Method → "ExplicitRungeKutta"];
StepDataPlot[sol]
```



This specifies that fixed step sizes should be used for the method "ExplicitRungeKutta".

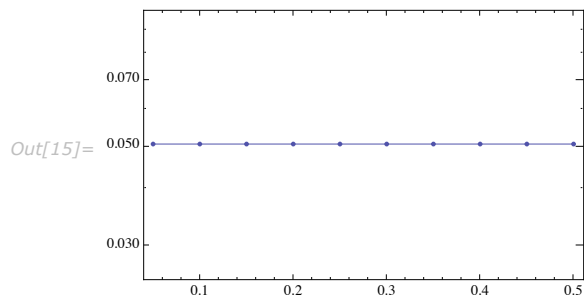
```
In[11]:= sol = NDSolve[system, StartingStepSize → 1 / 10,
Method → {"FixedStep", Method → "ExplicitRungeKutta"}];
StepDataPlot[sol, PlotRange → {0, 0.2}]
```



The option `MaxStepFraction` provides an absolute bound on the step size that depends on the integration interval.

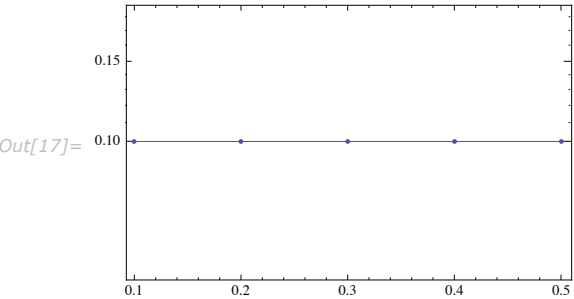
Since the default value of `MaxStepFraction` is `1/10`, the step size in this example is bounded by one-tenth of the integration interval, which leads to using a constant step size of `1/20`.

```
In[13]:= time = {T, 0, 1 / 2};
sol = NDSolve[system, time, StartingStepSize → 1 / 10,
Method → {"FixedStep", Method → "ExplicitRungeKutta"}];
StepDataPlot[sol, PlotRange → {0, 0.2}]
```



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}]
```



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 , $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_p) \in \mathbb{R}^{p \times p}$, where $\sigma_1 \geq \dots \geq \sigma_p \geq 0$. 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:

$$\min_{Z \in \mathbb{R}^{m \times p}} \{ \|A - Z\| : Z^T Z = I \}$$

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:

$$Y_{i+1} = Y_i + Y_i(I - Y_i^T Y_i)/2, Y_0 = A. \quad (1)$$

The Schulz iteration has an arithmetic operation count per iteration of $2m^2p + 2mp^2$ 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 ||YTY - 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

$$\begin{aligned} Y' &= F(Y) Y \\ &= (A + (I - Y Y^T)) Y \end{aligned}$$

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 $\lambda_1 = 1$, $\lambda_2 = \exp(i t \sqrt{3})$, $\lambda_3 = \exp(-i t \sqrt{3})$. Thus as t approaches $\pi/\sqrt{3}$, two of the eigenvalues of $Y(t)$ approach -1. The numerical integration is carried out on the interval $[0, 2]$.

In[7]:= `n = 3;`

$$\mathbf{A} = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & 1 \\ -1 & -1 & 0 \end{pmatrix};$$

`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};`

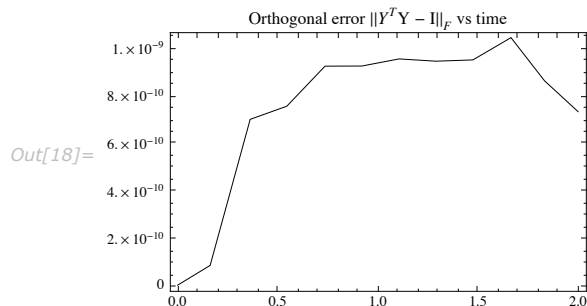
This computes the solution using an explicit Runge-Kutta method. The appropriate initial step size and method order are selected automatically, and the step size may vary throughout the integration interval, which is chosen in order to satisfy local relative and absolute error tolerances. Alternatively, 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]`



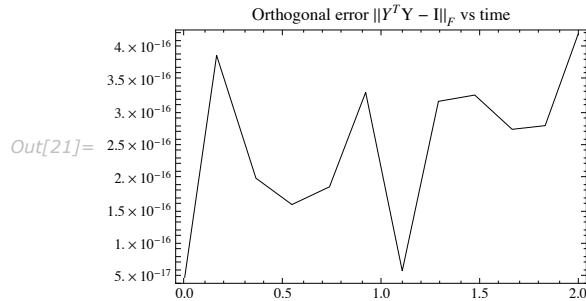
This computes the solution using an orthogonal projection method with an explicit Runge-Kutta method used for the basic integration step. The initial step size and method order are the same as earlier, but the step size sequence in the integration may differ.

```
In[19]:= solop = NDSolve[eqs, vars, time, Method -> {"OrthogonalProjection",
  Method -> "ExplicitRungeKutta", Dimensions -> Dimensions[Y]}];
```

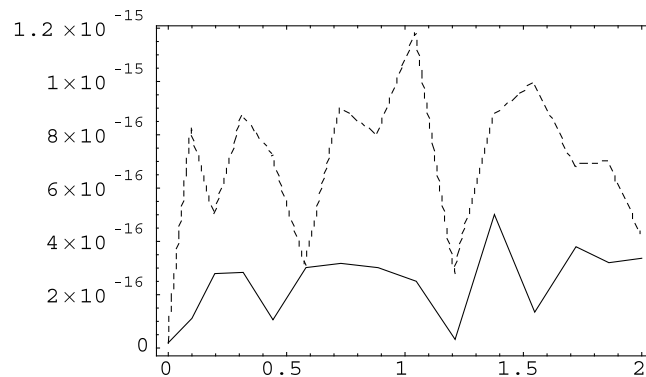
Using the orthogonal projection method, the orthogonal error is reduced to approximately the level of roundoff in IEEE double-precision arithmetic.

```
In[20]:= solop = Y /. First[solop];
```

```
OrthogonalErrorPlot[solop]
```



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 $V_{n,p}(\mathbb{R}) = \{Y \in \mathbb{R}^{n \times p} \mid Y^T Y = I_p\}$, $1 \leq p < n$, 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 $q_0 = \frac{1}{\sqrt{n}} [1, \dots, 1]^T$, $A = \text{diag}[a_1, \dots, a_n] \in \mathbb{R}^{n \times n}$, with $a_i = (-1)^i \alpha$, $i = 1, \dots, n$ 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 $V_{n,1}(\mathbb{R})$:

$$\begin{aligned} Y' &= F(Y) Y \\ &= (I_n - Y Y^T) A Y \end{aligned}$$

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];
```

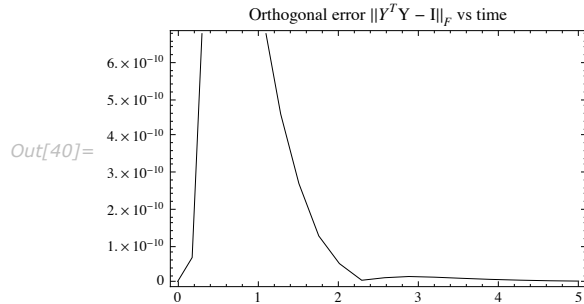
This computes the componentwise absolute global error at the end of the integration interval.

```
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]
```



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];
```

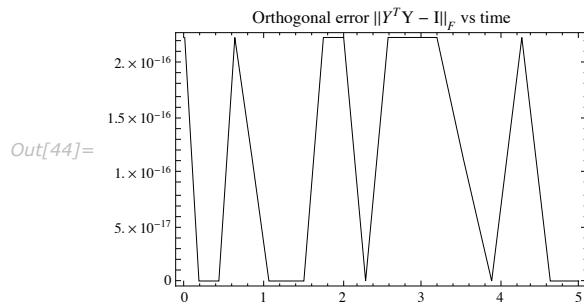
The componentwise absolute global error at the end of the integration interval is roughly the same as before since the absolute and relative tolerances used in the numerical integration are the same.

```
In[43]:= (solexact - solop) /. t → tfinal
```

```
Out[43]= {{-2.03407 × 10-11}, {2.55351 × 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]
```



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 ($ULP \approx 2.22045 \times 10^{-16}$ 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

$$\dot{y} = f(y), \quad y(t_0) = y_0, \tag{1}$$

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 $g : \mathbb{R}^n \mapsto \mathbb{R}^m$:

$$\mathcal{M} = \{y; g(y) = 0\}. \quad (1)$$

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]):

$$\begin{aligned} y_{n+1} &= \tilde{y}_{n+1} + g'(\tilde{y}_{n+1})^T \lambda \\ 0 &= g(\tilde{y}_{n+1}). \end{aligned} \quad (1)$$

To save work $g(y_{n+1})$ is approximated as $g(\tilde{y}_{n+1})$. Substituting the first relation into the second relation in (1) leads to the following simplified Newton scheme for λ :

$$\begin{aligned} \Delta \lambda_i &= - \left(g'(\tilde{y}_{n+1}) g'(\tilde{y}_{n+1})^T \right)^{-1} g(\tilde{y}_{n+1} + g'(\tilde{y}_{n+1})^T \lambda_i), \\ \lambda_{i+1} &= \lambda_i + \Delta \lambda_i \end{aligned} \quad (2)$$

with $\lambda_0 = 0$.

The first increment $\Delta \lambda_0$ is of size $O(h_n^{p+1})$ 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.

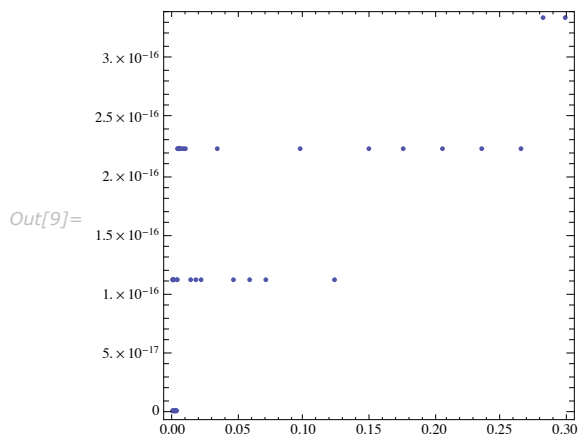
```
In[7]:= invariant = system["Invariants"]
```

```
Out[7]= {Y1[T] + Y2[T] + Y3[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]
```



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.

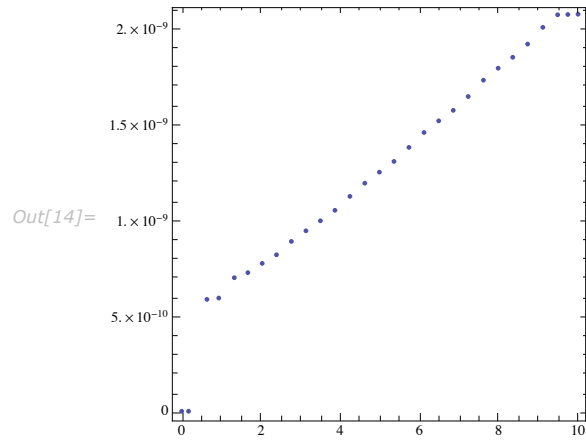
```
In[12]:= invariant = system["Invariants"]
```

```
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"];
```

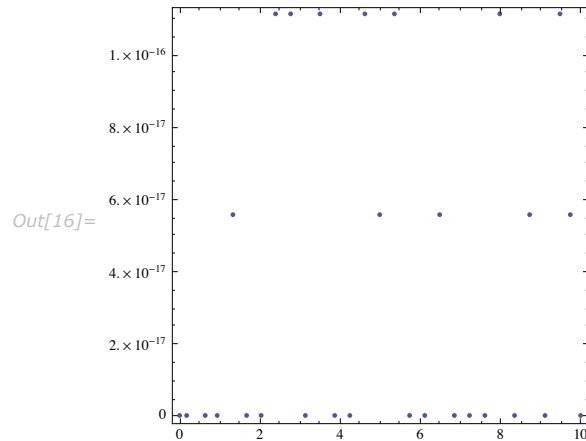
```
InvariantErrorPlot[invariant, vars, T, erksol]
```



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]
```



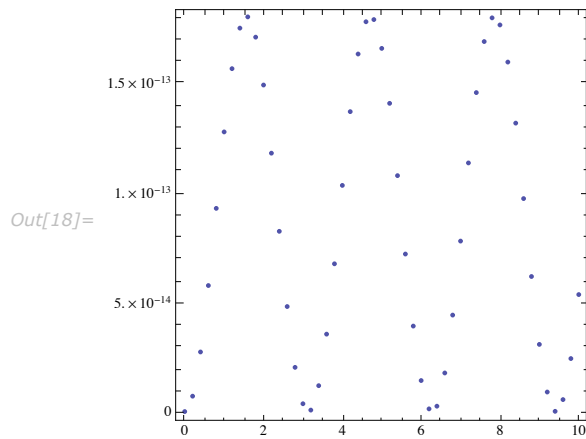
Since the system is Hamiltonian (the invariant is the Hamiltonian), a symplectic integrator performs well on this problem, giving a small bounded error.

```
In[17]:= projekrsol = NDSolve[system,  

Method → {"SymplecticPartitionedRungeKutta", "DifferenceOrder" → 8,  

"PositionVariables" → {Y1[T]}}, StartingStepSize → 1 / 5];
```

```
InvariantErrorPlot[invariant, vars, T, projekrsol]
```



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"]
```

$$\text{Out[23]} = \left\{ -\frac{1}{400 (Y_1[T]^2 + Y_2[T]^2)^{3/2}} - \frac{1}{\sqrt{Y_1[T]^2 + Y_2[T]^2}} + \frac{1}{2} (Y_3[T]^2 + Y_4[T]^2), -Y_2[T] Y_3[T] + Y_1[T] Y_4[T] \right\}$$

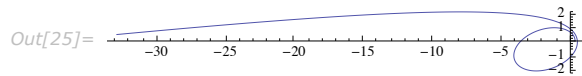
An experiment now illustrates the importance of using all the available invariants in the projective process (see [HLW02]). Consider the solutions obtained using:

- 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

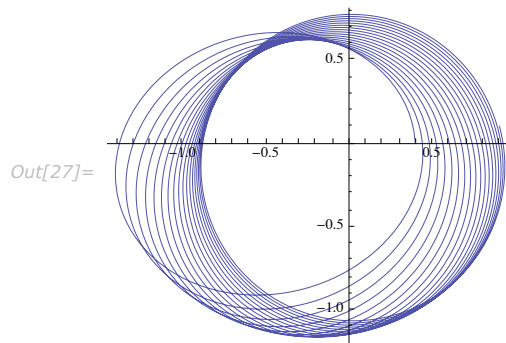
```
In[24]:= sol = NDSolve[system, Method -> "ExplicitEuler", StartingStepSize -> step];
```

```
ParametricPlot[Evaluate[pvars /. First[sol]], Evaluate[time]]
```



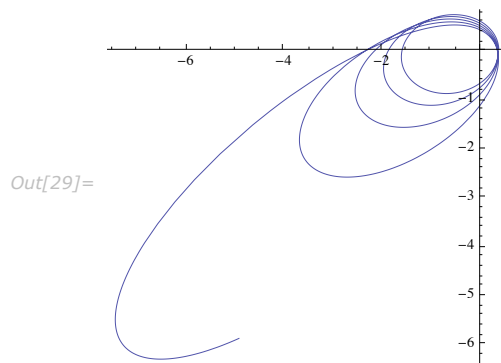
```
In[26]:= sol = NDSolve[system, Method -> {"Projection", Method -> "ExplicitEuler",  
"Invariants" -> {H}}, StartingStepSize -> step];
```

```
ParametricPlot[Evaluate[pvars /. First[sol]], Evaluate[time]]
```

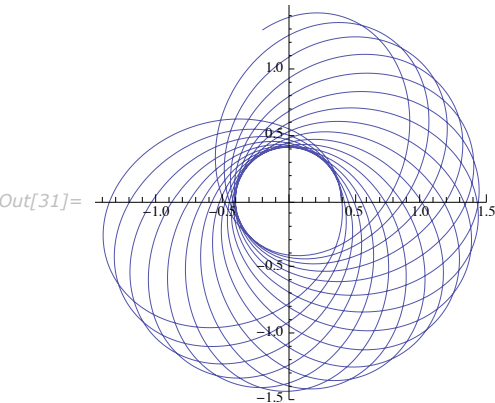


```
In[28]:= sol = NDSolve[system, Method -> {"Projection", Method -> "ExplicitEuler",  
"Invariants" -> {L}}, StartingStepSize -> step];
```

```
ParametricPlot[Evaluate[pvars /. First[sol]], Evaluate[time]]
```



```
In[30]:= sol = NDSolve[system, Method -> {"Projection", Method -> "ExplicitEuler",  
      "Invariants" -> {H, L}}, StartingStepSize -> step];  
  
ParametricPlot[Evaluate[pvars /. First[sol]], Evaluate[time]]
```



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

<i>option name</i>	<i>default value</i>	
"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]}}
```

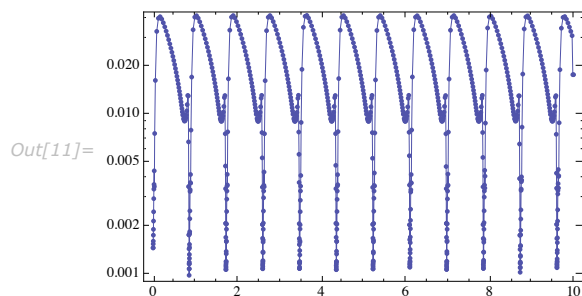
This sets the “ExplicitRungeKutta” method as a submethod of “StiffnessSwitching”.

```
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]
```



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.

<i>AlgorithmIdentifier</i> [<i>data</i>]	an algorithm object that contains any <i>data</i> that a particular numerical ODE integration algorithm may need to use; the data is effectively private to the algorithm; <i>AlgorithmIdentifier</i> should be a <i>Mathematica</i> symbol, and the algorithm is accessed from <code>NDSolve</code> by using the option <code>Method -> AlgorithmIdentifier</code>
--	---

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.

<code>AlgorithmObject ["Step" [<i>rhs</i>, <i>t</i>, <i>h</i>, <i>y</i>, <i>yp</i>]]</code>	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 $\{newh, \Delta y\}$ where $newh$ is the best size for the next step determined by the algorithm and Δy is the <i>increment</i> such that the approximate solution at time $t + h$ is given by $y + \Delta y$; 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)
<code>AlgorithmObject ["DifferenceOrder"]</code>	return the current asymptotic difference order of the algorithm
<code>AlgorithmObject ["StepMode"]</code>	return the step mode for the algorithm object where the step mode should either be <code>Automatic</code> or <code>Fixed</code> ; <code>Automatic</code> means that the algorithm has a means to estimate error and determines an appropriate size $newh$ for the next time step; <code>Fixed</code> 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 $rhs[t + h, y + \Delta y]$ at the step end, so to avoid recomputation, you can add that to the list.

<i>"Step" [rhs, t, h, y, yp] method output</i>	<i>interpretation</i>
<code>{newh, Δ y}</code>	successful step with computed solution increment Δ y and recommended next step <i>newh</i>
<code>{newh, Δ y, yph}</code>	successful step with computed solution increment Δ y and recommended next step <i>newh</i> and time derivatives computed at the step endpoint, $yph = rhs[t + h, y + \Delta y]$
<code>{newh, Δ y, yph, newobj}</code>	successful step with computed solution increment Δ y and recommended next step <i>newh</i> and time derivatives computed at the step endpoint, $yph = rhs[t + h, y + \Delta y]$; any changes in the object data are returned in the new instance of the method object, <i>newobj</i>
<code>{newh, Δ y, None, newobj}</code>	successful step with computed solution increment Δ y and recommended next step <i>newh</i> ; any changes in the object data are returned in the new instance of the method object, <i>newobj</i>
<code>{newh}</code>	rejected step with recommended next step <i>newh</i> such that $ newh < h $
<code>{newh, \$Failed, None, newobj}</code>	rejected step with recommended next step <i>newh</i> such that $ newh < h $; any changes in the object data are returned in the new instance of the method object, <i>newobj</i>

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  $\pi$ }, Method -> CRK4]  
Out[4]= {{x -> InterpolatingFunction[{{0., 6.28319}}, <>]}}
```

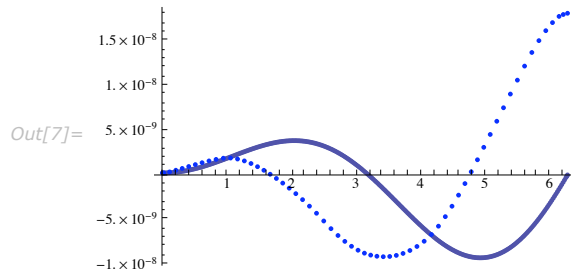
Generally using a fixed step size is less efficient than allowing the step size to vary with the local difficulty of the integration. Modern explicit Runge-Kutta methods (accessed in `NDSolve` with `Method -> "ExplicitRungeKutta"`) have a so-called embedded error estimator that makes it possible to very 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  $\pi$ }, 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]]
}]
```



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 $\{h_{n-k+1}, h_{n-k+2}, \dots, h_n\}$, 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.

$$\Phi_j(n) = \prod_{i=0}^{j-1} (t_n - t_{n-i}) \delta^k f[t_n, \dots, t_{n-j}]$$

$$(\Phi_j)^*(n) = \beta_j(n) \Phi_j(n) \quad \text{with} \quad \beta_j(n) = \prod_{i=0}^{j-1} \frac{t_{n+1} - t_{n-i}}{t_n - t_{n-i-1}}.$$

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];
  brat =  $\frac{\text{Drop}[\Delta h, -1]}{\text{Drop}[\Delta h, 1] - h}$ ;
  β = FoldList[Times, 1, brat];
  α =  $\frac{h}{\Delta h}$ ;
  σ = FoldList[Times, 1, α Range[Length[α]]];
  c[0] = Table[ $\frac{1}{q}$ , {q, 1, k}];
  c[1] = Table[ $\frac{1}{q(q+1)}$ , {q, 1, k}];
  Do[c[j] = Drop[c[j-1], -1] -  $\frac{\text{Drop}[c[j-1], 1] h}{\Delta h[[j]]}$ , {j, 2, k}];
  {(First[c[#1]] &) /@ Range[0, k], β, σ}
```

hlist is the list of step sizes $\{h_{n-k}, h_{n-k+1}, \dots, h_n\}$ 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,  $\Phi$ , mord},
mord = MaxDifferenceOrder /. Flatten[{opts, Options[AdamsBM]}];
If[mord  $\neq$   $\infty$  && ! (IntegerQ[mord] && mord > 0), Return[$Failed]];
prec = ndstate["WorkingPrecision"];
norm = ndstate["Norm"];
hlist = {};
 $\Phi$  = {ndstate["SolutionDerivativeVector"]["Active"]};
AdamsBM[{{hlist,  $\Phi$ , N[0, prec]  $\Phi$ [[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, x, x1, ns, starting, k, zero,
  g, β, σ, p, f, Δy, normh, ev, err, PE, knew, hnew, temp},
  {{hlist, x, x1}, {norm, prec, mord, ns, starting}} = data;
  (* Norm scaling will be based on current solution y. *)
  normh = (Abs[h] temp[x1, y] &) /. {temp -> norm};
  k = Length[x];
  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 x to x* *)
  x = x Reverse[β];
  (* PE: Predict and evaluate *)
  p = Reverse[Drop[g, -1]].x;
  f = rhs[h + t, hp + y];
  (* Update divided differences *)
  x = FoldList[Plus, zero x1, x];
  (* Compute scaled error estimate *)
  ev = f - Last[x];
  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, x, ev, normh, σ];
  If[err > 1,
    (* Rejected step: reduce h by half,
    make sure starting mode flag is unset and reset x to previous values *)
```

```

      h
hnew = -; Δy = $Failed; f = None; starting = False; x = data[[1, 2]],
      2
(* Successful step:
  CE: Correct and evaluate *)
Δy = h (p + ev Last[g]);
f = rhs[h + t, y + Δy]; temp = f - Last[x];
(* Update the divided differences *)
x = (temp + #1 &) / @ x;
(* Determine best order and stepsize for the next step *)
x1 = temp - x1;
knew = ChooseNextOrder[starting, PE, k, knew, x1, normh, σ, mord, ns];
hnew = ChooseNextStep[PE, knew, h];
(* Truncate hlist and x to the appropriate length for the chosen order. *)
hlist = Take[hlist, 1 - knew];
If[Length[x] > knew, x1 = x[[Length[x] - knew]]; x = Take[x, -knew];];
(* Return step data along with updated method data *)
{hnew, Δy, f, AdamsBM[{{hlist, x, x1}, {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_, x_, 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 + x[[2]]]];
    If[k > 2,
      PE_{k-2} = Abs[σ[[k - 1]] Moulton[k - 2] normh[ev + x[[3]]]];
      If[Max[PE_{k-1}, PE_{k-2}] < PE_k, knew = k - 1];
      If[PE_{k-1} <  $\frac{PE_k}{2}$ , knew = k - 1];
    ];
  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_, knew_, x1_, normh_, σ_, mord_, ns_] :=
  Module[{knew = knew},
    starting = starting && knew ≥ k && k < mord;

```

```

If[starting,
  knew = k + 1; PEk+1 = 0,
  If[knew ≥ k && ns ≥ k + 1,
    PEk+1 = Abs[Moulton[k + 1] normh[ $\frac{1}{2}$ ]];
    If[k > 1,
      If[PEk-1 ≤ Min[PEk, PEk+1],
        knew = k - 1,
        If[PEk+1 < PEk && k < mord, knew = k + 1]
      ],
      If[PEk+1 <  $\frac{PE_k}{2}$ , 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[PEk < 2-(k+2),
    2 h,
    If[PEk <  $\frac{1}{2}$ , h, h Max[ $\frac{1}{2}$ , Min[ $\frac{9}{10}$ ,  $\left(\frac{1}{2 PE_k}\right)^{\frac{1}{k+1}}$ ]]]
  ];

```

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}}, <>]}}

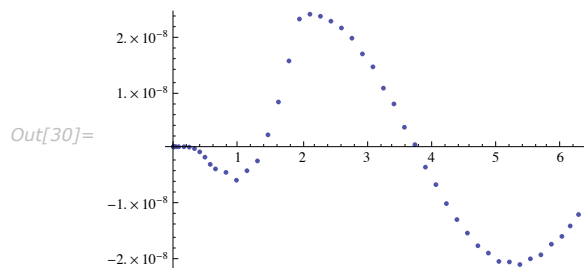
```

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]

```



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).

$$u_t = \frac{1}{8} u_{xx}, u(0, t) = \sin(2\pi t), u_x(1, t) = 0, u(x, 0) = 0 \quad (1)$$

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

$$u_{xx}(x, t) \simeq \frac{u(x+h, t) - 2u(x, t) + u(x-h, t)}{h^2} \quad (2)$$

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 , $0 \leq i \leq n$ 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; h_n =  $\frac{1}{n}$ ;
```

This defines the vector of u_i .

```
In[2]:= U[t_] = Table[u_i[t], {i, 0, n}]
```

```
Out[2]= {u_0[t], u_1[t], u_2[t], u_3[t], u_4[t], u_5[t], u_6[t], u_7[t], u_8[t], u_9[t], u_10[t]}
```

For $1 \leq i \leq 9$, 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 $0 \leq x \leq 2$ 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]/hn2, U[t], {1, 2}, {un-1[t]}]/8]]

Out[3]= {u0'[t] == 2 π Cos[2 π t], u1'[t] ==  $\frac{1}{8} (100 u_0[t] - 200 u_1[t] + 100 u_2[t])$ ,
      u2'[t] ==  $-\frac{1}{8} (100 u_1[t] - 200 u_2[t] + 100 u_3[t])$ ,
      u3'[t] ==  $-\frac{1}{8} (100 u_2[t] - 200 u_3[t] + 100 u_4[t])$ , u4'[t] ==  $-\frac{1}{8} (100 u_3[t] - 200 u_4[t] + 100 u_5[t])$ ,
      u5'[t] ==  $-\frac{1}{8} (100 u_4[t] - 200 u_5[t] + 100 u_6[t])$ , u6'[t] ==  $-\frac{1}{8} (100 u_5[t] - 200 u_6[t] + 100 u_7[t])$ ,
      u7'[t] ==  $-\frac{1}{8} (100 u_6[t] - 200 u_7[t] + 100 u_8[t])$ , u8'[t] ==  $-\frac{1}{8} (100 u_7[t] - 200 u_8[t] + 100 u_9[t])$ ,
      u9'[t] ==  $-\frac{1}{8} (100 u_8[t] - 200 u_9[t] + 100 u_{10}[t])$ , u10'[t] ==  $\frac{1}{8} (200 u_9[t] - 200 u_{10}[t])$ }
```

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`.

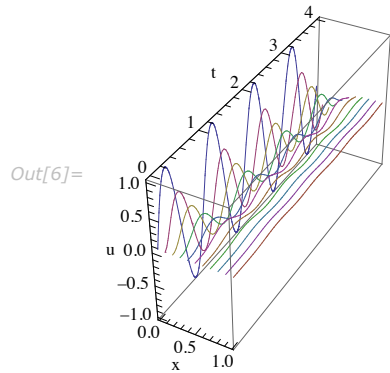
This solves the ODE initial value problem.

```
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"}]
```



The plot indicates why this technique is called the numerical "method of lines".

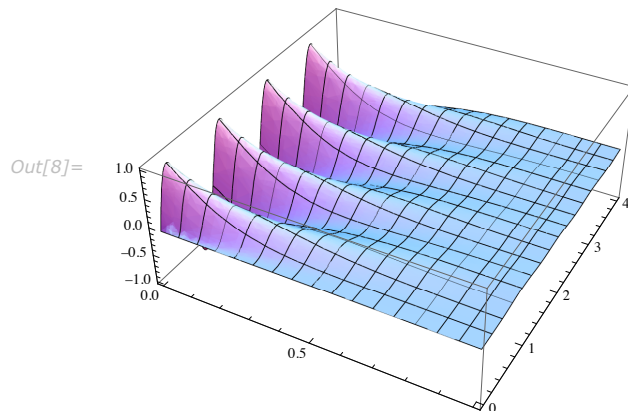
The solution in between lines can be found by interpolation. When `NDSolve` computes the solution for the PDE, the result is a two-dimensional `InterpolatingFunction`.

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]
```



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.

<i>option name</i>	<i>default value</i>	
<code>TemporalVariable</code>	<code>Automatic</code>	what variable to keep derivatives with respect to the derived ODE or DAE system
<code>Method</code>	<code>Automatic</code>	what method to use for integrating the ODEs or DAEs
<code>SpatialDiscretization</code>	<code>TensorProductGrid</code>	what method to use for spatial discretization
<code>DifferentiateBoundaryConditions</code>	<code>True</code>	whether to differentiate the boundary conditions with respect to the temporal variable
<code>ExpandFunctionSymbolically</code>	<code>False</code>	whether to expand the effective function symbolically or not
<code>DiscretizedMonitorVariables</code>	<code>False</code>	whether to interpret dependent variables given in monitors like <code>StepMonitor</code> or in method options for methods like <code>EventLocator</code> and <code>Projection</code> 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

$$f'(x_i) == \lim_{h \rightarrow 0} \frac{f(h + x_i) - f(x_i)}{h}$$

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

$$f(x_{i+1}) = f(x_i) + h f'(x_i) + \frac{h^2}{2} f''(x_i) + O(h^3)$$

from

$$f(x_{i-1}) = f(x_i) - h f'(x_i) + \frac{h^2}{2} f''(x_i) + O(h^3)$$

and solving for $f'(x_i)$ gives the second-order centered difference formula for the first derivative,

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1}))}{2h} + O(h^2)$$

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.

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1}))}{h^2} + O(h^2)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Out[13]= {1/(24 h), -9/(8 h), 9/(8 h), -1/(24 h)}
```

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

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

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

Finite difference formulas on uniform grids for the first derivative.

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

$f''(x_i) \simeq \frac{10 f(x_{i-1}) - 15 f(x_i) - 4 f(x_{i+1}) + 14 f(x_{i+2}) - 6 f(x_{i+3}) + f(x_{i+4})}{12 h^2}$	$\frac{13}{180} h^4 f^{(6)}$
$f''(x_i) \simeq \frac{45 f(x_i) - 154 f(x_{i+1}) + 214 f(x_{i+2}) - 156 f(x_{i+3}) + 61 f(x_{i+4}) - 10 f(x_{i+5})}{12 h^2}$	$\frac{137}{180} h^4 f^{(6)}$
$f'''(x_i) \simeq \frac{1}{180 h^2} (-126 f(x_{i-7}) + 1019 f(x_{i-6}) - 3618 f(x_{i-5}) + 7380 f(x_{i-4}) - 9490 f(x_{i-3}) + 7911 f(x_{i-2}) - 4014 f(x_{i-1}) + 938 f(x_i))$	$\frac{363}{560} h^6 f^{(8)}$
$f'''(x_i) \simeq \frac{1}{180 h^2} (11 f(x_{i-6}) - 90 f(x_{i-5}) + 324 f(x_{i-4}) - 670 f(x_{i-3}) + 855 f(x_{i-2}) - 486 f(x_{i-1}) - 70 f(x_i) + 126 f(x_{i+1}))$	$\frac{29}{560} h^6 f^{(8)}$
$f'''(x_i) \simeq \frac{1}{180 h^2} (-2 f(x_{i-5}) + 16 f(x_{i-4}) - 54 f(x_{i-3}) + 85 f(x_{i-2}) + 130 f(x_{i-1}) - 378 f(x_i) + 214 f(x_{i+1}) - 11 f(x_{i+2}))$	$\frac{47}{5040} h^6 f^{(8)}$
$f'''(x_i) \simeq \frac{2 f(x_{i-3}) - 27 f(x_{i-2}) + 270 f(x_{i-1}) - 490 f(x_i) + 270 f(x_{i+1}) - 27 f(x_{i+2}) + 2 f(x_{i+3})}{180 h^2}$	$\frac{1}{560} h^6 f^{(8)}$
$f'''(x_i) \simeq \frac{1}{180 h^2} (-11 f(x_{i-2}) + 214 f(x_{i-1}) - 378 f(x_i) + 130 f(x_{i+1}) + 85 f(x_{i+2}) - 54 f(x_{i+3}) + 16 f(x_{i+4}) - 2 f(x_{i+5}))$	$\frac{47}{5040} h^6 f^{(8)}$
$f'''(x_i) \simeq \frac{1}{180 h^2} (126 f(x_{i-1}) - 70 f(x_i) - 486 f(x_{i+1}) + 855 f(x_{i+2}) - 670 f(x_{i+3}) + 324 f(x_{i+4}) - 90 f(x_{i+5}) + 11 f(x_{i+6}))$	$\frac{29}{560} h^6 f^{(8)}$
$f'''(x_i) \simeq \frac{1}{180 h^2} (938 f(x_i) - 4014 f(x_{i+1}) + 7911 f(x_{i+2}) - 9490 f(x_{i+3}) + 7380 f(x_{i+4}) - 3618 f(x_{i+5}) + 1019 f(x_{i+6}) - 126 f(x_{i+7}))$	$\frac{363}{560} h^6 f^{(8)}$

Finite difference formulas on uniform grids for the second derivative.

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

NDSolve`FiniteDifferenceDerivative

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

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

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

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

approximate the partial derivative of order $(m_1, m_2, \dots, m_n)$ for the function of n variables that takes on values on the tensor product grid defined by the outer product of $(\text{grid}_1, \text{grid}_2, \dots, \text{grid}_n)$

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

compute the finite difference weights needed to approximate the partial derivative of order $(m_1, m_2, \dots, m_n)$ for the function of n variables on the tensor product grid defined by the outer product of $(\text{grid}_1, \text{grid}_2, \dots, \text{grid}_n)$; the result is returned as an `NDSolve`FiniteDifferenceDerivativeFunction`, which can be repeatedly applied to values on the grid

Finding finite difference approximations to derivatives.

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

`In[14]:= ugrid = h Range[0, 8]`

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

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

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

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

$$-\frac{f[0]}{f[h]} - \frac{5 f[h]}{2 f[2 h]} + \frac{3 f[2 h]}{2 f[4 h]} - \frac{f[3 h]}{f[5 h]} + \frac{f[4 h]}{f[2 h]} - \frac{2 f[3 h]}{2 f[5 h]} + \frac{2 f[3 h]}{2 f[7 h]} - \frac{f[4 h]}{f[6 h]},$$

$$\frac{12 h}{f[3 h]} - \frac{3 h}{2 f[4 h]} + \frac{3 h}{2 f[6 h]} - \frac{12 h}{f[7 h]}, \frac{12 h}{f[4 h]} - \frac{3 h}{2 f[5 h]} + \frac{3 h}{2 f[7 h]} - \frac{12 h}{f[8 h]},$$

$$\frac{12 h}{f[4 h]} - \frac{3 h}{2 f[5 h]} + \frac{3 h}{2 f[7 h]} - \frac{12 h}{f[8 h]}, \frac{12 h}{f[4 h]} - \frac{3 h}{2 f[5 h]} + \frac{3 h}{2 f[7 h]} - \frac{12 h}{f[8 h]},$$

$$\left. -\frac{12 h}{12 h} + \frac{3 h}{2 h} - \frac{3 h}{2 h} + \frac{12 h}{6 h} + \frac{12 h}{4 h}, \frac{12 h}{4 h} - \frac{3 h}{3 h} + \frac{3 h}{h} - \frac{4 f[7 h]}{h} + \frac{25 f[8 h]}{12 h} \right\}$$

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

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

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

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

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

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

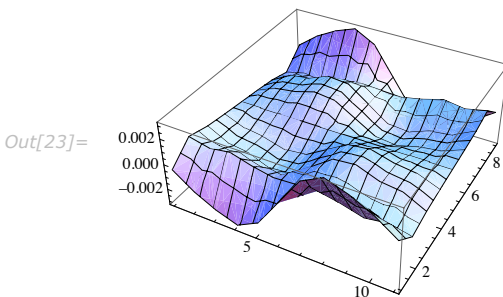
This shows the error in the approximations.

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

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

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

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



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

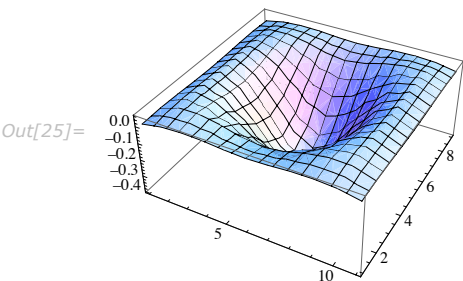
`NDSolve`FiniteDifferenceDerivative` does not compute weights for sums of derivatives. This means that for common operators like the Laplacian, you need to combine two approximations.

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

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

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

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



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

Options for `NDSolve`FiniteDifferenceDerivative`.

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

```
In[26]:= NDSolve`FiniteDifferenceDerivative[  
  1, rgrid, Sin[rgrid], PeriodicInterpolation -> True]  
Out[26]= {0.99895, 0.586765, 0.536072, 0.463601, -0.149152,  
  -0.215212, -0.747842, -0.795502, -0.97065, -0.247503, 0.994585, 0.99895}
```

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

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

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

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

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

NDSolve`FiniteDifferenceDerivativeFunction

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

```
NDSolve`FiniteDifferenceDerivative[{ $m_1, m_2, \dots$ }, { $grid_1, grid_2, \dots$ }]
```

compute the finite difference weights needed to approximate the partial derivative of order $(m_1, m_2, \dots)$ for the function of n variables on the tensor product grid defined by the outer product of $(grid_1, grid_2, \dots)$; the result is returned as an
NDSolve`FiniteDifferenceDerivativeFunction
object

```
NDSolve`FiniteDifferenceDerivativeFunction[Derivative[ $m$ ], data]
```

a data object that contains the weights and other data needed to quickly approximate the m^{th} -order derivative of a function; in the standard output form, only the Derivative[m] operator it approximates is shown

```
NDSolve`FiniteDifferenceDerivativeFunction[data][values]
```

approximate the derivative of the function that takes on values on the grid used to determine data

Computing finite difference weights for repeated use.

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

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

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

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

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

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

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

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

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

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


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

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

```
In[9]:= repeats = 10000;
{First[Timing[Do[fddf[values], {repeats}]]],
 First[Timing[Do[NDSolve`FiniteDifferenceDerivative[
   Derivative[2], grid, values], {repeats}]]]}
Out[10]= {0.047, 2.25}
```

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

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

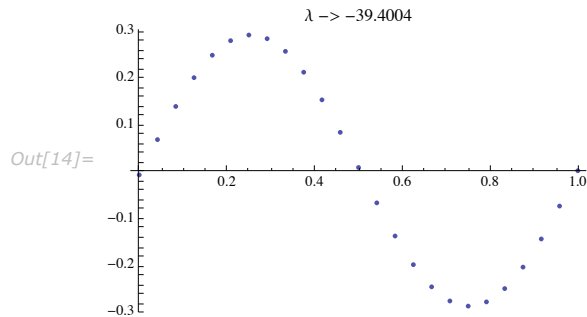
on the unit interval. (This simple method is not necessarily the best way to solve this particular problem, but it is useful as an example.)

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

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

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

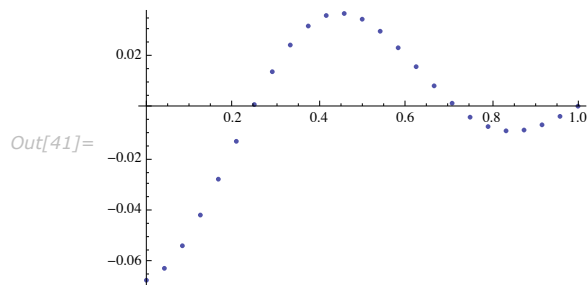
```
In[13]:= s4 = FindRoot[fun[u, λ], {u, values}, {λ, -4 π²}];
ListPlot[Transpose[{grid, u /. s4}], PlotLabel → ToString[Last[s4]]]
```



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

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

```
In[39]:= fddf = NDSolve`FiniteDifferenceDerivative[
  Derivative[2], grid, "DifferenceOrder" → 2];
s2 = FindRoot[fun[u, λ], {u, values}, {λ, -4 π²}];
ListPlot[Transpose[{grid, (u /. s4) - (u /. s2)}]]
```



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

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

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

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

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

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

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

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

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

This shows the dimensions of the outer product grid.

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

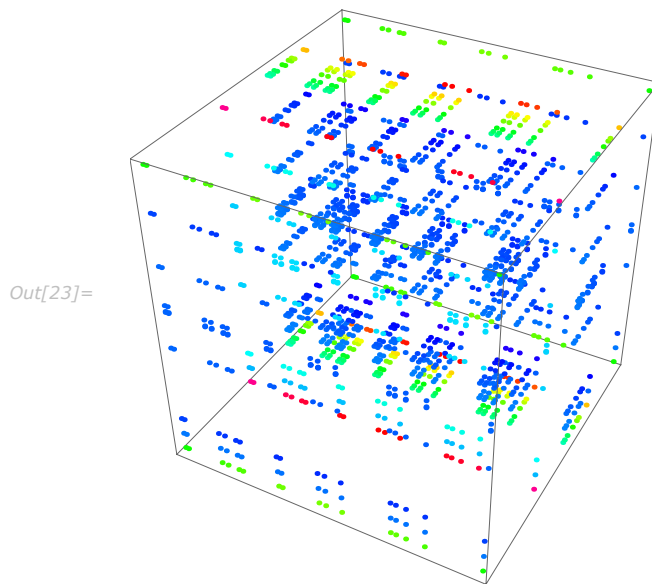
Note that the rank of the grid point tensor is one more than the dimensionality of the tensor product. This is because the three coordinates defining each point are in a list themselves. If you have a function that depends on the grid variables, you can use `Apply[f, fddf["Grid"], {n}]` where *n* = `Length[fddf["DerivativeOrder"]]`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

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

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

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

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



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

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

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

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

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

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

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

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

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

Pseudospectral Derivatives

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

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

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

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

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

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

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

Settings for the "DifferenceOrder" option.

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

```
In[27]:= ugrid = N[2  $\pi$  Range[0, 10] / 10];
      NDSolve`FiniteDifferenceDerivative[2, ugrid, Sin[ugrid],
      PeriodicInterpolation -> True, "DifferenceOrder" -> "Pseudospectral"]
```

```
Out[28]= {1., 0.809017, 0.309017, -0.309017, -0.809017, -1., -0.809017, -0.309017, 0.309017, 0.809017, 1.}
```

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

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

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

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

This defines a simple function that generates a grid of n points with leftmost point at x_0 and interval length L having the spacing of the Chebyshev-Gauss-Lobatto points.

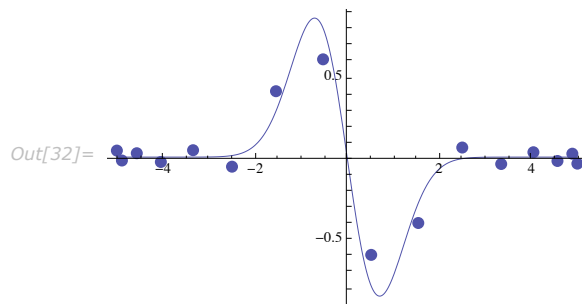
```
In[30]:= CGLGrid[x0_, L_, n_Integer /; n > 1] :=  
          1  
          x0 +  $\frac{1}{2}$  L (1 - Cos[ $\pi$  Range[0, n - 1] / (n - 1)])
```

This computes the pseudospectral derivative for a Gaussian function.

```
In[31]:= cgrid = CGLGrid[-5, 10., 16]; NDSolve`FiniteDifferenceDerivative[  
          1, cgrid, Exp[-cgrid2], "DifferenceOrder" -> "Pseudospectral"]  
Out[31]= {0.0402426, -0.0209922, 0.0239151, -0.0300589, 0.0425553, -0.0590871, 0.40663, 0.60336,  
          -0.60336, -0.40663, 0.0590871, -0.0425553, 0.0300589, -0.0239151, 0.0209922, -0.0402426}
```

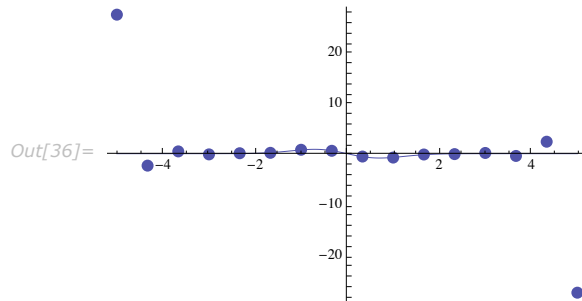
This shows a plot of the approximation and the exact values.

```
In[32]:= Show[{  
          ListPlot[Transpose[{cgrid, %}], PlotStyle -> PointSize[0.025]],  
          Plot[Evaluate[D[Exp[-x2], x]], {x, -5, 5}], PlotRange -> All]
```



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

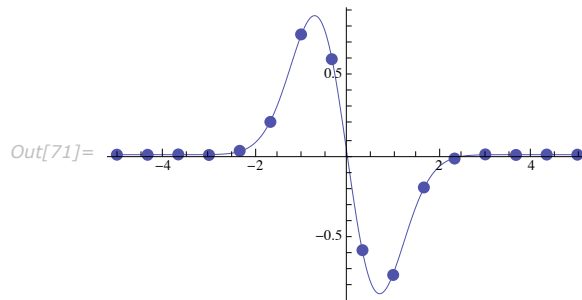
```
In[35]:= ugrid = -5 + 10. Range[0, 15] / 15;
Show[{
  ListPlot[
    Transpose[{ugrid, NDSolve`FiniteDifferenceDerivative[1, ugrid, Exp[-ugrid^2],
      "DifferenceOrder" → Length[ugrid] - 1]}], PlotStyle → PointSize[0.025]],
  Plot[Evaluate[D[Exp[-x^2], x]], {x, -5, 5}], PlotRange → All]
```



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

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

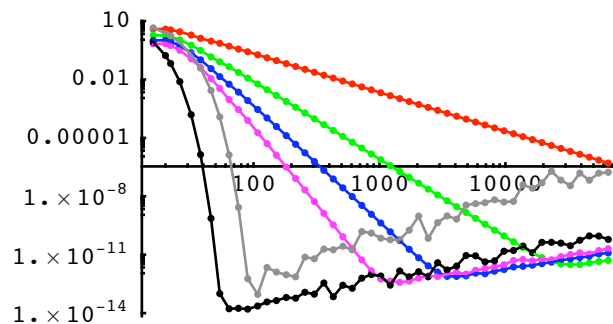
```
In[70]:= ugrid = -5 + 10. Range[0, 15] / 15;
Show[{
  ListPlot[Transpose[{ugrid, NDSolve`FiniteDifferenceDerivative[
    1, ugrid, Exp[-ugrid^2], "DifferenceOrder" → "Pseudospectral",
    PeriodicInterpolation → True]}], PlotStyle → PointSize[0.025]],
  Plot[Evaluate[D[Exp[-x^2], x]], {x, -5, 5}],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



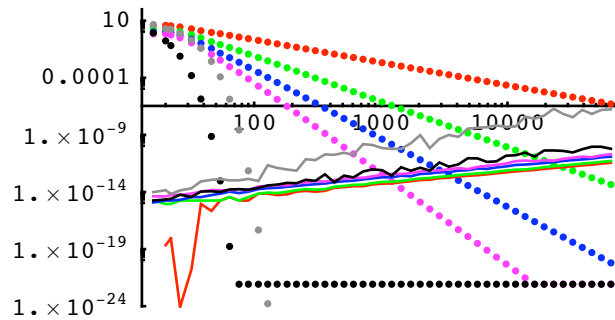
With the assumption of periodicity, the approximation is significantly improved. The accuracy of the periodic pseudospectral approximations is sufficiently high to justify, in some cases, using a larger computational domain to simulate periodicity, say for a pulse like the example. Despite the great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

Accuracy and Convergence of Finite Difference Approximations

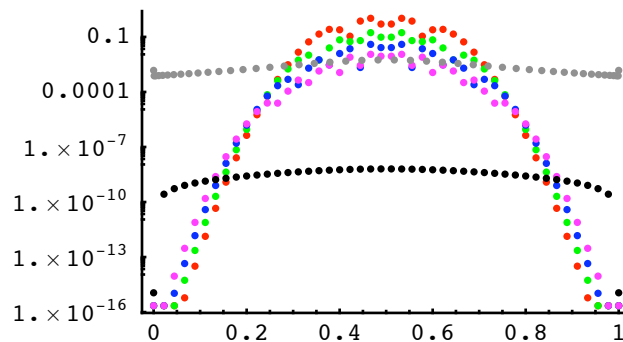
When using finite differences, it is important to keep in mind that the truncation error, or the asymptotic approximation error induced by cutting off the Taylor series approximation, is not the only source of error. There are two other sources of error in applying finite difference formulas; condition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



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



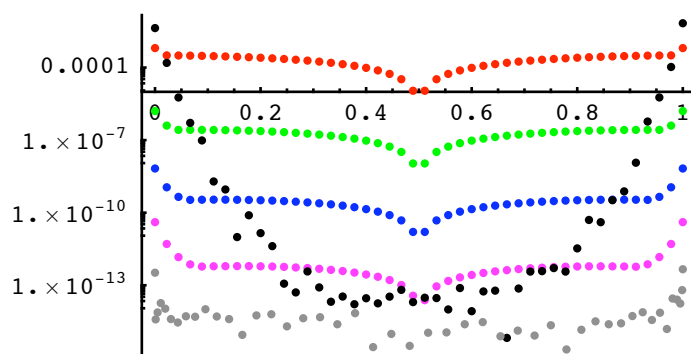
A logarithmic plot of the truncation error (dotted) and the condition and roundoff error (solid line) for approximating the first derivative of the Gaussian $f(x) = e^{-(15(x-1/2))^2}$ at points on a grid covering the interval $[0, 1]$ as a function of the number of grid points, n . Finite differences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



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

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

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

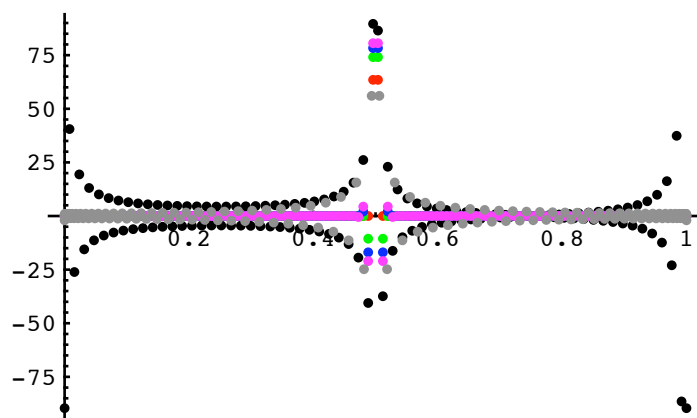


A semilogarithmic plot of the error for approximating the first derivative of the Gaussian $f(x) = e^{-(15(x-1/2))^2}$ as a function of x at points on a 45-point grid covering the interval $[0, 1]$. Finite differences of order 2, 4, 6, and 8 on a uniform grid are shown in red, green, blue, and magenta,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

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

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

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



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

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

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

Differentiation Matrices

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

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

<i>FDDF</i> @ <code>"DifferentiationMatrix"</code>	recast the linear operation of <i>FDDF</i> as a matrix that represents the linear operator
--	--

Forming a differentiation matrix.

This creates a `FiniteDifferenceDerivativeFunction` object.

```
In[37]:= fdd = NDSolve`FiniteDifferenceDerivative[2, Range[0, 10]]
```

```
Out[37]= NDSolve`FiniteDifferenceDerivativeFunction[Derivative[2], <>]
```

This makes a matrix representing the underlying linear operator.

```
In[38]:= smat = fdd["DifferentiationMatrix"]
```

```
Out[38]= SparseArray[<59>, {11, 11}]
```

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

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

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

```
Out[39]//MatrixForm=
```

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

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

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

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

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

This computes the eigenvalues of the differentiation matrix.

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

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

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

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

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

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

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

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

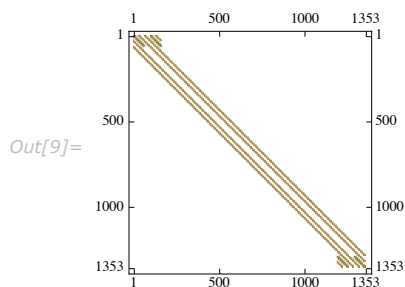
This computes the associated differentiation matrix.

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

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

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

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



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

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

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

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

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

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

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

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

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

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

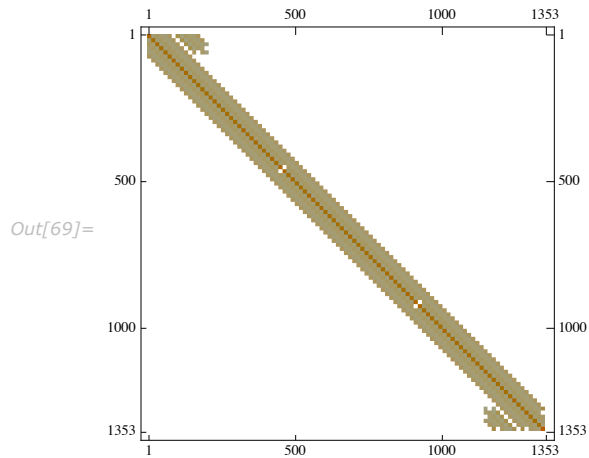
```
In[19]:= dmlist = Map[(Head[#]@"DifferentiationMatrix") &, List@@ First[flap]]
Out[19]= {SparseArray[<6929>, {1353, 1353}], SparseArray[<6897>, {1353, 1353}]}
```

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

```
In[68]:= slap = Total[dmlist]
Out[68]= SparseArray[<12473>, {1353, 1353}]
```


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

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



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

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

```
Out[64]= {0.14, 0.047, 1.39888 × 10-14}
```

Interpretation of Discretized Dependent Variables

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

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

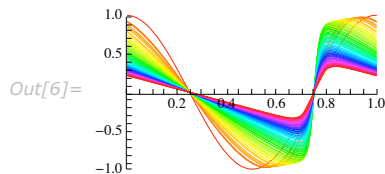
Another possible interpretation for PDEs is to consider the dependent variable at a particular time as representing the spatially discretized values at that time—that is, discretized both in time and space. You can request that monitors and methods use this fully discretized interpretation by using the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

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

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

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

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

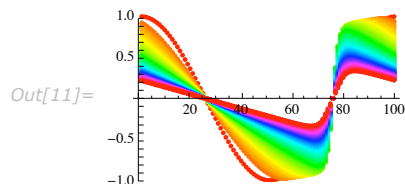


In executing the command above, $u[t, x]$ in the `StepMonitor` is effectively a function of x , so it can be plotted with `plot`. You could do other operations on it, such as numerical integration.

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

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

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



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

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

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

Boundary Conditions

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

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

```
NDSolve[{eqn1, eqn2, ..., u1[t, xmin] == u1[t, xmax], u2[t, xmin] == u2[t, xmax], ...},
{u1[t, x], u2[t, x], ...}, {t, tmin, tmax}, {x, xmin, xmax}]
```

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

```
NDSolve[{eqn1, eqn2, ..., u1[t, x1 min, x2, ...] == u1[t, x1 max, x2, ...],
u2[t, x1 min, x2, ...] == u2[t, x1 max, x2, ...], ...},
{u1[t, x1, x2, ...], u2[t, x1, x2, ...], ...}, {t, tmin, tmax}, {x, xmin, xmax}]
```

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

Giving boundary conditions for partial differential equations.

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

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

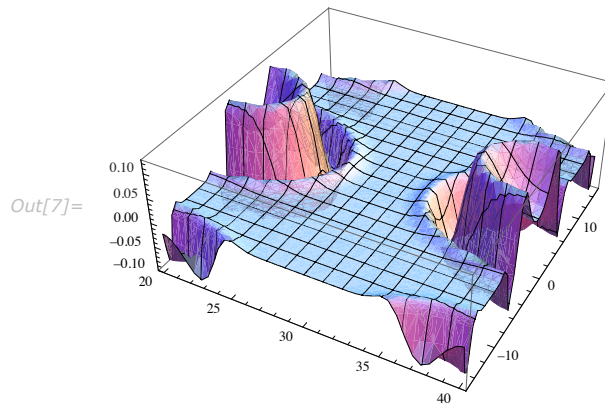
```
In[2]:= sol = NDSolve[{D[u[t, x, y], t, t] ==
D[u[t, x, y], x, x] + D[u[t, x, y], y, y] - Sin[u[t, x, y]],
u[0, x, y] == Exp[-(x^2 + y^2)], Derivative[1, 0, 0][u][0, x, y] == 0,
u[t, -10, y] == u[t, 10, y], u[t, x, -10] == u[t, x, 10]}, u, {t, 0, 6},
{x, -10, 10}, {y, -10, 10}, Method -> {"MethodOfLines",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

Out[2]= {{u -> InterpolatingFunction[{{0., 6.}, {-10., 10.}, {-10., 10.}, <>]]}}
```

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

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

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



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

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

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

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

This defines the vector of u_i .

```
In[9]:= U[t_] = Table[u_i[t], {i, 0, n}]
```

```
Out[9]= {u_0[t], u_1[t], u_2[t], u_3[t], u_4[t], u_5[t], u_6[t], u_7[t], u_8[t], u_9[t], u_10[t]}
```

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

```
In[10]:= bc = Last[NDSolve`FiniteDifferenceDerivative[1, h_n Range[0, n], U[t]]] == 0
```

```
Out[10]= 
$$\frac{5 u_6[t]}{2} - \frac{40 u_7[t]}{3} + 30 u_8[t] - 40 u_9[t] + \frac{125 u_{10}[t]}{6} == 0$$

```

This differentiates the discretized boundary condition with respect to t .

```
In[11]:= bcprime = D[bc, t]
```

```
Out[11]= 
$$\frac{5}{2} u_6'[t] - \frac{40}{3} u_7'[t] + 30 u_8'[t] - 40 u_9'[t] + \frac{125}{6} u_{10}'[t] == 0$$

```

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

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

```
In[12]:= eqns = Thread[
    D[U[t], t] ==  $\frac{1}{8}$  NDSolve`FiniteDifferenceDerivative[2, h_n Range[0, n], U[t]]]
```

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

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

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

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

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

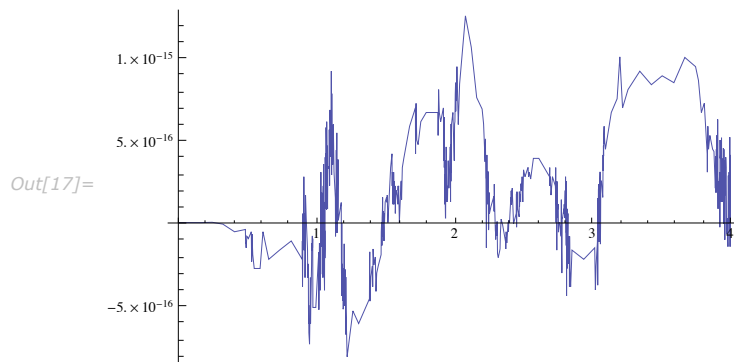
NDSolve is capable of solving the system as is for the appropriate derivatives, so it is ready for the ODEs.

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

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

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

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



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

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

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

$$\text{Out[20]} = \left\{ \begin{aligned} &u_0[t] == \text{Sin}[2 \pi t], \quad u_1'[t] == -\frac{1}{8} \left(\frac{250 u_0[t]}{3} - 125 u_1[t] - \frac{100 u_2[t]}{3} + \frac{350 u_3[t]}{3} - 50 u_4[t] + \frac{25 u_5[t]}{3} \right), \\ &u_2'[t] == -\frac{1}{8} \left(-\frac{25}{3} u_0[t] + \frac{400 u_1[t]}{3} - 250 u_2[t] + \frac{400 u_3[t]}{3} - \frac{25 u_4[t]}{3} \right), \\ &u_3'[t] == -\frac{1}{8} \left(-\frac{25}{3} u_1[t] + \frac{400 u_2[t]}{3} - 250 u_3[t] + \frac{400 u_4[t]}{3} - \frac{25 u_5[t]}{3} \right), \\ &u_4'[t] == -\frac{1}{8} \left(-\frac{25}{3} u_2[t] + \frac{400 u_3[t]}{3} - 250 u_4[t] + \frac{400 u_5[t]}{3} - \frac{25 u_6[t]}{3} \right), \\ &u_5'[t] == -\frac{1}{8} \left(-\frac{25}{3} u_3[t] + \frac{400 u_4[t]}{3} - 250 u_5[t] + \frac{400 u_6[t]}{3} - \frac{25 u_7[t]}{3} \right), \\ &u_6'[t] == -\frac{1}{8} \left(-\frac{25}{3} u_4[t] + \frac{400 u_5[t]}{3} - 250 u_6[t] + \frac{400 u_7[t]}{3} - \frac{25 u_8[t]}{3} \right), \\ &u_7'[t] == -\frac{1}{8} \left(-\frac{25}{3} u_5[t] + \frac{400 u_6[t]}{3} - 250 u_7[t] + \frac{400 u_8[t]}{3} - \frac{25 u_9[t]}{3} \right), \\ &u_8'[t] == -\frac{1}{8} \left(-\frac{25}{3} u_6[t] + \frac{400 u_7[t]}{3} - 250 u_8[t] + \frac{400 u_9[t]}{3} - \frac{25 u_{10}[t]}{3} \right), \\ &u_9'[t] == -\frac{1}{8} \left(\frac{25 u_5[t]}{3} - 50 u_6[t] + \frac{350 u_7[t]}{3} - \frac{100 u_8[t]}{3} - 125 u_9[t] + \frac{250 u_{10}[t]}{3} \right), \\ &\frac{5 u_6[t]}{2} - \frac{40 u_7[t]}{3} + 30 u_8[t] - 40 u_9[t] + \frac{125 u_{10}[t]}{6} == 0 \end{aligned} \right\}$$

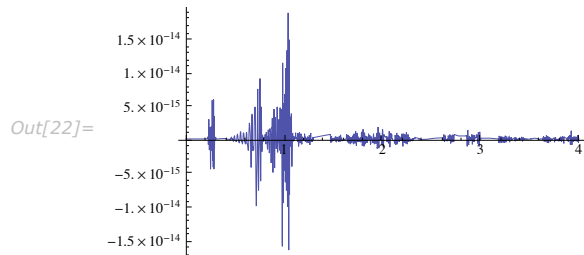
This solves the system of DAEs with NDSolve.

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

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

This shows how well the boundary condition was satisfied.

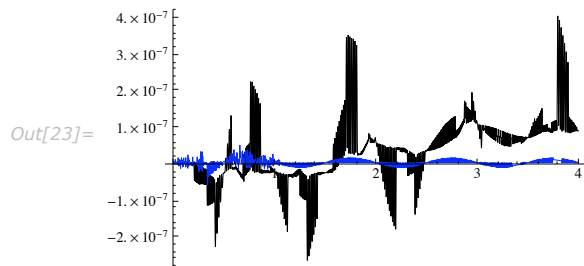
```
In[22]:= Plot[Evaluate[Apply[Subtract, bc] /. daesol], {t, 0, 4}, PlotRange -> All]
```



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

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

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



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

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

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

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

Out[25]= {{u -> InterpolatingFunction[{{0., 1.}, {0., 2.}}, <>]}}
```

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

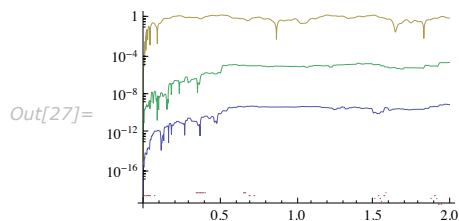
This forms a list of `InterpolatingFunctions` differentiated to the same order as each of the boundary conditions.

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

Out[26]= {{InterpolatingFunction[{{0., 1.}, {0., 2.}}, <>][0, t],
    InterpolatingFunction[{{0., 1.}, {0., 2.}}, <>][0, t]},
    {InterpolatingFunction[{{0., 1.}, {0., 2.}}, <>][1, t],
    InterpolatingFunction[{{0., 1.}, {0., 2.}}, <>][1, t]}}
```

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

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



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

Inconsistent Boundary Conditions

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

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

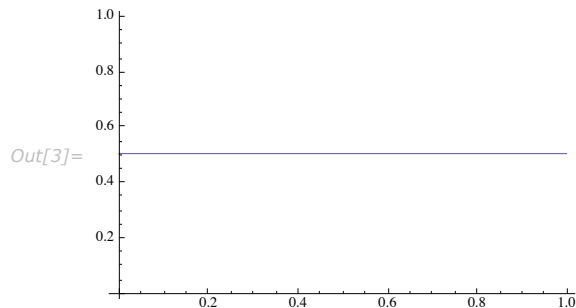
```
In[2]:= sol = NDSolve[{D[u[t, x], t] == D[u[t, x], x, x],
  u[t, 0] == 1, u[t, 1] == 0, u[0, x] == .5}, u, {t, 0, 1}, {x, 0, 1}]
```

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

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

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

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



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

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

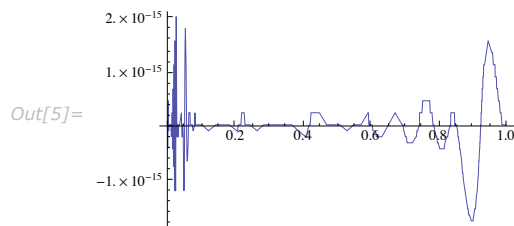
```
In[4]:= daesol = NDSolve[{D[u[t, x], t] == D[u[t, x], x, x],
    u[t, 0] == 1, u[t, 1] == 0, u[0, x] == 0}, u, {t, 0, 1}, {x, 0, 1},
    Method -> {"MethodOfLines", "DifferentiateBoundaryConditions" -> False}];
Plot[First[u[t, 0] /. daesol] - 1, {t, 0, 1}, PlotRange -> All]
```

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

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

NDSolve::ivres:

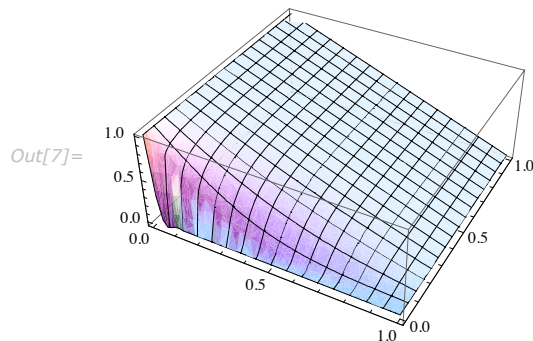
NDSolve has computed initial values that give a zero residual for the differential-algebraic system, but some components are different from those specified. If you need those to be satisfied, it is recommended that you give initial conditions for all dependent variables and derivatives of them.



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

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

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



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

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

Consider the wave equation

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

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

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

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

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

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

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

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

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

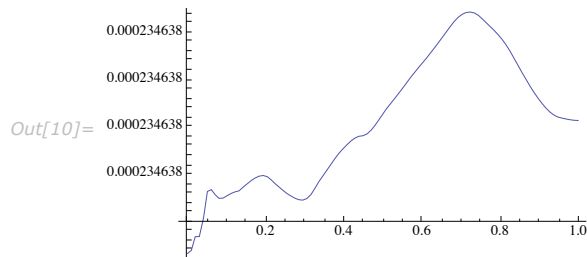
```
In[9]:= sol = NDSolve[{D[u[x, t], t] == D[u[x, t], x, x], u[x, 0] == 1 - Sin[4 * Pi * x] / (4 * Pi),  
u[0, t] == 1, u[1, t] + Derivative[1, 0][u][1, t] == 0}, u, {x, 0, 1}, {t, 0, 1}]
```

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

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

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

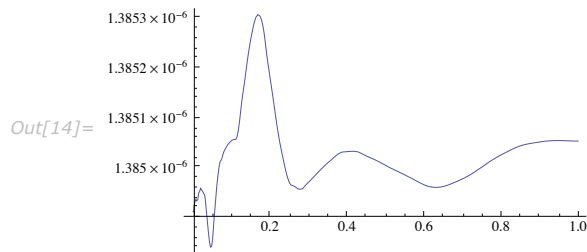
```
In[10]:= Plot[First[u[1, t] + Derivative[1, 0][u][1, t] /. sol], {t, 0, 1}]
```



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

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

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



Spatial Error Estimates

Overview

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

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

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

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

A priori Error Estimates

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

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

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

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


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

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

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

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

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

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

The next stage is to solve for the temporal derivatives.

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

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

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

This sets variables to reflect the default settings for “DifferenceOrder”, `AccuracyGoal`, and `PrecisionGoal`.

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

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

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

so the error in y_1 is estimated to be

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

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

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

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

```
In[12]:= Clear[h, grid];
          h[n_] :=  $\frac{10}{n}$ ;
          grid[n_] := N[-5 + Range[0, n] * h[n]];
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This applies the norm function to the two approximations found.

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

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

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

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

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

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

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

The condition to satisfy is

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

and you have estimated that

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

so you can project that

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

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

$$n_{\text{opt}} \simeq \left\lceil n \text{errest}(n)^{1/p} \right\rceil$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Out[30]//TableForm=
```

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

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

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

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

A posteriori Error Estimates

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

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

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

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

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

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

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

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

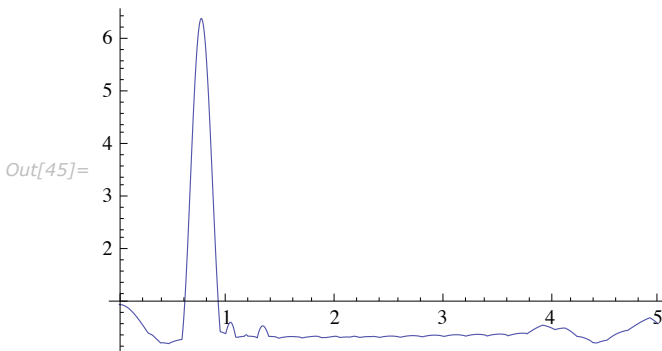
```
In[42]:= ndgrid = InterpolatingFunctionCoordinates[u /. ndsol][[1]]
pggrid = Drop[ndgrid, -1];
Out[42]= {-5., -4.89583, -4.79167, -4.6875, -4.58333, -4.47917, -4.375, -4.27083, -4.16667, -4.0625,
-3.95833, -3.85417, -3.75, -3.64583, -3.54167, -3.4375, -3.33333, -3.22917, -3.125,
-3.02083, -2.91667, -2.8125, -2.70833, -2.60417, -2.5, -2.39583, -2.29167, -2.1875,
-2.08333, -1.97917, -1.875, -1.77083, -1.66667, -1.5625, -1.45833, -1.35417, -1.25,
-1.14583, -1.04167, -0.9375, -0.833333, -0.729167, -0.625, -0.520833, -0.416667,
-0.3125, -0.208333, -0.104167, 0., 0.104167, 0.208333, 0.3125, 0.416667, 0.520833, 0.625,
0.729167, 0.833333, 0.9375, 1.04167, 1.14583, 1.25, 1.35417, 1.45833, 1.5625, 1.66667,
1.77083, 1.875, 1.97917, 2.08333, 2.1875, 2.29167, 2.39583, 2.5, 2.60417, 2.70833, 2.8125,
2.91667, 3.02083, 3.125, 3.22917, 3.33333, 3.4375, 3.54167, 3.64583, 3.75, 3.85417,
3.95833, 4.0625, 4.16667, 4.27083, 4.375, 4.47917, 4.58333, 4.6875, 4.79167, 4.89583, 5.}
```

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

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

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

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



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

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

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

`NDSolve::eerr:`

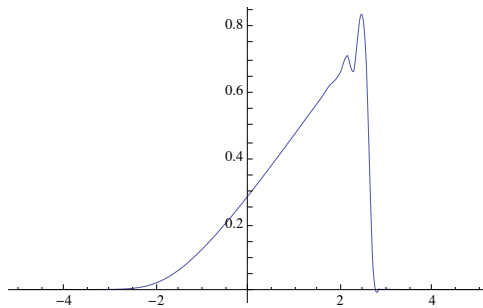
Warning: Scaled local spatial error estimate of 272.7279341590405` at  $t = 4.$  in the direction of independent variable  $x$  is much greater than prescribed error tolerance. Grid spacing with 75 points may be too large to achieve the desired accuracy or precision. A singularity may have formed or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

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

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

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

`Out[47]=`



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

Controlling the Spatial Grid Selection

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

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

Tensor product grid options for the method of lines.

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

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

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

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

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

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

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

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

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

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

NDSolve::eerr:

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

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

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

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

```
In[50]:= v = 0.01;
bsol3 = First[NDSolve[{D[u[x, t], t] == v D[u[x, t], x, x] - u[x, t] D[u[x, t], x],
  u[x, 0] == e-x2, u[-5, t] == u[5, t]}, u, {x, -5, 5},
  {t, 0, 4}, Method -> {"MethodOfLines", "SpatialDiscretization" ->
    {"TensorProductGrid", "MaxStepSize" -> (10 v / 2) / ((1.1) 851/4)}}]]]
Out[51]= {u -> InterpolatingFunction[{{-5., 5.}, {0., 4.}}, <>]}
```

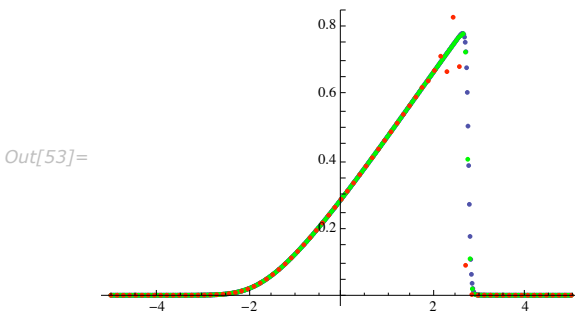
To compare solutions like this, it is useful to look at a plot of the solution only at the spatial grid points. Because the grid points are stored as a part of the `InterpolatingFunction`, it is fairly simple to define a function that does this.

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

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

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

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



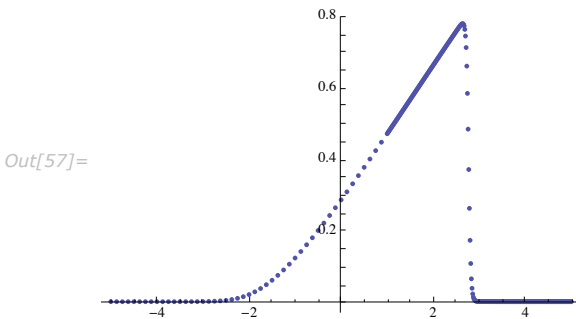
In this example, the left-hand side of the domain really does not need so many points. The points need to be clustered where the steep profile evolves, so it might make sense to consider explicitly specifying a grid that has more points where the profile appears.

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

```
In[54]:= mygrid = Join[-5. + 10 Range[0, 48] / 80, 1. + Range[1, 4 × 70] / 70];
v = 0.01;
bsolg = First[NDSolve[
  {D[u[x, t], t] == v D[u[x, t], x, x] - u[x, t] D[u[x, t], x], u[x, 0] == e-x2,
  u[-5, t] == u[5, t]}, u, {x, -5, 5}, {t, 0, 4}, Method → {"MethodOfLines",
  "SpatialDiscretization" → {"TensorProductGrid", "Coordinates" → {mygrid}}}]
Out[56]= {u → InterpolatingFunction[{{-5., 5.}, {0., 4.}}, <>]}
```

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

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



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

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

```
In[58]:= v = 0.075;
sol1 = First[NDSolve[{D[u[t, x, y], t] == v (D[u[t, x, y], x, x] + D[u[t, x, y], y, y]) -
  u[t, x, y] (2 D[u[t, x, y], x] - D[u[t, x, y], y]),
  u[0, x, y] == Exp[-(x2 + y2)], u[t, -4, y] == u[t, 4, y],
  u[t, x, -4] == u[t, x, 4]}, u, {t, 0, 2}, {x, -4, 4}, {y, -4, 4}]]
```

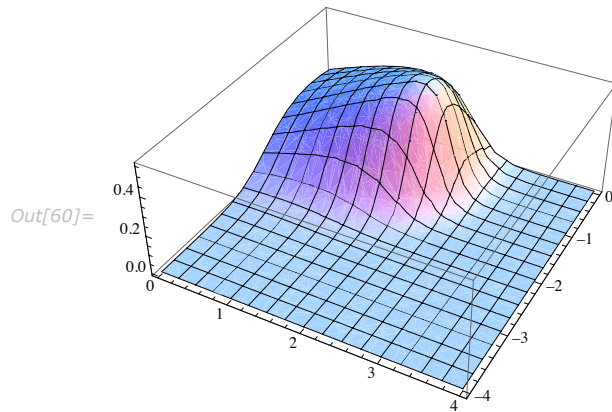
NDSolve::eerr:

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

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

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

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



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

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

```
In[61]:= ν = 0.075;
sol2 =
First[NDSolve[{D[u[t, x, y], t] == ν (D[u[t, x, y], x, x] + D[u[t, x, y], y, y]) -
    u[t, x, y] (2 D[u[t, x, y], x] - D[u[t, x, y], y]), u[0, x, y] == Exp[-(x^2 + y^2)],
    u[t, -4, y] == u[t, 4, y], u[t, x, -4] == u[t, x, 4]}, u, {t, 0, 2},
    {x, -4, 4}, {y, -4, 4}, Method -> {"MethodOfLines", "SpatialDiscretization" ->
    {"TensorProductGrid", "MinPoints" -> Ceiling[{1, 1/√2} 69 311/4]}}]]

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

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

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

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

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

```
In[63]:= v = 0.075;
xgrid = Join[Select[Part[u /. sol1, 3, 2], Negative],
  {0.}, Select[Part[u /. sol2, 3, 2], Positive]];
ygrid = Join[Select[Part[u /. sol2, 3, 3], Negative], {0.},
  Select[Part[u /. sol1, 3, 3], Positive]]; sol3 =
First[NDSolve[{D[u[t, x, y], t] == v (D[u[t, x, y], x, x] + D[u[t, x, y], y, y]) -
  u[t, x, y] (2 D[u[t, x, y], x] - D[u[t, x, y], y]), u[0, x, y] == Exp[-(x^2 + y^2)],
  u[t, -4, y] == u[t, 4, y], u[t, x, -4] == u[t, x, 4]}, u, {t, 0, 2},
  {x, -4, 4}, {y, -4, 4}, Method -> {"MethodOfLines", "SpatialDiscretization" ->
    {"TensorProductGrid", "Coordinates" -> {xgrid, ygrid}}]]]
Out[65]= {u -> InterpolatingFunction[{{0., 2.}, {-4., 4.}, {-4., 4.}}, <>]}
```

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

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

```

In[66]:= v = 0.01;
solutions = Map[Table[
  n = 2i + 1;
  u /.
  First[NDSolve[{D[u[x, t], t] == v D[u[x, t], x, x] - u[x, t] D[u[x, t], x],
    u[x, 0] == Exp[-x2], u[-5, t] == u[5, t]}, u, {x, -5, 5}, {t, 4, 4},
    AccuracyGoal → 10, PrecisionGoal → 10, MaxSteps → Infinity,
    Method → {"MethodOfLines", "SpatialDiscretization" →
      {"TensorProductGrid", "DifferenceOrder" → #, AccuracyGoal → 0,
        PrecisionGoal → 0, "MaxPoints" → n, "MinPoints" → n}}]],
  {i, 5, If[NumberQ[#], 12, 11]}
] &, {2, 4, 6, "Pseudospectral"}];

```

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

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

```

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

```

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

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

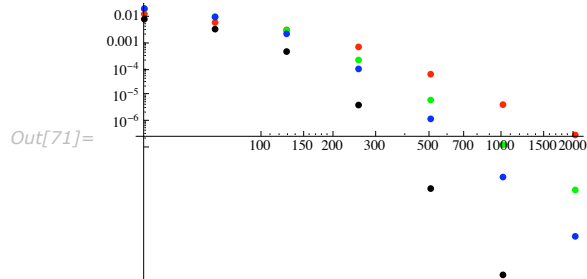
```

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

```



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



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

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

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

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

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

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

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

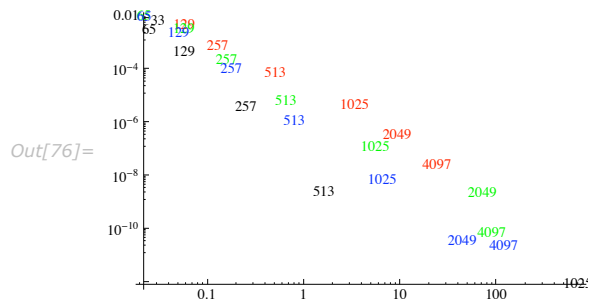
```
In[74]:= TimeAccuracy[do_][sol_] := Block[{tol, ag, n, solt, Second = 1},
  tol = errfun[4, sol, best];
  ag = -Log[10., tol];
  If[ag < 2,
    $Failed,
    n = Length[sol[[3, 1]]];
    secs = First[Timing[solt = First[
      u /. NDSolve[{D[u[x, t], t] == v D[u[x, t], x, x] - u[x, t] D[u[x, t],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

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

```
In[75]:= results =
  MapThread[Map[TimeAccuracy[#1], #2] &, {{2, 4, 6, "Pseudospectral"}, solutions}]
Out[75]= {{ $Failed, $Failed, {129, 2, 0.06, 0.00432122}, {257, 2, 0.12, 0.000724265},
  {513, 2, 0.671, 0.0000661853}, {1025, 2, 1.903, 4.44696×10-6},
  {2049, 2, 5.879, 3.10464×10-7}, {4097, 2, 17.235, 2.4643×10-8}},
  { $Failed, {65, 4, 0.02, 0.00979942}, {129, 4, 0.1, 0.00300281}, {257, 4, 0.161, 0.000213248},
  {513, 4, 1.742, 6.02345×10-6}, {1025, 4, 5.438, 1.13695×10-7},
  {2049, 4, 43.793, 2.10218×10-9}, {4097, 4, 63.551, 6.48318×10-11}},
  { $Failed, {65, 6, 0.03, 0.00853295}, {129, 6, 0.14, 0.00212781},
  {257, 6, 0.37, 0.0000935051}, {513, 6, 1.392, 1.1052×10-6}, {1025, 6, 7.14, 6.38732×10-9},
  {2049, 6, 35.121, 3.22349×10-11}, {4097, 6, 89.809, 2.15934×10-11}},
  {{33, Pseudospectral, 0.02, 0.00610004}, {65, Pseudospectral, 0.03, 0.00287949},
  {129, Pseudospectral, 0.08, 0.000417946}, {257, Pseudospectral, 0.22, 3.72935×10-6},
  {513, Pseudospectral, 2.063, 2.28232×10-9}, {1025, Pseudospectral, 544.974, 8.81844×10-13}}
```

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

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



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

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

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

Error at the Boundaries

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

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

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

```
In[2]:= solution = First[NDSolve[{ $\partial_t u[x, t] == \partial_{x,x} u[x, t]$ ,  $u[x, 0] == 1 - \frac{\text{Sin}[4 \pi x]}{4 \pi}$ ,  
u[0, t] == 1,  $u[1, t] + u^{(1,0)}[1, t] == 0$ }, u, {x, 0, 1}, {t, 0, 1}]]
```

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

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

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

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

```
In[3]:= solution =  
First[NDSolve[{ $\partial_t u[x, t] == \partial_{x,x} u[x, t]$ ,  $u[x, 0] == 1 - \frac{\text{Sin}[4 \pi x]}{4 \pi}$ ,  $u[0, t] == 1$ ,  
u[1, t] +  $u^{(1,0)}[1, t] == 0$ }, u, {x, 0, 1}, {t, 0, 1}, Method -> {"MethodOfLines",  
"SpatialDiscretization" -> {"TensorProductGrid", MinPoints -> 50}}]]
```

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

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

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

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

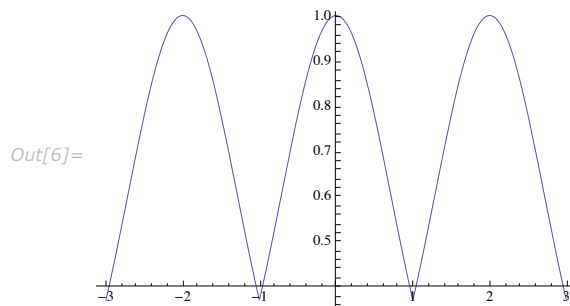
NDSolve::mxsst: Using maximum number of grid points 10000
allowed by the MaxPoints or MinStepSize options for independent variable x.

Out[5]= $Aborted
```

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

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

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



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

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

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

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

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

Numerical Solution of Boundary Value Problems

"Shooting" Method

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

Consider the BVP system

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

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

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

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

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

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

Linearization and Newton's Method

Linear problems can be described by

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

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

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

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

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

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

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

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

for any particular initial condition c_0 .

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

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

"StartingInitialConditions"

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

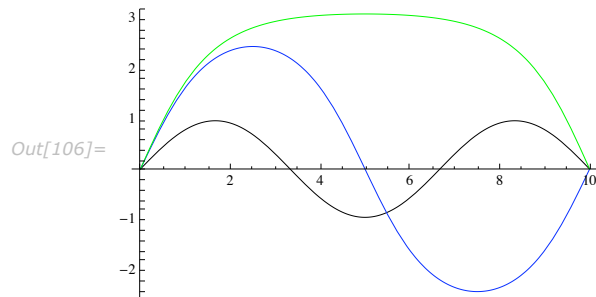
"StartingInitialConditions" is an option of the "Shooting" method that allows you to specify the values and position of the initial conditions to start the shooting process from.

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

This computes a very simple solution to the boundary value problem

$$x'' + \sin(x) = 0 \text{ with } x(0) = x(10) = 0.$$

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



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

Consider the linear boundary value problem

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

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

that has a solution

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

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

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

```
In[107]:= eqn =
  x''''[t] - 2 λ x'''[t] - λ² x'[t] + 2 λ³ x[t] == (λ² + π²) (2 λ Cos[π t] + π Sin[π t]);
bcs = {x[0] == 1 +  $\frac{1 + e^{-2\lambda} + e^{-\lambda}}{2 + e^{-\lambda}}$ , x[1] == 0, x'[1] ==  $\frac{3\lambda - e^{-\lambda}\lambda}{2 + e^{-\lambda}}$ };
xsol[t_] =  $\frac{e^{\lambda(t-1)} + e^{2\lambda(t-1)} + e^{-\lambda t}}{2 + e^{-\lambda}} + \text{Cos}[\pi t];$ 
```

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

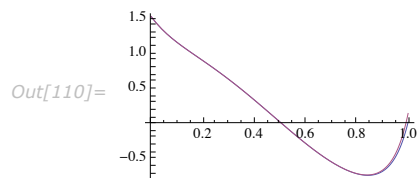
```
In[110]:= Block[{λ = 10},
  sol = First[NDSolve[{eqn, bcs}, x, t]];
  Plot[{xsol[t], x[t] /. sol}, {t, 0, 1}]]
```

NDSolve::bvluc:

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

NDSolve::berr:

There are significant errors $\{-1.11022 \times 10^{-16}, 6.95123 \times 10^{-6}, 0.000139029\}$ in the boundary value residuals. Returning the best solution found.

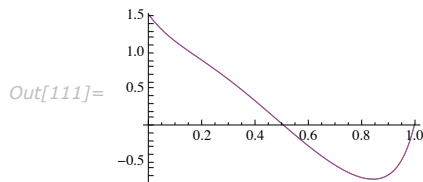


Shooting from $t=0$, the “Shooting” method gives warnings about an ill-conditioned matrix, and further that the boundary conditions are not satisfied as well as they should be. This is because a small error at $t=0$ is amplified by $e^{20} \approx 4 \times 10^8$. Since the reciprocal of this is of the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

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

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

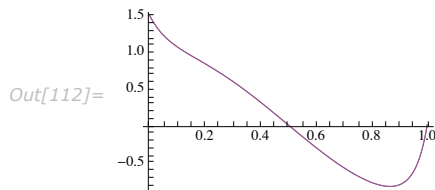
```
In[111]:= Block[{λ = 10},
  sol = First[NDSolve[{eqn, bcs}, x, t,
    Method → {"Shooting", "StartingInitialConditions" →
      {x[1] == 0, x'[1] ==  $\frac{3\lambda - e^{-\lambda}}{2 + e^{-\lambda}}$ , x''[1] == 0}}]];
  Plot[{xsol[t], x[t] /. sol}, {t, 0, 1}]]
```



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

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

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



Option summary

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

"Shooting" method options.

"Chasing" Method

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

Consider the linear ODE

$$X'(t) == A(t) X(t) + A_0(t)$$

(2)

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

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

(3)

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

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

(4)

where

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

The chasing method amounts to finding a vector function $\Phi_i(t)$ such that $\Phi_i(t_i) = \bar{B}_i$ and $\Phi_i(t) \bar{X}(t) = 0$.

Once the function $\Phi_i(t)$ is known, if there is a full set of boundary conditions, solving

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

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

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

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

Substituting the differential equation,

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

and transposing

$$\bar{X}(t) (\Phi_i'(t) + \bar{A}^T(t) \Phi_i(t)) = 0$$

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

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

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

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

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

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

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

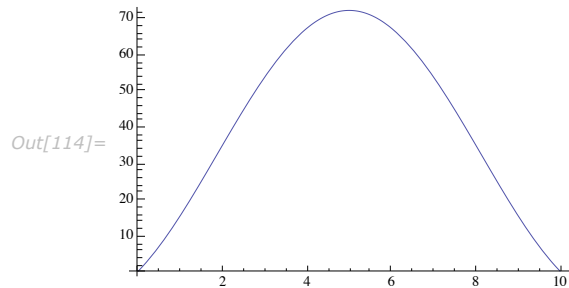
where $\tilde{A}$ is A with the j^{th} column removed and A_j is the j^{th} column of A . The nonlinear method can be more numerically stable than the linear method, but it has the disadvantage that integration along the real line may lead to singularities. This problem can be eliminated by integrating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

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

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

This shows a plot of the solution.

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



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

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

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

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

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

```
In[117]:= bconds /. First[nsol2]
Out[117]= {True, False, False, False}
```

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

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

```
In[118]:= Apply[Subtract, bconds, 1] /. First[nsol2]
Out[118]= {0., -2.5751×10-7, -4.13357×10-8, -2.95508×10-8}
```

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

This is a boundary value problem that has no solution.

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

NDSolve::bvluc:

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

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

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

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

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

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

DSolve::bvsing:

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

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

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

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

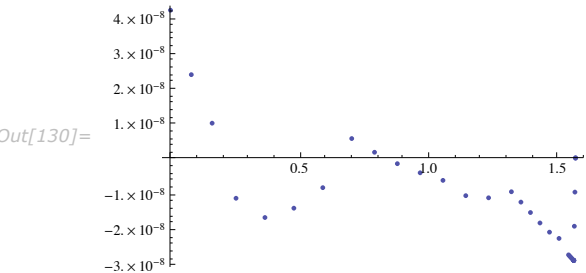
NDSolve::bvluc:

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

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

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

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



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

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

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

The method “ChasingType” -> “NonlinearChasing” itself has two options.

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

Options for the “NonlinearChasing” option of the “Chasing” method.

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

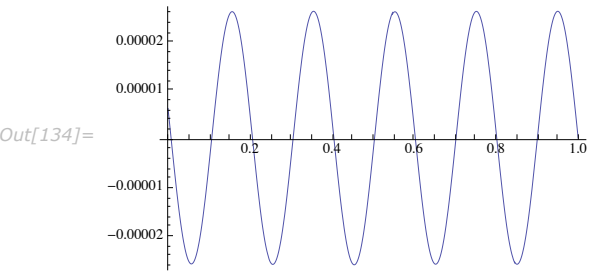
Here is a boundary value problem with a simple solution computed symbolically using DSolve.

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

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

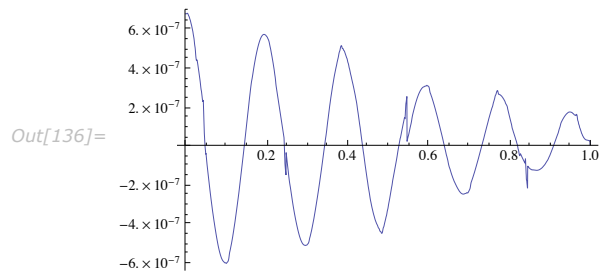
This shows the error in the solution computed using the chasing method in NDSolve.

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



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

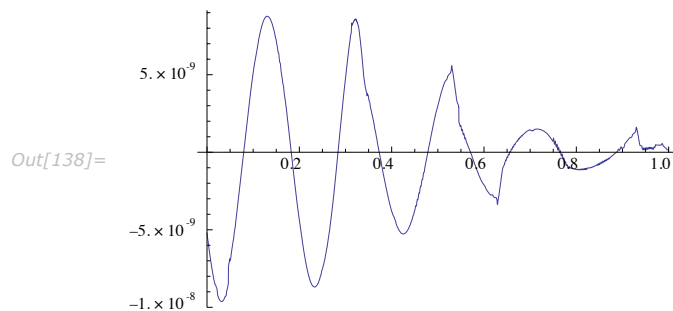
```
In[135]:= sol = First[x /. NDSolve[{x''[t] + 1000 x[t] == 0, x[0] == 0, x[1] == 1},
    x, {t, 0, 1}, Method -> {"Chasing", "ExtraPrecision" -> 10}]];
Plot[sol[t] - dsol[t], {t, 0, 1}]
```



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

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

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



Boundary Value Problems with Parameters

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

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

$$f''' - R((f')^2 - ff'') + Ra = 0$$

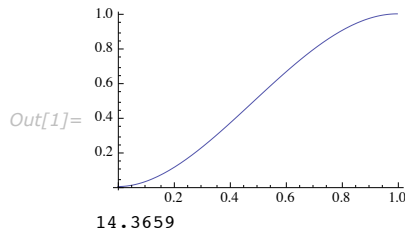
$$f(0) = f'(0) = 0, f(1) = 1, f'(1) = 0$$

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

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

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

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



Numerical Solution of Differential-Algebraic Equations

Introduction

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

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

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

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

$$F(t, x, x') = 0, \quad (7)$$

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

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

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

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

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

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

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

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

$$\text{In[11]:= DAE} = \left(\begin{array}{l} x_1'[t] = x_3[t] \\ x_2[t] (1 - x_2[t]) = 0 \\ x_1[t] x_2[t] + x_3[t] (1 - x_2[t]) = t \end{array} \right);$$

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

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

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

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

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

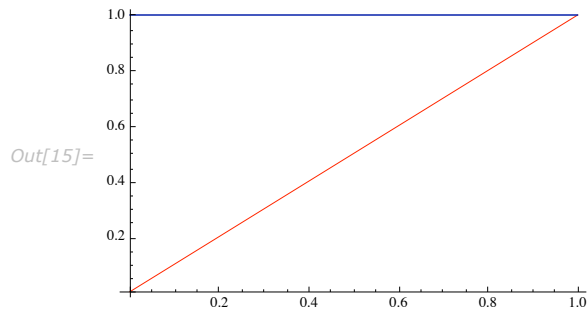
This shows the initial conditions found by `NDSolve`.

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

```
Out[13]= {{1.}, {0., 1., 1.}}
```

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

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



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

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

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

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

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

This shows the initial conditions found by `NDSolve`.

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

```
Out[17]= {{0.}, {0., 1., 0.}}
```

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

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

```
In[18]:= DAE /. x2[t] -> 1
```

```
Out[18]= {{x1'[t] == x3[t]}, {True}, {x1[t] == t}}
```

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

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

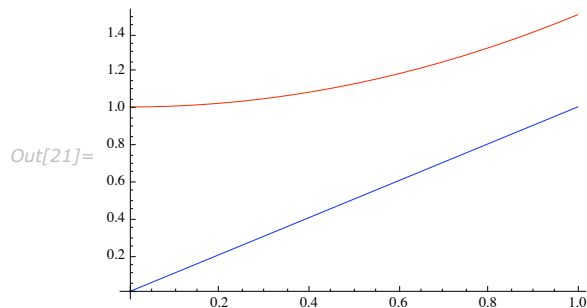
The other set of solutions for the problem is when $x_2[t] = 0$. You can find these by specifying that as an initial condition.

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

```
In[19]:= sol0 = NDSolve[{DAE, x1[0] == 1, x2[0] == 0}, {x1, x2, x3}, {t, 0, 1}]
Out[19]= {{x1 -> InterpolatingFunction[{{0., 1.}}, <>],
           x2 -> InterpolatingFunction[{{0., 1.}}, <>], x3 -> InterpolatingFunction[{{0., 1.}}, <>]}}
```

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

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



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

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

NDSolve fails to find a consistent initial condition.

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

NDSolve::icfail:

Unable to find initial conditions that satisfy the residual function within specified tolerances.

Try giving initial conditions for both values and derivatives of the functions.

```
Out[22]= {}
```

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

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

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

```
In[23]:= kinetics =
```

$$\left\{ y_1'[t] = -\frac{1}{25} y_1[t] + 10^4 y_2[t] y_3[t], y_2'[t] = \frac{1}{25} y_1[t] - 3 \times 10^7 y_2[t]^2 \right\};$$

$$\text{balance} = y_1[t] + y_2[t] + y_3[t] = 1;$$

$$\text{start} = \{y_1[0] = 1, y_2[0] = 0, y_3[0] = 0\};$$

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

```
In[26]:= odesol =
```

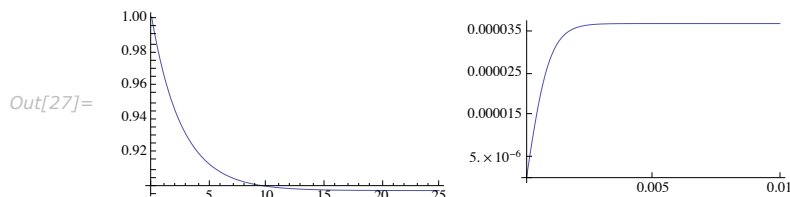
```
First[NDSolve[{kinetics, D[balance, t], start}, {y1, y2, y3}, {t, 0, 40000}]]
```

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

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

This shows the solutions y_1 and y_2 .

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



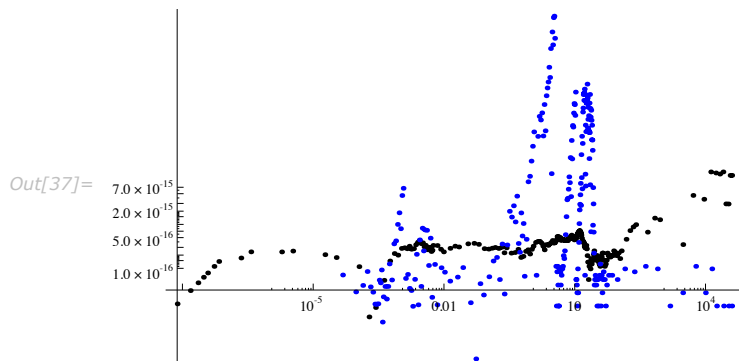
This solves the Robertson kinetics problem as a DAE.

```
In[33]:= daesol = First[NDSolve[{kinetics, balance, start}, {y1, y2, y3}, {t, 0, 40 000}]]
Out[33]= {y1 -> InterpolatingFunction[{{0., 40 000.}}, <>],
  y2 -> InterpolatingFunction[{{0., 40 000.}}, <>], y3 -> InterpolatingFunction[{{0., 40 000.}}, <>]}
```

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

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

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



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

You may want to solve some DAEs of the form

$$\begin{aligned}x'(t) &= f(t, x(t)) \\ g(t, x(t)) &= 0,\end{aligned}$$

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

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

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

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

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

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

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

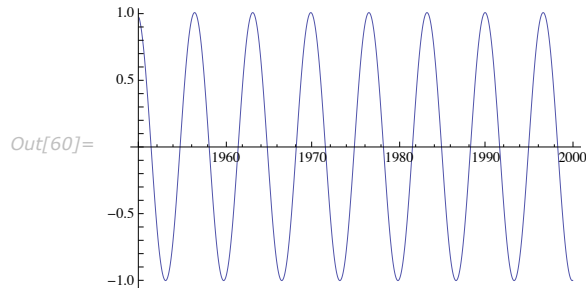
However, this solution may not be quite what you expect: the invariant equation has the solution $x[t] == \text{constant}$ when it starts with $x'[t] == 0$. In fact it does not have unique solutions from this starting point. This is because if you do actually solve for x' , the function does not satisfy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

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

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

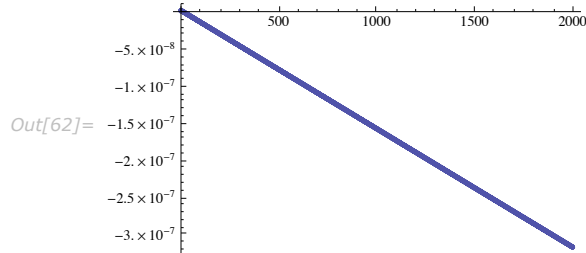
This shows the solution plotted over the last several periods.

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



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

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



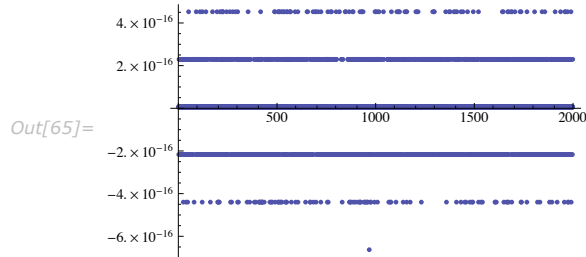
The error in the invariant is not large, but it does show a steady and consistent drift. Eventually, it could be large enough to affect the fidelity of the solution.

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

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

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

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



IDA Method for NDSolve

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

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

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

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

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

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

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

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

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

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

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

The values *prec* and *acc* are taken from the `NDSolve` settings for the `PrecisionGoal` $\rightarrow$ *prec* and `AccuracyGoal` $\rightarrow$ *acc*.

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

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

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

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

IDA method options.

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

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

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

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

This defines the linear equations along with initial conditions.

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

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

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

```
Out[4]= e^{2 t} \left( \cos\left[\sqrt{2} t\right] - \sqrt{2} \sin\left[\sqrt{2} t\right] \right)^2 + \frac{1}{4} e^{2 t} \left( 2 \cos\left[\sqrt{2} t\right] + \sqrt{2} \sin\left[\sqrt{2} t\right] \right)^2
```

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

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

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

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

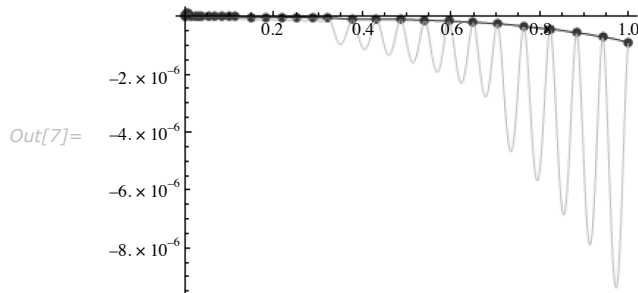
It can also be computed with dense output.

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

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

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

```
In[7]:= Needs["DifferentialEquations`InterpolatingFunctionAnatomy`"];
t1 = Cases[f1[t], (if_InterpolatingFunction)[t] →
  InterpolatingFunctionCoordinates[if], Infinity][[1, 1]];
pode = Show[Block[{$DisplayFunction = Identity},
  {ListPlot[Transpose[{t1, fexact[t1] - f1[t1]}], PlotStyle → PointSize[.02]],
  Plot[fexact[t] - f1[t], {t, 0, 1}, PlotStyle → RGBColor[.8, .8, .8]],
  Plot[fexact[t] - f1dense[t], {t, 0, 1}]}], PlotRange → All]
```



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

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

```
In[8]:= ByteCount /@ {f1[t], f1dense[t]}
Out[8]= {3560, 17 648}
```

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

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

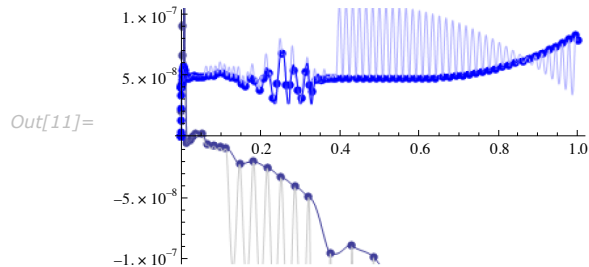
```
In[9]:= f2[t_] = First[g[t] /. NDSolve[{eqns, ics, g[t] == f[t]}, {x, y, g}, {t, 0, 1}]]
Out[9]= InterpolatingFunction[{{0., 1.}}, <>][t]
```

This computes the same solution with dense output.

```
In[10]:= f2dense[t_] = First[g[t] /. NDSolve[{eqns, ics, g[t] == f[t]},
  {x, y, g}, {t, 0, 1}, InterpolationOrder → All]]
Out[10]= InterpolatingFunction[{{0., 1.}}, <>][t]
```

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

```
In[11]:= t2 = InterpolatingFunctionCoordinates[Head[f2[t]]][[1]];
Show[{pode, ListPlot[Transpose[{t2, f_exact[t2] - f2[t2]}],
  PlotStyle -> {RGBColor[0, 0, 1], PointSize[0.02]}],
  Plot[f_exact[t] - f2[t], {t, 0, 1}, PlotStyle -> RGBColor[.7, .7, 1]],
  Plot[f_exact[t] - f2_dense[t], {t, 0, 1}, PlotStyle -> RGBColor[0, 0, 1]],
  PlotRange -> {{0, 1}, 1*^-7 {-1, 1}}]
```



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

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

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

Automatic

this is the default, automatically switch between using the *Mathematica* LinearSolve and Band methods depending on the band width of the Jacobian; for systems with larger band width, this will automatically switch to a direct sparse solver for large systems with sparse Jacobians

“Band”

use the IDA band method (see the IDA user manual for more information)

“Dense”

use the IDA dense method (see the IDA user manual for more information)

Possible settings for the “LinearSolveMethod” option.

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

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

“GMRES” method options.

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

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

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

This defines a function that satisfies Burger’s equation.

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

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

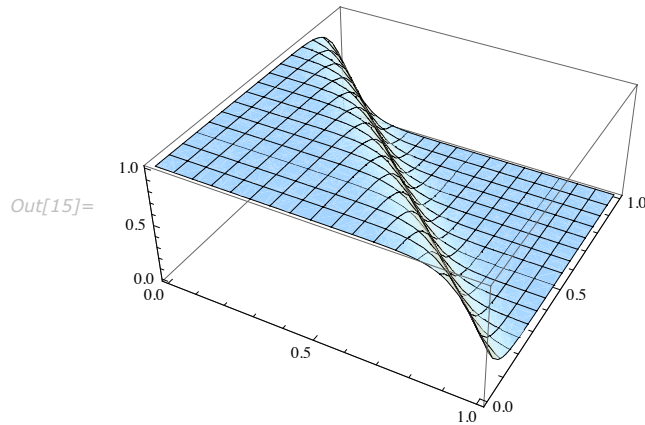
```
In[13]:= ic = u[0, x, y] == Bsol[0, x, y];
bc = {
  u[t, 0, y] == Bsol[t, 0, y], u[t, 1, y] == Bsol[t, 1, y],
  u[t, x, 0] == Bsol[t, x, 0], u[t, x, 1] == Bsol[t, x, 1]};
```

This defines the differential equation.

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


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

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



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

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

```
In[16]:= Timing[sol = NDSolve[{de, ic, bc}, u, {t, 0, 1}, {x, 0, 1}, {y, 0, 1},  
Method -> {"MethodOfLines", "DifferentiateBoundaryConditions" -> False}]]  
Out[16]= {2.233, {{u -> InterpolatingFunction[{{0., 1.}, {0., 1.}, {0., 1.}}, <>]}}}
```

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

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

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

This computes the maximal error for the computed solution.

```
In[18]:= errfun[sol]  
Out[18]= 0.000749446
```

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

This defines a function for comparing different IDA option settings.

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

No options use the previous defaults.

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

This uses the "Band" method.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
In[26]:= DM = NDSolve`FiniteDifferenceDerivative[{2, 0}, {X, Y}]@"DifferentiationMatrix" +
          NDSolve`FiniteDifferenceDerivative[{0, 2}, {X, Y}]@"DifferentiationMatrix";
          Short[diag = Tr[DM, List]]
Out[26]//Short= {18750., 6250., 3125., 3125., <<2593>>, 3125., 3125., 6250., 18750.}
```

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

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

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

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

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

```
In[29]:= TimeSolution["ImplicitSolver" → {"GMRES", "Preconditioner" → pfun}]
Out[29]= {1.161, 0.000716006, 88 Steps}
```

Thus, even with a crude preconditioner, the "GMRES" method computes the solution faster than the using the direct sparse solvers.

For PDE discretizations with higher-order temporal derivatives or systems of PDEs, you may need to look at the corresponding `NDSolve`StateData` object to determine how the variables are ordered so that you can get the structural form of the preconditioner correctly.

Delay Differential Equations

A delay differential equation is a differential equation where the time derivatives at the current time depend on the solution and possibly its derivatives at previous times:

$$\begin{cases} X'(t) = F(t, X(t), X(t - \tau_1), \dots, X(t - \tau_n), X'(t - \sigma_1), \dots, X'(t - \sigma_m)); & t \geq t_0 \\ X(t) = \phi(t); & t \leq t_0 \end{cases}$$

Instead of a simple initial condition, an initial history function $\phi(t)$ needs to be specified. The quantities $\tau_i \geq 0$, $i = 1, \dots, n$ and $\sigma_i \geq 0$, $i = 1, \dots, k$ are called the delays or time lags. The delays may be constants, functions $\tau(t)$ and $\sigma(t)$ of t (time dependent delays), or functions $\tau(t, X(t))$ and $\sigma(t, X(t))$ (state dependent delays). Delay equations with delays σ of the derivatives are referred to as neutral delay differential equations (NDDEs).

The equation processing code in `NDSolve` has been designed so that you can input a delay differential equation in essentially mathematical notation.

<code>x[t-τ]</code>	dependent variable x with delay τ
<code>x[t /; t ≤ t₀] == ϕ</code>	specification of initial history function as expression ϕ for t less than t_0

Inputting delays and initial history.

Currently, the implementation for DDEs in `NDSolve` only supports constant delays.

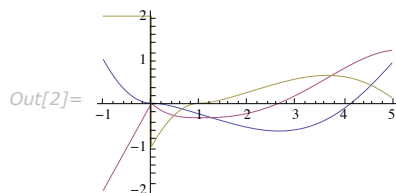
Solve a second order delay differential equation.

```
In[1]:= sol = NDSolve[{x''[t] + x[t - 1] == 0, x[t /; t ≤ 0] == t^2}, x, {t, -1, 5}]
```

```
Out[1]= {{x → InterpolatingFunction[{{-1., 5.}}, <>]}}
```

Plot the solution and its first two derivatives.

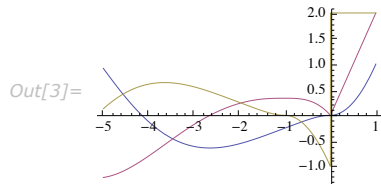
```
In[2]:= Plot[Evaluate[{x[t], x'[t], x''[t]} /. First[sol]], {t, -1, 5}, PlotRange → All]
```



For simplicity, this documentation is written assuming that integration always proceeds from smaller to larger t . However, `NDSolve` supports integration in the other direction if the initial history function is given for value above t_0 and the delays are negative.

Solve a second order delay differential equation in the direction of negative t .

```
In[3]:= nsol = NDSolve[{x''[t] + x[t + 1] == 0, x[t /; t ≥ 0] == t^2}, x, {t, -5, 1}];
Plot[Evaluate[{x[t], x'[t], x''[t]} /. First[nsol]], {t, -5, 1}, PlotRange → All]
```

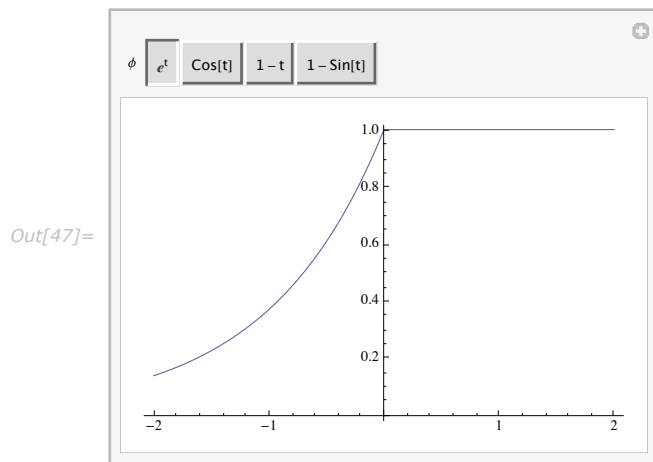


Comparison and Contrast with ODEs

While DDEs look a lot like ODEs the theory for them is quite a bit more complicated and there are some surprising differences with ODEs. This section will show a few examples of the differences.

Look at the solutions of $x'(t) = x(t-1)(x(t)-1)$ for different initial history functions.

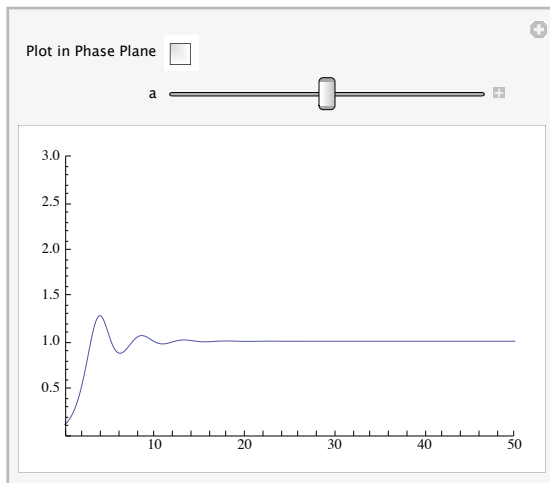
```
In[47]:= Manipulate[
Module[
{sol = NDSolve[{x'[t] == x[t - 1] (1 - x[t])}, x[t /; t ≤ 0] == φ}, x, {t, -2, 2}},
Plot[Evaluate[x[t] /. First[sol]], {t, -2, 2}],
{φ, {Exp[t], Cos[t], 1 - t, 1 - Sin[t]}}
```



As long as the initial function satisfies $\phi(0)=1$, the solution for $t>0$ is always 1. [Z06] With ODEs, you could always integrate backwards in time from a solution to obtain the initial condition.

Investigate at the solutions of $x'(t) = a x(t) (1 - x(t - 1))$ for different values of the parameter a .

```
In[1]:= Manipulate[
  Module[{T = 50, sol, x, t}, sol = First[x /. NDSolve[
    {x'[t] == a x[t] (1 - x[t - 1]), x[t /; t ≤ 0] == 0.1}, x, {t, 0, T}]];
  If[pp, ParametricPlot[{sol[t], sol[t - 1]}, {t, 1, T},
    PlotRange → {{0, 3}, {0, 3}}],
    Plot[sol[t], {t, 0, T}, PlotRange → {{0, 50}, {0, 3}}]],
  {{pp, False, "Plot in Phase Plane"}, {False, True}}, {{a, 1}, 0, 2}]
```



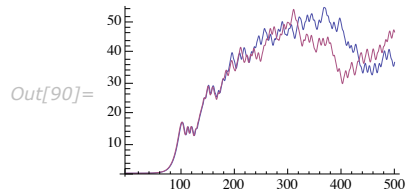
For $a < \frac{1}{e}$, the solutions are monotonic, for $\frac{1}{e} \leq a \leq \frac{\pi}{2}$ the solutions oscillate. and for $a > \frac{\pi}{2}$ the solutions approach a limit cycle. Of course, for the scalar ODE, solutions are monotonic independent of a .

Solve the Ikeda delay differential equation, $x'(t) = \sin(x(t - 2\pi))$ for two nearby constant initial functions.

```
In[88]:= sol1 = First[NDSolve[{x'[t] == Sin[x[t - 20]], x[t /; t ≤ 0] == .0001}, x, {t, 0, 500}]];
sol2 = First[NDSolve[{x'[t] == Sin[x[t - 20]], x[t /; t ≤ 0] == .00011},
  x, {t, 0, 500}]];
```

Plot the solutions.

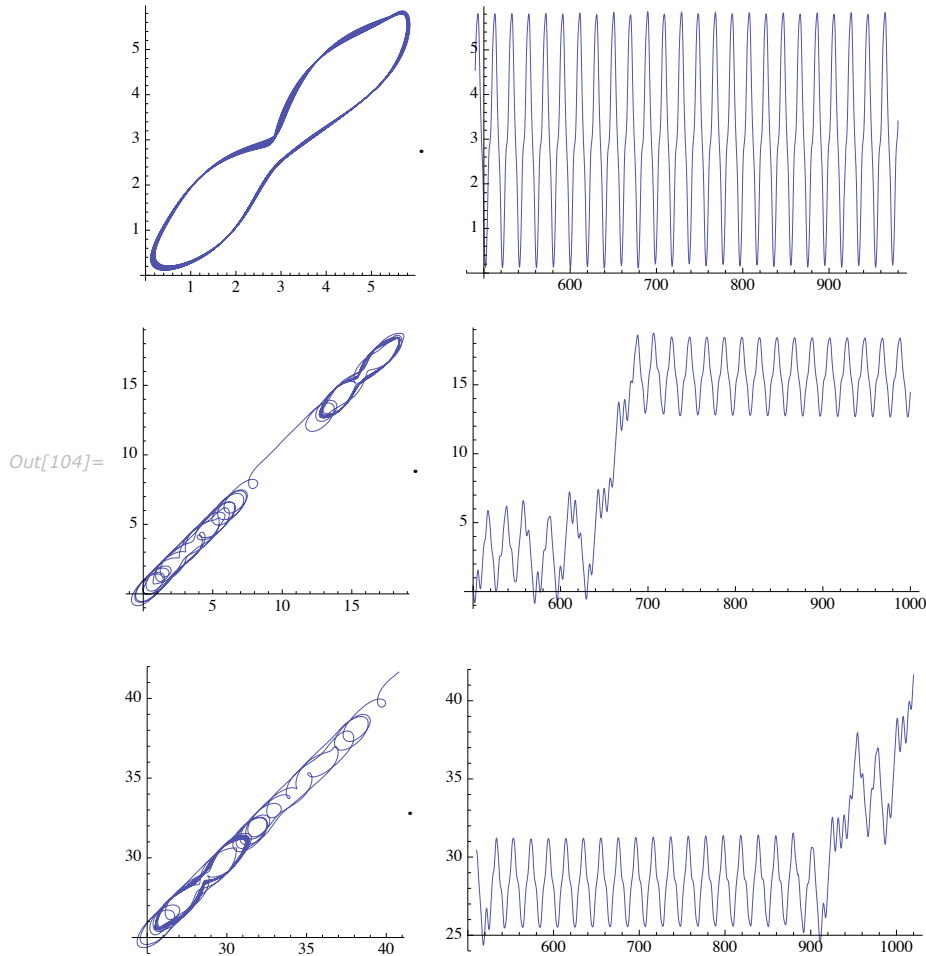
```
In[90]:= Plot[Evaluate[x[t] /. {sol1, sol2}], {t, 0, 500}]
```



This simple scalar delay differential equation has chaotic solutions and the motion shown above looks very much like Brownian motion. [S07] As the delay τ is increased beyond $\tau = \pi/2$ a limit cycle appears, followed eventually by a period doubling cascade leading to chaos before $\tau = 5$.

Compare solutions for $\tau=4.9$, 5.0, and 5.1

```
In[104]:= Grid[Table[sol = First[NDSolve[{x'[t] == Sin[x[t -  $\tau$ ]}, x[t /; t ≤ 0] == .1],
    x, {t, 100  $\tau$ , 200  $\tau$ }, MaxSteps → Infinity]];
    {ParametricPlot[Evaluate[{x[t - 1], x[t]} /. sol], {t, 101  $\tau$ , 200  $\tau$ }}.
    Plot[Evaluate[x[t] /. sol], {t, 100  $\tau$ , 200  $\tau$ }}], { $\tau$ , 4.9, 5.1, .1}]
```

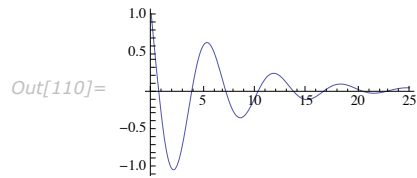


Stability is much more complicated for delay equations as well. It is well known that the linear ODE test equation $x'(t) = \lambda x(t)$ has asymptotically stable solutions if $\text{Re}(\lambda) < 0$ and is unstable if $\text{Re}(\lambda) > 0$.

The closest corresponding DDE is $x'(t) = \lambda x(t) + \mu x(t-1)$. Even if you consider just real λ and μ the situation is no longer so clear cut. Shown below are some plots of solutions indicating this.

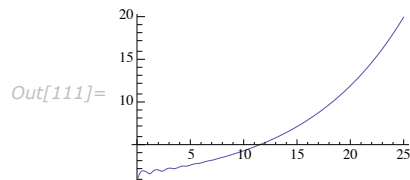
The solution is stable with $\lambda = \frac{1}{2}$ and $\mu = -1$

```
In[110]:= Block[{λ = 1 / 2, μ = -1, T = 25}, Plot[
  Evaluate[First[x[t]] /. NDSolve[{x'[t] == λ x[t] + μ x[t - 1], x[t /; t ≤ 0] == 1 - t},
    x, {t, 0, T}]]], {t, 0, T}, PlotRange → All]]
```



The solution is unstable with $\lambda = -\frac{7}{2}$ and $\mu = 4$

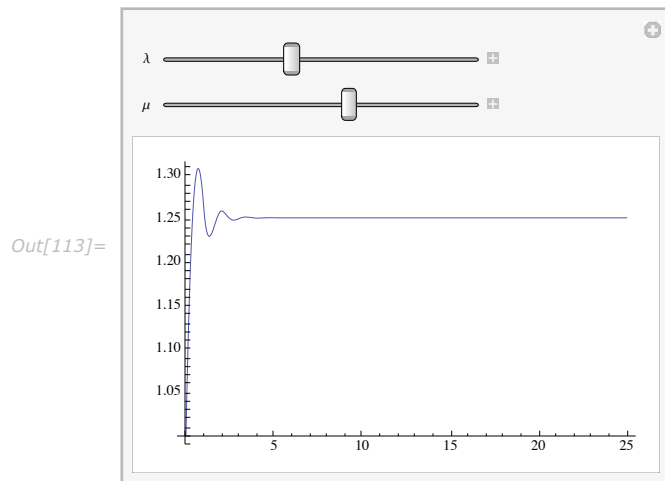
```
In[111]:= Block[{λ = -7 / 2, μ = 4, T = 25}, Plot[
  Evaluate[First[x[t]] /. NDSolve[{x'[t] == λ x[t] + μ x[t - 1], x[t /; t ≤ 0] == 1 - t},
    x, {t, 0, T}]]], {t, 0, T}, PlotRange → All]]
```



So the solution can be stable with $\lambda > 0$ and unstable with $\lambda < 0$ depending on the value of μ . A Manipulate is set up below so that you can investigate the λ - μ plane.

Investigate by varying λ and μ

```
In[113]:= Manipulate[Module[{T = 25, x, t}, Plot[Evaluate[First[x[t]] /.
  NDSolve[{x'[t] == λ x[t] + μ x[t - 1], x[t /; t ≤ 0] == 1 - t}, x, {t, 0, T}]]],
  {t, 0, T}, PlotRange → All]], {λ, -5, 5}, {μ, -5, 5}]
```



Propagation and Smoothing of Discontinuities

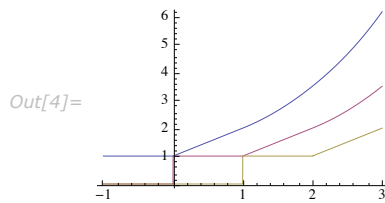
The way discontinuities are propagated by the delays is an important feature of DDEs and has a profound effect on numerical methods for solving them.

Solve $x'(t) = x(t-1)$ with $x(t) = 1$ for $t \leq 0$.

```
In[3]:= sol = First[NDSolve[{x'[t] == x[t-1], x[t /; t ≤ 0] == 1}, x, {t, -1, 3}]]
```

```
Out[3]= {x → InterpolatingFunction[{{-1., 3.}}, <>]}
```

```
In[4]:= Plot[Evaluate[{x[t], x'[t], x''[t]} /. sol], {t, -1, 3}]
```



In the example above, $x(t)$ is continuous, but there is a jump discontinuity in $x'(t)$ at $t=0$ since approaching from the left the value is 0, given by the derivative of the initial history function $x'(t) = \phi'(t) = 0$ while approaching from the right the value is given by the DDE, giving $x'(t) = x(t-1) = \phi(t-1) = 1$.

Near $t=1$, we have by the continuity of x at 0 $\lim_{t \rightarrow 1^-} x'(t) = \lim_{t \rightarrow 1^-} x(t-1) = \lim_{z \rightarrow 0^-} x(z) = \lim_{z \rightarrow 0^+} x(z) = \lim_{t \rightarrow 1^+} x'(t)$ and so $x'(t)$ is continuous at $t=1$.

Differentiating the equation, we can conclude that $(x')'(t) = x'(t-1)$ so $(x')'(t)$ has a jump discontinuity at $t=1$. Using essentially the same argument as above, we can conclude that at $t=2$ the second derivative is continuous.

Similarly, $x^{(k)}(t)$ is continuous at $t=k$ or, in other words, at $t=k$, $x(t)$ is k times differentiable. This is referred to as smoothing and holds generally for non-neutral delay equations. In some cases the smoothing can be faster than one order per interval.[Z06]

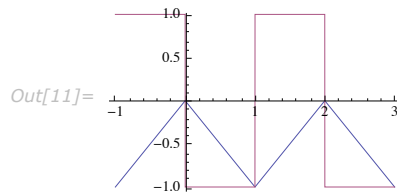
For neutral delay equations the situation is quite different.

Solve $x'(t) = x'(t-1)$ with $x(t) = -t$ for $t \leq 0$.

```
In[10]:= sol = First[NDSolve[{x'[t] == -x'[t-1], x[t /; t ≤ 0] == t}, x, {t, -1, 3}]]
```

```
Out[10]= {x → InterpolatingFunction[{{-1., 3.}}, <>]}
```

```
In[11]:= Plot[Evaluate[{x[t], x'[t]} /. sol], {t, -1, 3}]
```



It is easy to see that the solution is piecewise with $x[t]$ continuous. However,

$$x'(t) = \begin{cases} -1 & 0 < \text{mod}(t, 2) < 1 \\ 1 & 1 < \text{mod}(t, 2) < 2 \end{cases}$$

which has a discontinuity at every non negative integer.

In general, there is no smoothing of discontinuities for neutral DDEs.

The propagation of discontinuities is very important from the standpoint of numerical solvers. If the possible discontinuity points are ignored, then the order of the solver will be reduced. If a discontinuity point is known a more accurate solution can be found by integrating just up to the discontinuity point and then restarting the method just past the point with the new function values. This way, the integration method is used on smooth parts of the solution leading to better accuracy and fewer rejected steps. From any given discontinuity points, future discontinuity points can be determined from the delays and detected by treating them as events to be located.

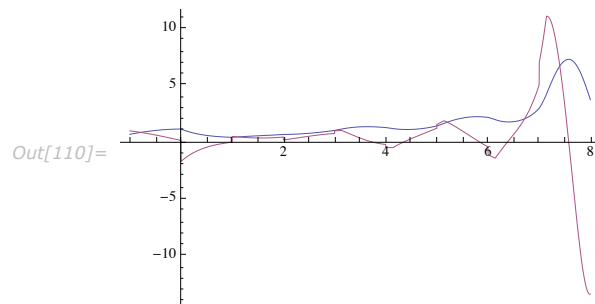
When there are multiple delays, the propagation of discontinuities can become quite complicated.

Solve a neutral delay differential equation with two delays.

```
In[109]:= sol =
  NDSolve[{x'[t] == x[t] (x[t - Pi] - x'[t - 1]), x[t /; t ≤ 0] == Cos[t]}, x, {t, -1, 8}]
Out[109]= {{x → InterpolatingFunction[{{-1., 8.}}, <>]}}
```

Plot the solution.

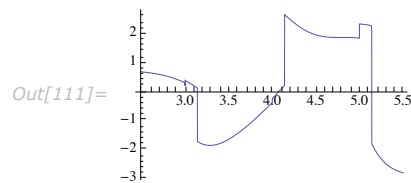
```
In[110]:= Plot[Evaluate[{x[t], x'[t]} /. First[sol]], {t, -1, 8}, PlotRange -> All]
```



It is clear from the plot that there is a discontinuity at each non negative integer as would be expected from the neutral delay $\sigma = 1$. However, looking at the second and third derivative, it is clear that there are also discontinuities associated with points like $t = \pi, 1 + \pi, 2 + \pi$ propagated from the jump discontinuities in $x'(t)$.

Plot the second derivative

```
In[111]:= Plot[Evaluate[x''[t] /. First[sol]], {t, 2.5, 5.5}, PlotRange -> All]
```



In fact, there is a whole tree of discontinuities that are propagated forward in time. A way of determining and displaying the discontinuity tree for a solution interval is shown in the subsection below.

Discontinuity Tree

Define a command that gives the graph for the propagated discontinuities for a DDE with the given delays

```
In[112]:= DiscontinuityTree[t0_, Tend_, delays_] :=
Module[{dt, next, ord},
ord[t_] := Infinity;
ord[t0] = 0;
next[b_, order_, del_] := Map[dt[b, #, order, del] &, del];
dt[t_, {d_, nq_}, order_, del_] := Module[{b = t + d},
If[b ≤ Tend,
o = order + Boole[! nq];
ord[b] = Min[ord[b], o];
 Sow[{t → b, d}];
 next[b, o, del]]];
rules = Reap[next[t0, 0, delays]] [[2, 1]];
rules = Tally[rules] [[All, 1]];
f[x_?NumericQ] := {x, ord[x]};
f[a_ → b_] := f[a] → f[b];
rules[[All, 1]] = Map[f, rules[[All, 1]]];
rules]
```

Get the discontinuity tree for the example above up to $t = 8$.

```
In[113]:= tree = Tally[DiscontinuityTree[0, 8, {{1, True}, {π, False}}]] [[All, 1]]

Out[113]= {{{0, 0} → {1, 0}, 1}, {{1, 0} → {2, 0}, 1}, {{2, 0} → {3, 0}, 1}, {{3, 0} → {4, 0}, 1},
{{4, 0} → {5, 0}, 1}, {{5, 0} → {6, 0}, 1}, {{6, 0} → {7, 0}, 1}, {{7, 0} → {8, 0}, 1},
{{4, 0} → {4 + π, 1}, π}, {{3, 0} → {3 + π, 1}, π}, {{3 + π, 1} → {4 + π, 1}, 1},
{{2, 0} → {2 + π, 1}, π}, {{2 + π, 1} → {3 + π, 1}, 1}, {{1, 0} → {1 + π, 1}, π},
{{1 + π, 1} → {2 + π, 1}, 1}, {{1 + π, 1} → {1 + 2 π, 2}, π}, {{0, 0} → {π, 1}, π},
{{π, 1} → {1 + π, 1}, 1}, {{π, 1} → {2 π, 2}, π}, {{2 π, 2} → {1 + 2 π, 2}, 1}}
```

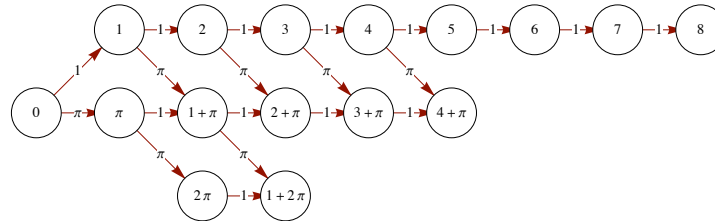
Define a command that shows a plot of $x^{(k)}(t)$ and $x^{(k+1)}(t)$ for a discontinuity of order k .

```
In[116]:= ShowDiscontinuity[{dt_, o_}, ifun_, Δ_] :=
Quiet[
Plot[Evaluate[{Derivative[o][ifun][t], Derivative[o + 1][ifun][t]}], {t, dt - Δ,
dt + Δ}, Exclusions → {dt}, ExclusionsStyle → Red, Frame → True, FrameLabel →
{None, None, {Derivative[o][x][t], Derivative[o + 1][x][t]}, None}]]
```

Plot as a layered graph, showing the discontinuity plot as a tooltip for each discontinuity.

```
In[117]:= LayeredGraphPlot[tree, Left, VertexLabeling -> True, VertexRenderingFunction ->
  Function[Tooltip[{White, EdgeForm[Black], Disk[#, .3], Black, Text[#2[[1]], #1]},
    ShowDiscontinuity[#2, First[x /. sol], 1]]]]
```

Out[117]=



Storing History Data

Once the solution has advanced beyond the first discontinuity point, some of the delayed values that need to be computed are outside of the domain of the initial history function and the computed solution needs to be used to get the values, typically by interpolating between steps previously taken. For the DDE solution to be accurate it is essential that the interpolation be as accurate as the method. This is achieved by using dense output for the ODE integration method (the output you get if you use the option `InterpolationOrder -> All` in `NDSolve`). `NDSolve` has a general algorithm for obtaining dense output from most methods, so you can use just about any method as the integrator. Some methods, including the default for DDEs have their own way of getting dense output which is usually more efficient than the general method. Methods that are low enough order, such as “`ExplicitRungeKutta`” with “`DifferenceOrder`” -> 3 can just use a cubic Hermite polynomial as the dense output so there is essentially no extra cost in keeping the history.

Since the history data is accessed frequently, it needs to have a quick look up mechanism to determine which step to interpolate within. In `NDSolve`, this is done with a binary search mechanism and the search time is negligible compared with the cost of actual function evaluation.

The data for each successful step is saved before attempting the next step and is saved in a data structure that can repeatedly be expanded efficiently. When `NDSolve` produces the solution, it simply takes this data and restructures it into an `InterpolatingFunction` object, so DDE solutions are always returned with dense output.

The Method of Steps

For constant delays, it is possible to get the entire set of discontinuities as fixed time. The idea of the method of steps is to simply integrate the smooth function over these intervals and restart on the next interval, being sure to reevaluate the function from the right. As long as the intervals do not get too small, the method works quite well in practice.

The method currently implemented for `NDSolve` is based on the method of steps.

Symbolic method of steps

This section defines a symbolic method of steps that illustrates how the method works. Note that to keep the code simpler and more to the point, it does not do any real argument checking. Also, the data structure and look up for the history is not done in an efficient way, but for symbolic solutions this is a minor issue.

Use `DSolve` to integrate over an interval where the solution is smooth.

```
In[16]:= IntegrateSmooth[rhs_, history_, delayvars_, pfun_, dvars_, {t_, t0_, t1_}] :=
Module[{delayvals, dvt, tau, hrule, dvrule, dvrules, oderhs, ode, init, sol},
  dvt[tau_] = Map[# [tau] &, dvars];
  hrule[pos_] :=
    Thread[dvars -> Map[Function[Evaluate[{t}], #] &, history[[pos]]]];
  dvrule[(dv_) [z_]] := Module[{delay, pos},
    delay = t - z;
    pos = pfun[t0 - delay];
    dv[z] -> (dv[z] /. hrule[pos])];
  dvrules = Map[dvrule, delayvars];
  oderhs = rhs /. dvrules;
  ode = Thread[D[dvt[t], t] == oderhs];
  init = Thread[dvt[t0] == (dvt[t0] /. hrule[-1])];
  sol = DSolve[{ode, init}, dvars, t];
  If[Head[sol] === DSolve || Length[sol] == 0,
    Message[DDESteps::stuck, ode, init];
    Throw[$Failed]];
  dvt[t] /. First[sol]
];
DDESteps::stuck =
  "DSolve was not able to find a solution for `1` with initial conditions `2`.";
```


Define a method of steps function that returns Piecewise functions.

```
In[21]:= DDESteps[rhsin_, phin_, dvarsin_, {t_, tinit_, tend_}] :=
Module[{rhs = Listify[rhsin], phi = Listify[phin], dvars = Listify[dvarsin],
  history, delayvars, delays, dtree, intervals, p, pfun, next, pieces, hfun},
  history = {phi};
  delayvars = Cases[rhs, (v : ((dv_[z_] | Derivative[1][dv_][z_]) /;
    (MemberQ[dvars, dv] && UnsameQ[z, t]))) -> {v, t - z}, Infinity];
  {delayvars, delays} = Map[Union, Transpose[delayvars]];
  dtree = DiscontinuityTree[tinit, tend, Map[{#, True} &, delays]];
  dtree = Union[Flatten[{tinit, tend, dtree[[All, 1, 2, 1]]}]];
  dtree = SortBy[dtree, N];
  intervals = Partition[dtree, 2, 1];
  p = 2;
  pfun =
  Join[{1, t < tinit}], Apply[Function[{p++, #1 ≤ t < #2}], intervals, {1}]];
  pfun = Function[Evaluate[{t}], Evaluate[Piecewise[pfun, p]]];
  Catch[Do[
    next = IntegrateSmooth[rhs,
      history, delayvars, pfun, dvars, Prepend[interval, t]];
    history = Append[history, next],
    {interval, intervals}]];
  pieces =
  Flatten[{t < tinit, Apply[{#1 ≤ t < #2} &, Drop[intervals, -1], {1}],
    Apply[{#1 ≤ t ≤ #2} &, Last[intervals]}]];
  pieces = Take[pieces, Length[history]];
  hfun = Map[Function[Evaluate[{t}], Evaluate[Piecewise[
    Transpose[{#, pieces}], Indeterminate]] &, Transpose[history]];
  Thread[dvars -> hfun]
];
Listify[x_List] := x;
Listify[x_] := {x};
```

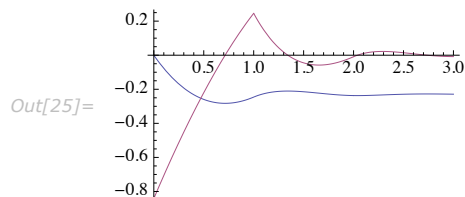
Find the solution for the DDE $x'(t) = x(t-1) - x(t)$ with $\phi(t) = \sin(t)$.

```
In[24]:= sol = DDESteps[x[t - 1] - x[t], Sin[t], x, {t, 0, 3}]
```

```
Out[24]= {x -> Function[{t},
{
  Sin[t]                                                                 t < 0
  - 1/2 e^-t (-Cos[1] + e^t Cos[1 - t] - Sin[1] + e^t Sin[1 - t])      0 ≤ t
  - 1/2 e^-t (e - Cos[1] - e t Cos[1] + e^t Cos[2 - t] - Sin[1] + e Sin[1] - e t Sin[1]) 1 ≤ t
  - 1/4 e^-t (2 e - 2 e^2 + 2 e^2 t - 2 Cos[1] - e^2 Cos[1] - 2 e t Cos[1] +
    2 e^2 t Cos[1] - e^2 t^2 Cos[1] + e^t Cos[3 - t] - 2 Sin[1] + 2 e Sin[1] -
    3 e^2 Sin[1] - 2 e t Sin[1] + 4 e^2 t Sin[1] - e^2 t^2 Sin[1] - e^t Sin[3 - t]) 2 ≤ t
  Indeterminate                                                         True
}]
```

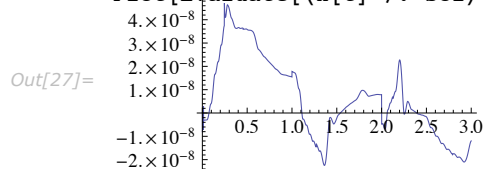
Plot the solution.

```
In[25]:= Plot[Evaluate[{x[t], x'[t]} /. sol], {t, 0, 3}]
```



Check the quality of the solution found by `NDSolve` by comparing to the exact solution.

```
In[26]:= ndsol = First[NDSolve[{x'[t] == -x[t] + x[t - 1], x[t /; t ≤ 0] == Sin[t]}, x, {t, 0, 3}]];
Plot[Evaluate[(x[t] /. sol) - (x[t] /. ndsol)], {t, 0, 3}, PlotRange → All]
```



The method will also work for neutral DDEs.

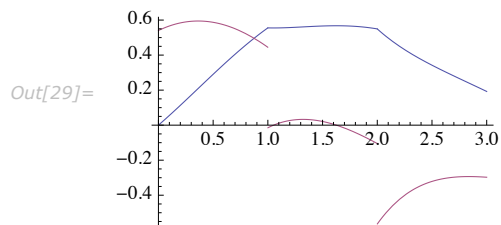
Find the solution for the neutral DDE $x'(t) = x'(t-1) - x(t)$ with $\phi(t) = \sin(t)$.

```
In[28]:= sol = DDESteps[x'[t - 1] - x[t], Sin[t], x, {t, 0, 3}]
```

```
Out[28]= {x → Function[{t},
  { Sin[t]                                                                 t < 0
    1/2 e^-t (-Cos[1] + e^t Cos[1 - t] + Sin[1] - e^t Sin[1 - t])        0 ≤ t
    1/2 e^-t (e - Cos[1] - 2 e Cos[1] + e t Cos[1] + e^t Cos[2 - t] + Sin[1] + e Sin[1] - e t Sin[1]) 1 ≤ t
    1/4 e^-t (2 e + 6 e^2 - 2 e^2 t - 2 Cos[1] - 4 e Cos[1] - 13 e^2 Cos[1] +
      2 e t Cos[1] + 8 e^2 t Cos[1] - e^2 t^2 Cos[1] + e^t Cos[3 - t] + 2 Sin[1] + 2 e Sin[1] +
      7 e^2 Sin[1] - 2 e t Sin[1] - 6 e^2 t Sin[1] + e^2 t^2 Sin[1] + e^t Sin[3 - t]) 2 ≤ t
    Indeterminate                                                         True
  ]}]
```

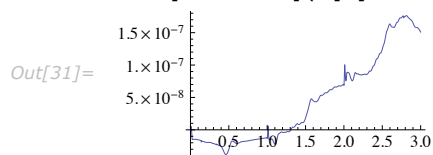
Plot the solution.

```
In[29]:= Plot[Evaluate[{x[t], x'[t]} /. sol], {t, 0, 3}]
```



Check the quality of the solution found by `NDSolve` by comparing to the exact solution.

```
In[30]:= ndsol = First[NDSolve[{x'[t] == -x[t] + x'[t - 1], x[t /; t ≤ 0] == Sin[t]}, x, {t, 0, 3}]];
Plot[Evaluate[(x[t] /. sol) - (x[t] /. ndsol)], {t, 0, 3}, PlotRange → All]
```



The symbolic method will also work with symbolic parameter values as long as `DSolve` is able to still able to find the solution.

Find the solution to a simple linear DDE with symbolic coefficients.

```
In[32]:= sol = DDESteps[λ x[t] + μ x[t - 1], t, x, {t, 0, 2}]
```

```
Out[32]= {x → Function[{t},
  {
    t, t < 0
    - (1 - et λ - λ + et λ λ + t λ) μ, 0 ≤ t < 1
    -  $\frac{e^{-λ} μ (e^{t λ} λ - e^{λ + t λ} λ + e^{λ + t λ} λ^2 - e^λ μ + e^{t λ} μ + e^λ λ μ - e^{t λ} t λ μ - e^{t λ} λ^2 μ + e^{t λ} t λ^2 μ)}{λ^3}$ , 1 ≤ t ≤ 2
    Indeterminate, True
  }
}
```

The reason the code was designed to take lists was so that it would work with systems

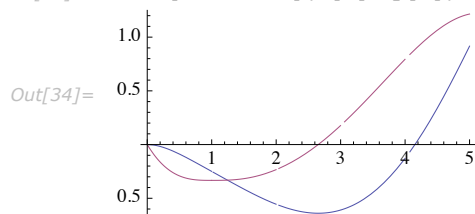
Solve a system of DDEs.

```
In[33]:= ssol = DDESteps[{y[t], -x[t - 1]}, {t^2, 2 t}, {x, y}, {t, 0, 5}]
```

```
Out[33]= {x → Function[{t},
  {
    t^2, t < 0
     $\frac{1}{12} (-6 t^2 + 4 t^3 - t^4)$ , 0 ≤ t
     $\frac{1}{360} (52 - 216 t + 165 t^2 - 140 t^3 + 60 t^4 - 12 t^5 + t^6)$ , 1 ≤ t
     $\frac{-3744 + 8640 t - 18088 t^2 + 11872 t^3 - 5040 t^4 + 1456 t^5 - 252 t^6 + 24 t^7 - t^8}{20160}$ , 2 ≤ t
     $\frac{1}{1814400} (804654 - 2371680 t + 2210265 t^2 - 1643400 t^3 + 771120 t^4 - 236376 t^5 + 51030 t^6 - 7560 t^7 + 720 t^8 - 40 t^9 + t^{10})$ , 3 ≤ t
     $\frac{1}{239500800} (-168512584 + 394727040 t - 534391836 t^2 + 359788000 t^3 - 165844800 t^4 + 55576224 t^5 - 13370280 t^6 + 2347488 t^7 - 300960 t^8 + 27280 t^9 - 1650 t^{10} + 60 t^{11} - t^{12})$ , 4 ≤ t
    Indeterminate, True
  },
  y → Function[{t},
    {
      2 t, t < 0
       $\frac{1}{3} (-3 t + 3 t^2 - t^3)$ , 0 ≤ t
       $\frac{1}{60} (-36 + 55 t - 70 t^2 + 40 t^3 - 10 t^4 + t^5)$ , 1 ≤ t
       $\frac{1080 - 4522 t + 4452 t^2 - 2520 t^3 + 910 t^4 - 189 t^5 + 21 t^6 - t^7}{2520}$ , 2 ≤ t
       $\frac{-237168 + 442053 t - 493020 t^2 + 308448 t^3 - 118188 t^4 + 30618 t^5 - 5292 t^6 + 576 t^7 - 36 t^8 + t^9}{181440}$ , 3 ≤ t
       $\frac{1}{19958400} (32893920 - 89065306 t + 89947000 t^2 - 55281600 t^3 + 23156760 t^4 - 6685140 t^5 + 1369368 t^6 - 200640 t^7 + 20460 t^8 - 1375 t^9 + 55 t^{10} - t^{11})$ , 4 ≤ t
      Indeterminate, True
    }
  ]
}
```

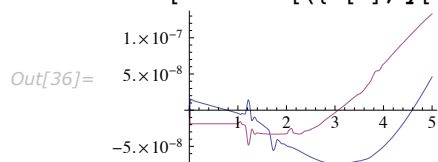
Plot the solution.

```
In[34]:= Plot[Evaluate[{x[t], y[t]} /. ssol], {t, 0, 5}]
```



Check the quality of the solution found by NDSolve by comparing to the exact solution.

```
In[35]:= ndssol = First[NDSolve[{x'[t] == y[t], y'[t] == -x[t - 1],
  x[t /; t ≤ 0] == t^2, y[t /; t ≤ 0] == 2 t}, {x, y}, {t, 0, 5}]];
Plot[Evaluate[{x[t], y[t]} /. ssol] - ({x[t], y[t]} /. ndssol), {t, 0, 5}]
```



Since the method computes the discontinuity tree, it will also work for multiple constant delays. However, with multiple delays, the solution may become quite complicated quickly and DSolve can bog down with huge expressions.

Solve a nonlinear neutral DDE with two delays.

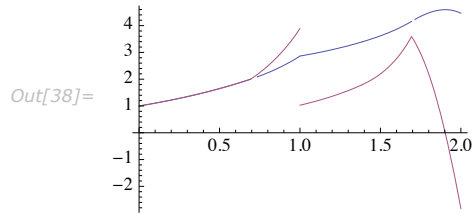
```
In[37]:= sol = DDESteps[x[t] (x[t - Log[2]] - x'[t - 1]), 1, x, {t, 0, 2}]
```

Out[37]=

$$\left\{ x \rightarrow \text{Function}\left[\{t\}, \begin{cases} 1 & t < 0 \\ e^t & 0 \leq t < \text{Log}[2] \\ 2 e^{-1 + \frac{e^t}{2}} & \text{Log}[2] \leq t < 1 \\ 2 e^{\frac{1}{2}(-2+e)} e^{-1+t} & 1 \leq t < 2 \text{Log}[2] \\ 2 e^{2 - e^{-1+t} - \frac{2 \text{ExpIntegralEi}[1]}{e} + \frac{2 \text{ExpIntegralEi}\left[\frac{e^t}{4}\right]}{e}} & 2 \text{Log}[2] \leq t < 1 + \text{Log}[2] \\ 2 e^{2 - 2 e^{-1 + \frac{e^{-1+t}}{2} - \frac{2 \text{ExpIntegralEi}[1]}{e} - 2 \text{ExpIntegralEi}\left[\frac{1}{2}(-2+e)\right] + \frac{2 \text{ExpIntegralEi}\left[\frac{e^t}{2}\right]}{e} + 2 \text{ExpIntegralEi}\left[\frac{1}{4}(-2+e) e^{-1+t}\right]} & 1 + \text{Log}[2] \leq t \leq 2 \\ \text{Indeterminate} & \text{True} \end{cases} \right] \right\}$$

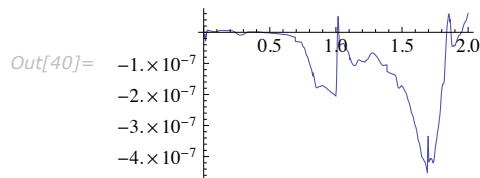
Plot the solution.

```
In[38]:= Plot[Evaluate[{x[t], x'[t]} /. sol], {t, 0, 2}]
```



Check the quality of the solution found by NDSolve by comparing to the exact solution.

```
In[39]:= ndsol = First[NDSolve[
  {x'[t] == x[t] (x[t - Log[2]] - x'[t - 1]), x[t /; t ≤ 0] == 1}, x, {t, 0, 2}]];
Plot[Evaluate[(x[t] /. sol) - (x[t] /. ndsol)], {t, 0, 2}, PlotRange -> All]
```



Examples

Lotka-Volterra equations with delay

The Lotka-Volterra system models the growth and interaction of animal species assuming that the effect of one species on another is continuous and immediate. A delayed effect of one species on another can be modeled by introducing time lags in the interaction terms.

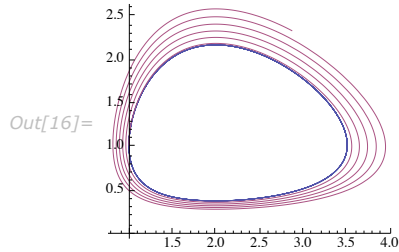
Consider the system

$$Y_1'(t) = Y_1(t) (Y_2(t - \tau_2) - 1), \quad Y_2'(t) = Y_2(t) (2 - Y_1(t - \tau_1)). \quad (9)$$

With no delays, $\tau_1 = \tau_2 = 0$ the system (1) has an invariant $H(t) = 2 \ln Y_1 - Y_1 + \ln Y_2 - Y_1$ that is constant for all t and there is a (neutrally) stable periodic solution.

Compare the solution with and without delays.

```
In[13]:= lvsystem[τ1_, τ2_] := {
  Y1'[t] == Y1[t] (Y2[t - τ1] - 1), Y1[0] == 1,
  Y2'[t] == Y2[t] (2 - Y1[t - τ2]), Y2[0] == 1};
lv = First[NDSolve[lvsystem[0, 0], {Y1, Y2}, {t, 0, 25}]];
lvd = Quiet[First[NDSolve[lvsystem[.01, 0], {Y1, Y2}, {t, 0, 25}]]];
ParametricPlot[Evaluate[{Y1[t], Y2[t]} /. {lv, lvd}], {t, 0, 25}]
```



In this example, the effect of even a small delay is to destabilize the periodic orbit. With different parameters in the delayed Lotka-Volterra system it has been shown that there are globally attractive equilibria.[TZ08]

Enzyme kinetics

Consider the system

$$\begin{aligned}
 y_1'(t) &= I_s - z y_1(t) \\
 y_2'(t) &= z y_1(t) - y_2(t) \\
 y_3'(t) &= y_2(t) - y_3(t) \\
 y_4'(t) &= y_3(t) - \frac{1}{2} y_4(t)
 \end{aligned}
 \quad z = \frac{k_1}{1 + \alpha (y_4(t - \tau))^n} \quad (10)$$

modeling enzyme kinetics where I_s is a substrate supply maintained at a constant level and n molecules of the end product y_4 inhibits the reaction step $y_1 \rightarrow y_2$. [HNW93]

The system has an equilibrium when $\{y_1 = I_s/z, y_2 = y_3 = I_s, y_4 = 2 I_s\}$.

Investigate solutions of (1) starting a small perturbation away from the equilibrium.

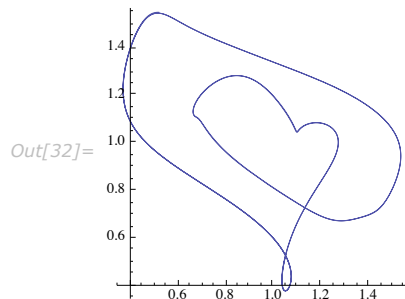
```
In[43]:= Manipulate[
Module[{t, y1, y2, y3, y4, z, sol},
  z = k1 / (1 +  $\alpha$  y4[t -  $\tau$ ]^n);
  sol = First[NDSolve[{
    y1'[t] == Is - z y1[t], y1[t /; t ≤ 0] == Is * (1 +  $\alpha$  (2 Is)^n) +  $\epsilon$ ,
    y2'[t] == z y1[t] - y2[t], y2[t /; t ≤ 0] == Is,
    y3'[t] == y2[t] - y3[t], y3[t /; t ≤ 0] == Is,
    y4'[t] == y3[t] - y4[t] / 2, y4[t /; t ≤ 0] == 2 Is},
    {y1, y2, y3, y4}, {t, 0, 200}]];
  Plot[Evaluate[{y1[t], y2[t], y3[t], y4[t]} /. sol], {t, 0, 200}]],
{{Is, 10.5}, 1, 20}, {{ $\alpha$ , 0.0005}, 0, .001}, {{k1, 1}, 0, 2},
{{n, 3}, 1, 10, 1}, {{ $\tau$ , 4}, 0, 10}, {{ $\epsilon$ , 0.1}, 0, .25}]
```

Mackey-Glass equation

The Mackey-Glass equation $x'[t] = a x[t-\tau] / (1 + x[t-\tau]^n) - b x[t]$ was proposed to model the production of white blood cells. There are both periodic and chaotic solutions.

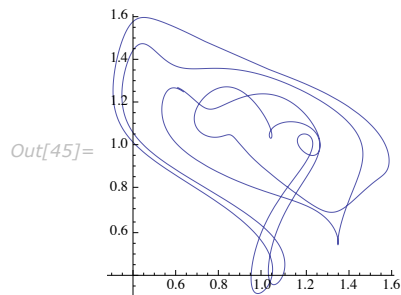
Here is a periodic solution of the Mackey-Glass equation. The plot is only shown after $t = 300$ to let transients die out.

```
In[31]:= sol = First[NDSolve[{x'[t] == (1/4) x[t - 15] / (1 + x[t - 15]^10) - x[t] / 10,
  x[t /; t ≤ 0] == 1/2}, x, {t, 0, 500}]];
ParametricPlot[Evaluate[{x[t], x[t - 15]} /. sol], {t, 300, 500}]
```



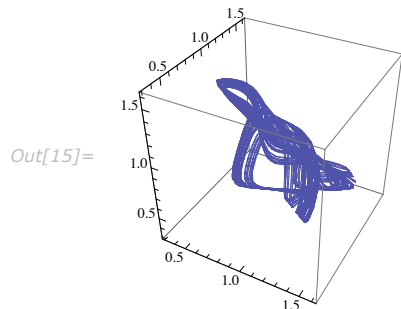
Here is a chaotic solution of the Mackey-Glass equation.

```
In[44]:= sol = First[NDSolve[{x'[t] == (1/4) x[t - 17] / (1 + x[t - 17]^10) - x[t] / 10,
  x[t /; t ≤ 0] == 1/2}, x, {t, 0, 500}]];
ParametricPlot[Evaluate[{x[t], x[t - 17]} /. sol], {t, 300, 500}]
```



This shows an embedding of the solution above in 3D $\{x(t), x(t - \tau), x(t - 2\tau)\}$.

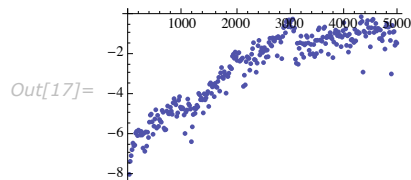
```
In[14]:= sol = First[NDSolve[{x'[t] == (1/4) x[t - 17] / (1 + x[t - 17]^10) - x[t] / 10,
    x[t] /; t <= 0 == 1/2}, x, {t, 0, 5000}, MaxSteps -> ∞]];
ParametricPlot3D[Evaluate[{x[t], x[t - 17], x[t - 34]} /. sol], {t, 500, 5000}]
```



It is interesting to check the accuracy of the chaotic solution.

Compute the chaotic solution with another method and plot $\log_{10} |d|$ for the difference d between $x(t)$ computed by the different methods.

```
In[16]:= solrk = First[NDSolve[{x'[t] == (1/4) x[t - 17] / (1 + x[t - 17]^10) - x[t] / 10,
    x[t] /; t <= 0 == 1/2}, x, {t, 0, 5000}, MaxSteps -> ∞,
    Method -> {"ExplicitRungeKutta", "DifferenceOrder" -> 3}]];
ListPlot[Table[{t, RealExponent[(x[t] /. sol) - (x[t] /. solrk)]},
    {t, 17, 5000, 17}]]
```



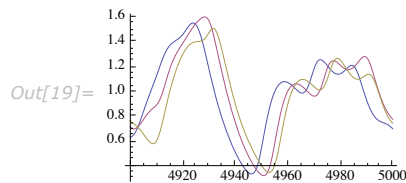
By the end of the interval, the differences between methods is order 1. Large deviation is typical in chaotic systems and in practice it is not possible or even necessary to get a very accurate solution for a large interval. However, if you do want a high quality solution, NDSolve allows you to use higher precision. For DDEs with higher precision, the "StiffnessSwitching" method is recommended.

Compute the chaotic solution with higher precision and tolerances.

```
In[18]:= hpsol = First[NDSolve[{x'[t] == (1/4) x[t - 17] / (1 + x[t - 17]^10) - x[t] / 10,
    x[t] /; t <= 0 == 1/2}, x, {t, 0, 5000}, MaxSteps -> ∞,
    Method -> "StiffnessSwitching", WorkingPrecision -> 32 ]];
```


Plot the three solutions near the final time.

`In[19]:= Plot[Evaluate[x[t] /. {hpsol, sol, solrk}], {t, 4900, 5000}, PlotRange -> All]`



Norms in NDSolve

NDSolve uses norms of error estimates to determine when solutions satisfy error tolerances. In nearly all cases the norm has been weighted, or scaled, such that it is less than 1 if error tolerances have been satisfied and greater than one if error tolerances are not satisfied. One significant advantage of such a scaled norm is that a given method can be written without explicit reference to tolerances: the satisfaction of tolerances is found by comparing the scaled norm to 1, thus simplifying the code required for checking error estimates within methods.

Suppose that v is vector and u is a reference vector to compute weights with (typically u is an approximate solution vector). Then the scaled vector w to which the norm is applied has components:

$$w_i = \frac{|v_i|}{t_a + t_r |u_i|} \tag{11}$$

where absolute and relative tolerances t_a and t_r are derived respectively from the AccuracyGoal $\rightarrow ag$ and PrecisionGoal $\rightarrow pg$ options by $t_a = 10^{-ag}$ and $t_r = 10^{-pg}$.

The actual norm used is determined by the setting for the NormFunction option given to NDSolve.

option name	default value	
NormFunction	Automatic	a function to use to compute norms of error estimates in NDSolve

NormFunction option to NDSolve.

The setting for the `NormFunction` option can be any function that returns a scalar for a vector argument and satisfies the properties of a norm. If you specify a function that does not satisfy the required properties of a norm, `NDSolve` will almost surely run into problems and give an answer, if any, which is incorrect.

The default value of `Automatic` means that `NDSolve` may use different norms for different methods. Most methods use an infinity-norm, but the IDA method for DAEs uses a 2-norm because that helps maintain smoothness in the merit function for finding roots of the residual. It is strongly recommended that you use `Norm` with a particular value of p . For this reason, you can also use the shorthand `NormFunction -> p` in place of `NormFunction -> (Norm[#, p] / Length[#] ^ (1 / p) &)`. The most commonly used implementations for $p=1$, $p=2$, and $p=\infty$ have been specially optimized for speed.

This compares the overall error for computing the solution to the simple harmonic oscillator over 100 cycles with different norms specified.

```
In[1]:= Map[
  First[{1 - x[100 π]} /. NDSolve[{x''[t] + x[t] == 0, x[0] == 1, x'[0] == 0}, x,
    {t, 0, 100 π}, Method -> ExplicitRungeKutta, NormFunction -> #]] &, {1, 2, ∞}]

Out[1]= {8.62652 × 10-8, 7.50564 × 10-8, 5.81547 × 10-8}
```

The reason that error decreases with increasing p is because the norms are normalized by multiplying with $1/n^{1/p}$, where n is the length of the vector. This is often important in `NDSolve` because in many cases, an attempt is being made to check the approximation to a function, where more points should give a better approximation, or less error.

Consider a finite difference approximation to the first derivative of a periodic function u given by $u'_i = \frac{u_{i+1} - u_i}{h}$ where $u_i = u(x_i)$ on a grid with uniform spacing $h = x_{i+1} - x_i$. In *Mathematica*, this can easily be computed using `ListCorrelate`.

This computes the error of the first derivative approximation for the cosine function on a grid with 16 points covering the interval $[0, 2\pi]$.

```
In[2]:= h = 2 π / 16.;
grid = h Range[16];
err16 = Sin[grid] - ListCorrelate[{1, -1] / h, Cos[grid], {1, 1}]

Out[2]= {-0.169324, -0.11903, -0.0506158, 0.0255046, 0.0977423, 0.1551, 0.188844, 0.193839,
  0.169324, 0.11903, 0.0506158, -0.0255046, -0.0977423, -0.1551, -0.188844, -0.193839}
```

This computes the error of the first derivative approximation for the cosine function on a grid with 32 points covering the interval $[0, 2\pi]$.

```
In[3]:= h = 2  $\pi$  / 32.;
grid = h Range[32];
err32 = Sin[grid] - ListCorrelate[{1, -1} / h, Cos[grid], {1, 1}]

Out[3]:= {-0.0947283, -0.0879564, -0.0778045, -0.0646625, -0.0490356, -0.0315243, -0.0128016, 0.00641315,
0.0253814, 0.0433743, 0.0597003, 0.0737321, 0.0849304, 0.0928648, 0.0972306, 0.0978598,
0.0947283, 0.0879564, 0.0778045, 0.0646625, 0.0490356, 0.0315243, 0.0128016, -0.00641315,
-0.0253814, -0.0433743, -0.0597003, -0.0737321, -0.0849304, -0.0928648, -0.0972306, -0.0978598}
```

It is quite apparent that the pointwise error is significantly less with a larger number of points.

The 2 norms of the vectors are of the same order of magnitude.

```
In[4]:= {Norm[err16, 2], Norm[err32, 2]}

Out[4]:= {0.552985, 0.392279}
```

The norms of the vectors are comparable because is because the number of components in the vector has increased, so the usual linear algebra norm does not properly reflect the convergence. Normalizing by multiplying by $1/n^{1/p}$ reflects the convergence in the function space properly.

The normalized 2 norms of the vectors reflect the convergence to the actual function. Since the approximation is first order, doubling the number of grid points should approximately halve the error.

```
In[5]:= {Norm[err16, 2] / Sqrt[16], Norm[err32, 2] / Sqrt[32]}

Out[5]:= {0.138246, 0.0693457}
```

Note that if you specify a function an option value, and you intend to use it for PDE or function approximation solutions, you should be sure to include a proper normalization in the function.

ScaledVectorNorm

Methods that have error control need to determine whether a step satisfies local error tolerances or not. To simplify the process of checking this, utility function `ScaledVectorNorm` does the scaling (1) and computes the norm. The table includes the formulas for specific values of p for reference.

<code>ScaledVectorNorm[p, {t_r, t_a}][v, u]</code>	compute the normalized p -norm of the vector v scaling using scaling (1) with reference vector u and relative and absolute tolerances t_a and t_r
<code>ScaledVectorNorm[fun, {t_r, t_a}][v, u]</code>	compute the norm of the vector v using scaling (1) with reference vector u and relative and absolute tolerances t_a and t_r and the norm function fun
<code>ScaledVectorNorm[2, {t_r, t_a}][v, u]</code>	compute $\sqrt{\frac{1}{n} \sum_{i=1}^n \left(\frac{ v_i }{t_a + t_r u_i } \right)^2}$ where n is the length of vectors v and u
<code>ScaledVectorNorm[∞, {t_r, t_a}][v, u]</code>	compute $\max\left(\frac{ v_i }{t_a + t_r u_i }\right)$, $1 \leq i \leq n$ where n is the length of vectors v and u

`ScaledVectorNorm`.

This sets up a scaled vector norm object with the default machine-precision tolerances used in `NDSolve`.

```
In[10]:= svn = NDSolve`ScaledVectorNorm[2, {10.-8, 10.-8}]
```

```
Out[10]= NDSolve`ScaledVectorNorm[2, {1.×10-8, 1.×10-8}]
```

This applies the scaled norm object with a sample error and solution reference vector.

```
In[11]:= svn[{9.×10.-9, 10.-8}, {2., 1.}]
```

```
Out[11]= 0.412311
```

Because of the absolute tolerance term, the value comes out reasonably even if some of the components of the reference solution are zero.

```
In[12]:= svn[{9.×10.-9, 10.-8, 2×10-8}, {1., 0., 0.}]
```

```
Out[12]= 1.31688
```

When setting up a method for `NDSolve`, you can get the appropriate `ScaledVectorNorm` object to use using the “`Norm`” method function of the `NDSolve`StateData` object.

Here is an `NDSolve`StateData` object.

```
In[13]:= state = First[NDSolve`ProcessEquations[{x'[t] + x[t] == 0, x[0] == 1, x'[0] == 0}, x, t]]
```

```
Out[13]= NDSolve`StateData[<0.>]
```

This gets the appropriate scaled norm to use from the state data.

```
In[14]:= svn = state["Norm"]
```

```
Out[14]= NDSolve`ScaledVectorNorm[ $\infty$ , { $1.05367 \times 10^{-8}$ ,  $1.05367 \times 10^{-8}$ }, NDSolve]
```

This applies it to a sample error vector using the initial condition as reference vector.

```
In[15]:= svn[{10.^-9, 10.^-8}, state@"SolutionVector"["Forward"]]
```

```
Out[15]= 0.949063
```

Stiffness Detection

Overview

Many differential equations exhibit some form of stiffness which restricts the step-size and hence effectiveness of explicit solution methods.

A number of implicit methods have been developed over the years to circumvent this problem.

For the same step size, implicit methods can be substantially less efficient than explicit methods, due to the overhead associated with the intrinsic linear algebra.

This cost can offset by the fact that, in certain regions, implicit methods can take substantially larger step sizes.

Several attempts have been made to provide user-friendly codes that automatically attempt to detect stiffness at runtime and switch between appropriate methods as necessary.

A number of strategies that have been proposed to automatically equip a code with a stiffness detection device are outlined here.

Particular attention is given to the problem of estimation of the dominant eigenvalue of a matrix in order to describe how stiffness detection is implemented in `NDSolve`.

Numerical examples illustrate the effectiveness of the strategy.

Initialization

Load some packages with predefined examples and utility functions.

```
In[1]:= Needs["DifferentialEquations`NDSolveProblems`"];
Needs["DifferentialEquations`NDSolveUtilities`"];
Needs["FunctionApproximations`"];
```

Introduction

Consider the numerical solution of initial value problems:

$$y'(t) = f(t, y(t)), \quad y(0) = y_0, \quad f: \mathbb{R} \times \mathbb{R}^n \mapsto \mathbb{R}^n \quad (12)$$

Stiffness is a combination of problem, solution method, initial condition and local error tolerances.

Stiffness limits the effectiveness of explicit solution methods due to restrictions on the size of steps that can be taken.

Stiffness arises in many practical systems as well as in the numerical solution of partial differential equations by the method of lines.

Example

The van der Pol oscillator is a non-conservative oscillator with nonlinear damping and is an example of a stiff system of ordinary differential equations:

$$\begin{aligned} y_1'(t) &= y_2(t), \\ \varepsilon y_2'(t) &= -y_1(t) + (1 - y_1(t)^2) y_2(t), \end{aligned}$$

with $\varepsilon = 3/1000$.

Consider initial conditions.

$$y_1(0) = 2, \quad y_2(0) = 0$$

and solve over the interval $t \in [0, 10]$.

The method "StiffnessSwitching" uses a pair of extrapolation methods by default:

- Explicit modified midpoint (Gragg smoothing), double-harmonic sequence 2, 4, 6,...
- Linearly implicit Euler, sub-harmonic sequence 2, 3, 4,...

Solution

This loads the problem from a package.

```
In[4]:= system = GetNDSolveProblem["VanderPol"];
```

Solve the system numerically using a nonstiff method.

```
In[5]:= solns = NDSolve[system, {T, 0, 10}, Method -> "Extrapolation"];
```

NDSolve::ndstf:

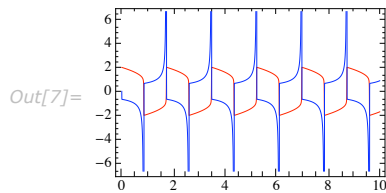
At T == 0.022920104414210326`, system appears to be stiff. Methods Automatic, BDF or StiffnessSwitching may be more appropriate. >>

Solve the system using a method that switches when stiffness occurs.

```
In[6]:= sols = NDSolve[system, {T, 0, 10},
  Method -> {"StiffnessSwitching", "NonstiffTest" -> False}];
```

Here is a plot of the two solution components. The sharp peaks (in blue) extend out to about 450 in magnitude and have been cropped.

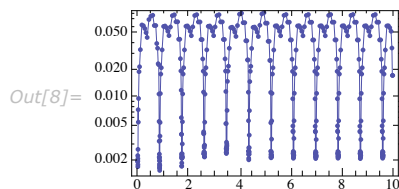
```
In[7]:= Plot[Evaluate[Part[sols, 1, All, 2]], {T, 0, 10},
  PlotStyle -> {{Red}, {Blue}}, Axes -> False, Frame -> True]
```



Stiffness can often occur in regions that follow rapid transients.

This plots the step sizes taken against time.

```
In[8]:= StepDataPlot[sols]
```



The problem is that when the solution is changing rapidly, there is little point using a stiff solver, since local accuracy is the dominant issue.

For efficiency, it would be useful if the method could automatically detect regions where local accuracy (and not stability) is important.

Linear Stability

Linear stability theory arises from the study of Dahlquist's scalar linear test equation:

$$y'(t) = \lambda y(t), \quad \lambda \in \mathbb{C}, \quad \operatorname{Re}(\lambda) < 0 \quad (13)$$

as a simplified model for studying the initial value problem (12).

Stability is characterized by analyzing a method applied to (1) to obtain

$$y_{n+1} = R(z) y_n \quad (14)$$

where $z = h \lambda$ and $R(z)$ is the (rational) stability function.

The boundary of absolute stability is obtained by considering the region:

$$|R(z)| = 1$$

Explicit Euler Method

The explicit or forward Euler method:

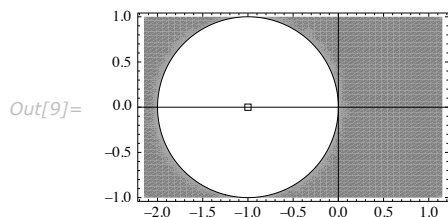
$$y_{n+1} = y_n + h f(t_n, y_n)$$

applied to (1) gives:

$$R(z) = 1 + z.$$

The shaded region represents instability, where $|R(z)| > 1$.

```
In[9]:= OrderStarPlot[1 + z, 1, z, FrameTicks -> True]
```



The Linear Stability Boundary is often taken as the intersection with the negative real axis.

For the explicit Euler method $\text{LSB} = -2$.

For an eigenvalue of $\lambda = -1$, linear stability requirements mean that the step-size needs to satisfy $h < 2$, which is a very mild restriction.

However, for an eigenvalue of $\lambda = -10^6$, linear stability requirements mean that the step size needs to satisfy $h < 2 \times 10^{-6}$, which is a very severe restriction.

Example

This example shows the effect of stiffness on the step-size sequence when using an explicit Runge-Kutta method to solve a stiff system.

This system models a chemical reaction.

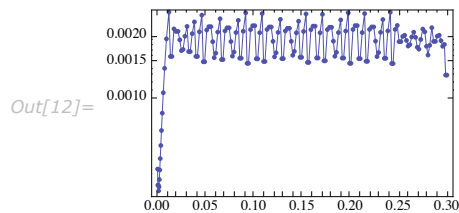
```
In[10]:= system = GetNDSolveProblem["Robertson"];
```

The system is solved by disabling the built-in stiffness detection.

```
In[11]:= sol = NDSolve[system, Method -> {"ExplicitRungeKutta", "StiffnessTest" -> False}];
```

The step-size sequence starts to oscillate when the stability boundary is reached.

```
In[12]:= StepDataPlot[sol]
```



- A large number of step rejections often has a negative impact on performance.
- The large number of steps taken adversely affects the accuracy of the computed solution.

The built-in detection does an excellent job of locating when stiffness occurs.

```
In[13]:= sol = NDSolve[system, Method -> {"ExplicitRungeKutta", "StiffnessTest" -> True}];
```

NDSolve::ndstf:

At T == 0.012555829610695773`, system appears to be stiff. Methods Automatic, BDF or StiffnessSwitching may be more appropriate. >>

Implicit Euler Method

The implicit or backward Euler method:

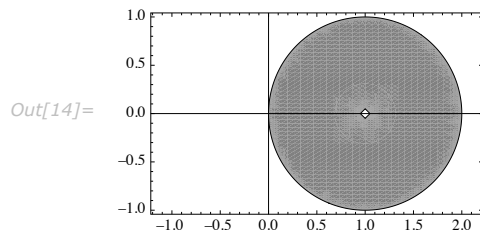
$$y_{n+1} = y_n + h f(t_n, y_{n+1})$$

applied to (1) gives:

$$R(z) = \frac{1}{1 - z}$$

The method is unconditionally stable for the entire left half-plane.

```
In[14]:= OrderStarPlot[1 / (1 - z), 1, z, FrameTicks -> True]
```



This means that to maintain stability there is no longer a restriction on the step size.

The drawback is that an implicit system of equations now has to be solved at each integration step.

Type Insensitivity

A type-insensitive solver recognizes and responds efficiently to stiffness at each step and so is insensitive to the (possibly changing) type of the problem.

One of the most established solvers of this class is LSODA [H83], [P83].

Later generations of LSODA such as CVODE no longer incorporate a stiffness detection device. The reason is because LSODA use norm bounds to estimate the dominant eigenvalue and these bounds, as will be seen later, can be quite inaccurate.

The low order of $A(\alpha)$ -stable BDF methods means that LSODA and CVODE are not very suitable for solving systems with high accuracy or systems where the dominant eigenvalue has a large

imaginary part. Alternative methods, such as those based on extrapolation of linearly implicit schemes, do not suffer from these issues.

Much of the work on stiffness detection was carried out in the 1980s and 1990s using standalone FORTRAN codes.

New linear algebra techniques and efficient software have since become available and these are readily accessible in *Mathematica*.

Stiffness can be a transient phenomenon, so detecting nonstiffness is equally important [S77], [B90].

"StiffnessTest" Method Option

There are several approaches that can be used to switch from a nonstiff to a stiff solver.

Direct Estimation

A convenient way of detecting stiffness is to directly estimate the dominant eigenvalue of the Jacobian J of the problem (see [S77], [P83], [S83], [S84a], [S84c], [R87] and [HW96]).

Such an estimate is often available as a by-product of the numerical integration and so it is reasonably inexpensive.

If v denotes an approximation to the eigenvector corresponding to dominant eigenvalue of the Jacobian, with $\|v\|$ sufficiently small, then by the mean value theorem a good approximation to the leading eigenvalue is:

$$\tilde{\lambda} = \frac{\|f(t, y + v) - f(t, y)\|}{\|v\|}.$$

Richardson's extrapolation provides a sequence of refinements that yields a quantity of this form, as do certain explicit Runge-Kutta methods.

Cost is at most two function evaluations, but often at least one of these is available as a by-product of the numerical integration, so it is reasonably inexpensive.

Let LSB denote the linear stability boundary—the intersection of the linear stability region with the negative real axis.

The product $h\tilde{\lambda}$ gives an estimate that can be compared to the linear stability boundary of a method in order to detect stiffness:

$$\left| h\tilde{\lambda} \right| \leq s |LSB| \quad (15)$$

where s is a safety factor.

Description

The methods `"DoubleStep"`, `"Extrapolation"`, and `"ExplicitRungeKutta"` have the option `"StiffnessTest"`, which can be used to identify whether the method applied with the specified `AccuracyGoal` and `PrecisionGoal` tolerances to a given problem is stiff.

The method option `"StiffnessTest"` itself accepts a number of options that implement a weak form of (15) where the test is allowed to fail a specified number of times.

The reason for this is that some problems can be only mildly stiff in a certain region and an explicit integration method may still be efficient.

"NonstiffTest" Method Option

The `"StiffnessSwitching"` method has the option `"NonstiffTest"`, which is used to switch back from a stiff method to a nonstiff method.

The following settings are allowed for the option `"NonstiffTest"`

- None or False (perform no test).
- "NormBound".
- "Direct".
- "SubspaceIteration".
- "KrylovIteration".
- "Automatic".

Switching to a Nonstiff Solver

An approach that is independent of the stiff method is used.

Given the Jacobian J (or an approximation) compute one of:

$$\text{Norm Bound: } \|J\|$$

$$\text{Spectral Radius: } \rho(J) = \max |\lambda_i|$$

$$\text{Dominant Eigenvalue } \lambda_i : |\lambda_i| > |\lambda_j|$$

Many linear algebra techniques focus on solving a single problem to high accuracy.

For stiffness detection, a succession of problems with solutions to one or two digits are adequate.

For a numerical discretization

$$0 = t_0 < t_1 < \dots < t_n = T$$

consider a sequence k of matrices in some sub-interval(s)

$$J_{t_i}, J_{t_{i+1}}, \dots, J_{t_{i+k-1}}$$

The spectra of the succession of matrices often changes very slowly from step to step.

The goal is to find a way of estimating (bounds on) dominant eigenvalues of a succession of matrices J_{t_i} that:

- Costs less than the work carried out in the linear algebra at each step in the stiff solver.
- Takes account of the step-to-step nature of the solver.

NormBound

A simple and efficient technique of obtaining a bound on the dominant eigenvalue is to use the norm of the Jacobian $\|J\|_p$ where typically $p = 1$ or $p = \infty$.

The method has complexity $O(n^2)$, which is less than the work carried out in the stiff solver.

This is the approach used by LSODA.

- Norm bounds for dense matrices overestimate and the bounds become worse as the dimension increases.
- Norm bounds can be tight for sparse or banded matrices of quite large dimension.

The setting “NormBound” of the option “NonstiffTest” computes $\|J\|_1$ and $\|J\|_\infty$ and returns the smaller of the two values.

Example

The following Jacobian matrix arises in the numerical solution of the van der Pol system using a stiff solver.

```
In[18]:= a = {{0., 1.}, {2623.532160943381, -69.56342161343568}};
```

Bounds based on norms overestimate the spectral radius by more than an order of magnitude.

```
In[19]:= {Abs[First[Eigenvalues[a]]], Norm[a, 1], Norm[a, Infinity]}
```

```
Out[19]= {96.6954, 2623.53, 2693.1}
```

Direct Eigenvalue Computation

For small problems ($n \leq 32$) it can be efficient just to compute the dominant eigenvalue directly.

- Hermitian matrices use the LAPACK function xgeev
- General matrices use the LAPACK function xsyevr

The setting “Direct” of the option “NonstiffTest” computes the dominant eigenvalue of J using the same LAPACK routines as Eigenvalues.

For larger problems the cost of direct eigenvalue computation is $O(n^3)$ which becomes prohibitive when compared to the cost of the linear algebra work in a stiff solver.

A number of iterative schemes have been implemented for this purpose. These effectively work by approximating the dominant eigenspace in a smaller subspace and using dense eigenvalue methods for the smaller problem.

The Power Method

Shampine has proposed the use of the power method for estimating the dominant eigenvalue of the Jacobian [S91].

The power method is perhaps not a very well-respected method, but has received a resurgence of interest due to its use in Google's page ranking.

The power method can be used when

- $A \in \mathbb{C}^{n \times n}$ has n linearly independent eigenvectors (diagonalizable)
- The eigenvalues can be ordered in magnitude as $|\lambda_1| > |\lambda_2| \geq \dots \geq |\lambda_n|$
- λ_1 is the dominant eigenvalue of A .

Description

Given a starting vector $v_0 \in \mathbb{C}^n$ compute

$$v_k = A v_{k-1}$$

The Rayleigh quotient is used to compute an approximation to the dominant eigenvalue:

$$\lambda_1^{(k)} = \frac{v_{k-1}^* A v_{k-1}}{v_{k-1}^* v_{k-1}} = \frac{v_k^* v_{k-1}}{v_{k-1}^* v_{k-1}}$$

In practice, the approximate eigenvector is scaled at each step:

$$\hat{v}_k = \frac{v_k}{\|v_k\|}$$

Properties

The power method converges linearly with rate:

$$\left| \frac{\lambda_1}{\lambda_2} \right|$$

which can be slow.

In particular, the method does not converge when applied to a matrix with a dominant complex conjugate pair of eigenvalues.

Generalizations

The power method can be adapted to overcome the issue of equimodular eigenvalues (e.g. LAPACK)

However the modification does not generally address the issue of the slow rate of convergence for clustered eigenvalues.

There are two main approaches to generalizing the power method:

- Subspace iteration for small to medium dimensions
- Arnoldi iteration for large dimensions

Although the methods work quite differently, there are a number of core components that can be shared and optimized.

Subspace and Krylov iteration cost $O(n^2 m)$ operations.

They project an $n \times n$ matrix to an $m \times m$ matrix, where $m \ll n$.

The small matrix represents the dominant eigenspace and approximation uses dense eigenvalue routines.

Subspace Iteration

Subspace (or simultaneous) iteration generalizes the ideas in the power method by acting on m vectors at each step.

Start with an *orthonormal* set of vectors $V^{(0)} = \mathbb{C}^{n \times m}$, where usually $m \ll n$:

$$V^{(0)} = [v_1, \dots, v_m]$$

Form the product with the matrix A :

$$Z^{(k)} = A V^{(k-1)}$$

In order to prevent all vectors from converging to multiples of the same dominant eigenvector v_1 of A , they are orthonormalized:

$$Q^{(k)} R^{(k)} = Z^{(k)} \quad \text{reduced QR factorization}$$

$$V^{(k)} = Q^{(k)}$$

The orthonormalization step is expensive compared to the matrix product.

Rayleigh-Ritz Projection

Input: matrix A and an orthonormal set of vectors V

- Compute the Rayleigh quotient $S = V^* A V$
- Compute the Schur decomposition $U^* S U = T$

The matrix S has small dimension $m \times m$.

Note that the Schur decomposition can be computed in real arithmetic when $S \in \mathbb{R}^{m \times m}$ using a quasi upper-triangular matrix T .

Convergence

Subspace (or simultaneous) iteration generalizes the ideas in the power method by acting on m vectors at each step.

SRRIT converges linearly with rate:

$$\left| \frac{\lambda_i}{\lambda_{m+1}} \right|, \quad i = 1, \dots, m$$

In particular the rate for the dominant eigenvalue is:

$$\left| \frac{\lambda_1}{\lambda_{m+1}} \right|$$

Therefore it can be beneficial to take e.g. $m = 3$ or more even if we are only interested in the dominant eigenvalue.

Error Control

A relative error test on successive approximations, dominant eigenvalue is:

$$\left| \frac{\lambda_1^{(k)} - \lambda_1^{(k-1)}}{\lambda_1^{(k)}} \right| \leq \text{tol}$$

This is not sufficient since it can be satisfied when convergence is slow.

If $|\lambda_i| = |\lambda_{i-1}|$ or $|\lambda_i| = |\lambda_{i+1}|$ then the i^{th} column of $Q^{(k)}$ is not uniquely determined.

The residual test used in SRRIT is:

$$r^{(k)} = A \hat{q}_i^{(k)} - \hat{Q}^{(k)} t_i^{(k)}, \quad \|r^{(k)}\|_2 \leq \text{tol}$$

where $\hat{Q}^{(k)} = Q^{(k)} U^{(k)}$, $\hat{q}_i^{(k)}$ is the i^{th} column of $\hat{Q}^{(k)}$ and $t_i^{(k)}$ is the i^{th} column of $T^{(k)}$.

This is advantageous since it works for equimodular eigenvalues.

The first column position of the upper triangular matrix $T^{(k)}$ is tested because of the use of an ordered Schur decomposition.

Implementation

There are several implementations of subspace iteration.

- LOPSI [SJ81]
- Subspace iteration with Chebyshev acceleration [S84b], [DS93]
- Schur Rayleigh-Ritz iteration ([BS97] and [SLEPc05])

The implementation for use in "NonstiffTest" is based on:

- Schur Rayleigh-Ritz iteration [BS97]

"An attractive feature of SRRIT is that it displays monotonic consistency, that is, as the convergence tolerance decreases so does the size of the computed residuals" [LS96].

SRRIT makes use of an ordered Schur decomposition where eigenvalues of largest modulus appear in the upper-left entries.

Modified Gram-Schmidt with reorthonormalization is used to form $Q^{(k)}$, which is faster than Householder transformations.

The approximate dominant subspace $V_{t_i}^{(k)}$ at integration time t_i is used to start the iteration at the next integration step t_{i+1} :

$$V_{t_{i+1}}^{(0)} = V_{t_i}^{(k)}$$

KrylovIteration

Given an $n \times m$ matrix V whose columns v_i comprise an orthogonal basis of a given subspace $\mathcal{V}$:

$$V^T V = I \text{ and } \text{span}\{v_1, v_2, \dots, v_m\} = \mathcal{V}$$

The Rayleigh-Ritz procedure consists of computing $H = V^T A V$ and solving the associated eigenproblem $H y_i = \theta_i y_i$.

The approximate eigenpairs of the original problem $\tilde{\lambda}_i, \tilde{x}_i$ satisfy $\tilde{\lambda} = \theta_i$ and $\tilde{x}_i = V y_i$, which are called Ritz values and Ritz vectors.

The process works best when the subspace $\mathcal{V}$ approximates an invariant subspace of A .

This process is effective when $\mathcal{V}$ is equal to the Krylov subspace associated with a matrix A and a given initial vector x as:

$$K_m(A, x) = \text{span}\{x, Ax, A^2x, \dots, A^{m-1}x\}.$$

Description

The method of Arnoldi is a Krylov-based projection algorithm that computes an orthogonal basis of the Krylov subspace and produces a projected $m \times m$ matrix H with $m \ll n$.

Input: matrix A , the number of steps m , an initial vector v_1 of norm 1

Output: (V_m, H_m, f, β) with $\beta = \|f\|_2$

```

For  $j = 1, 2, \dots, m - 1$ 
     $w = A v_j$ 
    Orthogonalize  $w$  with respect to  $V_j$  to obtain  $h_{i,j}$  for  $i = 1, \dots, j$ 
     $h_{j+1,j} = w$  (if  $h_{j+1,j} = 0$  stop)
     $v_{j+1} = w / h_{j+1,j}$ 
end
 $f = A v_m$ 
Orthogonalize  $f$  with respect to  $V_m$  to obtain  $h_{i,m}$  for  $i = 1, \dots, m$ 
 $\beta = \|f\|_2$ 

```

In the case of Arnoldi, H has an unreduced upper Hessenberg form (upper triangular with an additional nonzero subdiagonal).

Orthogonalization is usually carried out by means of a Gram-Schmidt procedure.

The quantities computed by the algorithm satisfy:

$$A V_m = V_m H_m + f e_m^*$$

The residual ε gives an indication of proximity to an invariant subspace and the associated norm β indicates the accuracy of the computed Ritz pairs:

$$\|A \tilde{x}_i - \tilde{\lambda}_i \tilde{x}_i\|_2 = \|A V_m y_i - \theta_i V_m \tilde{x}_i\|_2 = \|(A V_m - V_m \tilde{x}_i) y_i\|_2 = \beta |e_m^* y_i|$$

Restarting

The Ritz pairs converge quickly if the initial vector x is rich in the direction of the desired eigenvalues.

When this is not the case then a restarting strategy is required in order to avoid excessive growth in both work and memory.

There are a several of strategies for restarting, in particular:

- Explicit restart — a new starting vector is a linear combination of a subset of the Ritz vectors.
- Implicit restart — a new starting vector is formed from the Arnoldi process combined with an implicitly shifted QR algorithm.

Explicit restart is relatively simple to implement, but implicit restart is more efficient since it retains the relevant eigeninformation of the larger problem. However implicit restart is difficult to implement in a numerically stable way.

An alternative which is much simpler to implement, but achieves the same effect as implicit restart, is a Krylov-Schur method [S01].

Implementation

A number of software implementations are available, in particular:

- ARPACK [ARPACK98]
- SLEPc [SLEPc05]

The implementation in “NonstiffTest” is based on Krylov-Schur Iteration.

Automatic Strategy

The “Automatic” setting uses an amalgamation of the methods as follows.

- For $n \leq 2*m$ direct eigenvalue computation is used. Either $m = \min(n, m_{\text{si}})$ or $m = \min(n, m_{\text{ki}})$ is used depending on which method is active.
- For $n > 2*m$ subspace iteration is used with a default basis size of $m_{\text{si}} = 8$. If the method succeeds then the resulting basis is used to start the method at the next integration step.
- If subspace iteration fails to converge after max_{si} iterations then the dominant vector is used to start the Krylov method with a default basis size of $m_{\text{ki}} = 16$. Subsequent integration steps use the Krylov method, starting with the resulting vector from the previous step.
- If Krylov iteration fails to converge after max_{ki} iterations then norm bounds are used for the current step. The next integration step will continue to try to use Krylov iteration.
- Since they are so inexpensive, norm bounds are always computed when subspace or Krylov iteration is used and the smaller of the absolute values is used.

Step Rejections

Caching of the time of evaluation ensures that the dominant eigenvalue estimate is not recomputed for rejected steps.

Stiffness detection is also performed for rejected steps since:

- Step rejections often occur for nonstiff solvers when working near the stability boundary
- Step rejections often occur for stiff solvers when resolving fast transients

Iterative Method Options

The iterative methods of “NonstiffTest” have options that can be modified:

```
In[20]:= Options[NDSolve`SubspaceIteration]
```

```
Out[20]= {BasisSize → Automatic, MaxIterations → Automatic, Tolerance →  $\frac{1}{10}$ }
```

```
In[21]:= Options[NDSolve`KrylovIteration]
```

```
Out[21]= {BasisSize → Automatic, MaxIterations → Automatic, Tolerance →  $\frac{1}{10}$ }
```

The default tolerance aims for just one correct digit, but often obtains substantially more accurate values—especially after a few successful iterations at successive steps.

The default values limiting the number of iterations are:

- For subspace iteration $\max_{\text{si}} = \max(25, n/(2 m_{\text{si}}))$.
- For Krylov iteration $\max_{\text{ki}} = \max(50, n/m_{\text{ki}})$.

If these values are set too large then a convergence failure becomes too costly.

In difficult problems, it is better to share the work of convergence across steps. Since the methods effectively refine the basis vectors from the previous step, there is a reasonable chance of convergence in subsequent steps.

Latency and Switching

It is important to incorporate some form of latency in order to avoid a cycle where the “StiffnessSwitching” method continually tries to switch between stiff and nonstiff methods.

The options “MaxRepetitions” and “SafetyFactor” of “StiffnessTest” and “NonstiffTest” are used for this purpose.

The default settings allow switching to be quite reactive, which is appropriate for one-step integration methods.

- “StiffnessTest” is carried out at the end of a step with a nonstiff method. When either value of the option “MaxRepetitions” is reached, a step rejection occurs and the step is recomputed with a stiff method.
- “NonstiffTest” is preemptive. It is performed before a step is taken with a stiff solve using the Jacobian matrix from the previous step.

Examples

Van der Pol

Select an example system.

```
In[22]:= system = GetNDSolveProblem["VanderPol"];
```

StiffnessTest

The system is integrated successfully with the given method and the default option settings for "StiffnessTest".

```
In[23]:= NDSolve[system, Method → "ExplicitRungeKutta"]
```

```
Out[23]= {{Y1[T] → InterpolatingFunction[{{0., 2.5}}, <>][T],  
          Y2[T] → InterpolatingFunction[{{0., 2.5}}, <>][T]}}
```

A longer integration is aborted and a message is issued when the stiffness test condition is not satisfied.

```
In[24]:= NDSolve[system, {T, 0, 10}, Method → "ExplicitRungeKutta"]
```

```
NDSolve::ndstf: At T == 4.353040548903924`, system appears to be stiff.  
Methods Automatic, BDF or StiffnessSwitching may be more appropriate.
```

```
Out[24]= {{Y1[T] → InterpolatingFunction[{{0., 4.35304}}, <>][T],  
          Y2[T] → InterpolatingFunction[{{0., 4.35304}}, <>][T]}}
```

Using a unit safety factor and specifying that only one stiffness failure is allowed effectively gives a strict test. The specification uses the nested method option syntax.

```
In[25]:= NDSolve[system, Method → {"ExplicitRungeKutta",  
    "StiffnessTest" → {True, "MaxRepetitions" → {1, 1}, "SafetyFactor" → 1}}]
```

```
NDSolve::ndstf:  
At T == 0., system appears to be stiff. Methods Automatic, BDF or StiffnessSwitching may  
be more appropriate.
```

```
Out[25]= {{Y1[T] → InterpolatingFunction[{{0., 0.}}, <>][T],  
          Y2[T] → InterpolatingFunction[{{0., 0.}}, <>][T]}}
```

NonstiffTest

For such a small system, direct eigenvalue computation is used.

The example serves as a good test that the overall stiffness switching framework is behaving as expected.

Set up a function to monitor the switch between stiff and nonstiff methods and the step size taken. Data for the stiff and nonstiff solvers is put in separate lists by using a different tag for "Sow".

```
In[26]:= SetAttributes[SowSwitchingData, HoldFirst];
```

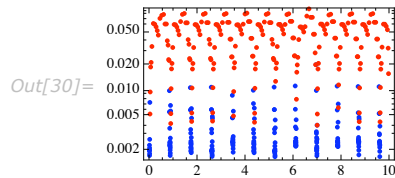
```
SowSwitchingData[told_, t_, method_NDSolve`StiffnessSwitching] :=  
(Sow[{t, t - told}, method["ActiveMethodPosition"]];  
 told = t;);
```

Solve the system and collect the data for the method switching.

```
In[28]:= T0 = 0;
data =
  Last[
    Reap[
      sol = NDSolve[system, {T, 0, 10},
        Method -> StiffnessSwitching,
        "MethodMonitor" -> (SowSwitchingData[T0, T, NDSolve`Self];)
      ];
    ]
  ];
```

Plot the step sizes taken using an explicit solver (blue) and an implicit solver (red).

```
In[30]:= ListLogPlot[data, Axes -> False, Frame -> True, PlotStyle -> {Blue, Red}]
```



Compute the number of nonstiff and stiff steps taken (including rejected steps).

```
In[31]:= Map[Length, data]
```

```
Out[31]= {266, 272}
```

CUSP

The cusp catastrophe model for the nerve impulse mechanism [Z72]:

$$-\varepsilon y'(t) = y(t)^3 + a y(t) + b$$

Combining with the van der Pol oscillator gives rise to the CUSP system [HW96]:

$$\frac{\partial y}{\partial t} = -\frac{1}{\varepsilon} (y^3 + a y + b) + \sigma \frac{\partial^2 y}{\partial x^2}$$

$$\frac{\partial a}{\partial t} = -b + \frac{7}{100} v + \sigma \frac{\partial^2 a}{\partial x^2}$$

$$\frac{\partial b}{\partial t} = (1 - a^2) b - a - \frac{2}{5} y + \frac{7}{200} v + \sigma \frac{\partial^2 b}{\partial x^2}$$

where

$$v = \frac{u}{u - 1/10}, \quad u = (y - 7/10)(y - 13/10)$$

and $\sigma = 1/144$ and $\varepsilon = 10^{-4}$.

Discretization of the diffusion terms using the method of lines is used to obtain a system of ODEs of dimension $3n = 96$.

Unlike the van der Pol system, because of the size of the problem, iterative methods are used for eigenvalue estimation.

Step Size and Order Selection

Select the problem to solve.

```
In[32]:= system = GetNDSolveProblem["Cusp-Discretized"];
```

Set up a function to monitor the type of method used and step size. Additionally the order of the method is included as a Tooltip.

```
In[33]:= SetAttributes[SowOrderData, HoldFirst];

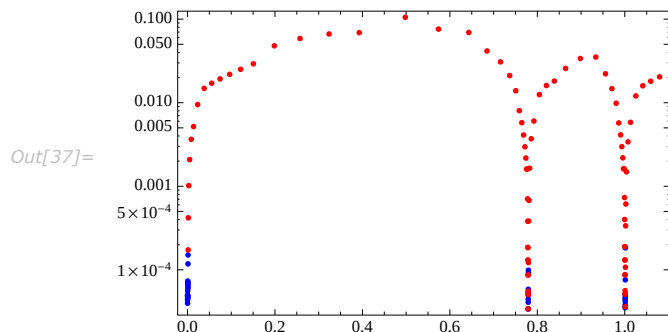
SowOrderData[told_, t_, method_NDSolve`StiffnessSwitching] :=
(Sow[
  Tooltip[{t, t - told}, method["DifferenceOrder"]],
  method["ActiveMethodPosition"]
];
told = t;);
```

Collect the data for the order of the method as the integration proceeds.

```
In[35]:= T0 = 0;
data =
Last[
  Reap[
    sol = NDSolve[system,
      Method -> "StiffnessSwitching",
      "MethodMonitor" -> (SowOrderData[T0, T, NDSolve`Self]);
    ];
  ];
```

Plot the step sizes taken using an explicit solver (blue) and an implicit solver (red). A Tooltip shows the order of the method at each step.

```
In[37]:= ListLogPlot[data, Axes → False, Frame → True, PlotStyle → {Blue, Red}]
```



Compute the total number of nonstiff and stiff steps taken (including rejected steps).

```
In[39]:= Map[Length, data]
```

```
Out[39]= {46, 120}
```

Jacobian Example

Define a function to collect the first few Jacobian matrices.

```
In[41]:= SetAttributes[StiffnessJacobianMonitor, HoldFirst];
```

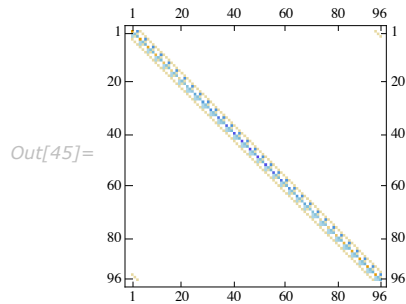
```
StiffnessJacobianMonitor[i_, method_NDSolve`StiffnessSwitching] :=  
  If[SameQ[method["ActiveMethodPosition"], 2] && i < 5,  
    If[MatrixQ[#],  
      Sow[#];  
      i = i + 1  
    ] & @ method["Jacobian"]  
  ];
```

```
In[43]:= i = 0;  
jacdata = Reap[sol = NDSolve[system, Method → "StiffnessSwitching",  
  "MethodMonitor" → (StiffnessJacobianMonitor[i, NDSolve`Self];)];  
][[  
  -1,  
  1]];
```

A switch to a stiff method occurs near 0.00113425 and the first test for nonstiffness occurs at the next step $t_k \approx 0.00127887$.

Graphical illustration of the Jacobian J_{t_k} .

```
In[45]:= MatrixPlot[First[jacdata]]
```



Define a function to compute and display the first few eigenvalues of J_{t_k} , $J_{t_{k+1}}$, ... and the norm bounds.

```
In[46]:= DisplayJacobianData[jdata_] :=
Module[{evdata, hlabels, vlabels},
  evdata =
    Map[
      Join[Eigenvalues[Normal[#], 4], {Norm[#, 1], Norm[#, Infinity]}] &, jdata];
  vlabels = {{""}, {"λ1"}, {"λ2"}, {"λ3"}, {"λ4"}, {"||Jtk||1"}, {"||Jtk||∞"}};
  hlabels = Table[Jtk, {k, Length[jdata]}];
  Grid[
    MapThread[Join, {vlabels, Join[{hlabels}, Transpose[evdata]]}], Frame → All]
];
```

```
In[47]:= DisplayJacobianData[jacdata]
```

Out[47]=

	J_{t_1}	J_{t_2}	J_{t_3}	J_{t_4}	J_{t_5}
λ_1	-56 013.2	-56 009.7	-56 000.	-55 988.2	-55 959.6
λ_2	-56 007.9	-56 003.8	-55 992.2	-55 978.	-55 943.5
λ_3	-55 671.3	-55 670.7	-55 669.1	-55 667.1	-55 662.2
λ_4	-55 660.3	-55 658.3	-55 652.6	-55 645.7	-55 628.9
$\ J_{t_k}\ _1$	56 027.5	56 024.1	56 014.4	56 002.6	55 973.9
$\ J_{t_k}\ _\infty$	81 315.4	81 311.3	81 299.7	81 285.6	81 251.4

Norm bounds are quite sharp in this example.

Korteweg-deVries

The Korteweg-deVries partial differential equation is a mathematical model of waves on shallow water surfaces:

$$\frac{\partial U}{\partial t} + 6U \frac{\partial U}{\partial x} + \frac{\partial^3 U}{\partial x^3} = 0$$

We consider boundary conditions:

$$U(0, x) = e^{-x^2}, U(t, -5) = U(t, 5)$$

and solve over the interval $t \in [0, 1]$.

Discretization using the method of lines is used to form a system of 192 ODEs.

Step Sizes

Select the problem to solve.

```
In[48]:= system = GetNDSolveProblem["Korteweg-deVries-PDE"];
```

The Backward Differentiation Formula methods used in LSODA run into difficulties solving this problem.

```
In[49]:= First[Timing[sollsoda = NDSolve[system, Method -> LSODA];]]
```

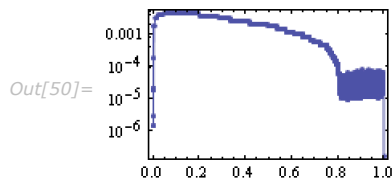
NDSolve::eerr:

Warning: Scaled local spatial error estimate of 806.6079731642326` at T = 1.` in the direction of independent variable X is much greater than prescribed error tolerance. Grid spacing with 193 points may be too large to achieve the desired accuracy or precision. A singularity may have formed or you may want to specify a smaller grid spacing using the MaxStepSize or MinPoints method options. >>

```
Out[49]= 0.971852
```

A plot shows that the step sizes rapidly decrease.

```
In[50]:= StepDataPlot[sollsoda]
```

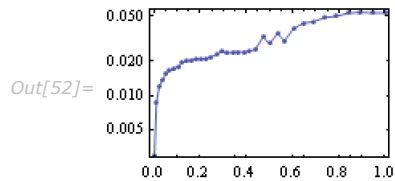


In contrast `StiffnessSwitching` immediately switches to using the linearly implicit Euler method which needs very few integration steps.

```
In[51]:= First[Timing[sol = NDSolve[system, Method -> "StiffnessSwitching"];]]
```

```
Out[51]= 0.165974
```

```
In[52]:= StepDataPlot[sol]
```



The extrapolation methods never switch back to a nonstiff solver once the stiff solver is chosen at the beginning of the integration.

Therefore this is a form of worst case example for the nonstiff detection.

Despite this, the cost of using subspace iteration is only a few percent of the total integration time.

Compute the time taken with switching to a nonstiff method disabled.

```
In[53]:= First[Timing[sol = NDSolve[system,
Method -> {"StiffnessSwitching", "NonstiffTest" -> False}]]];
Out[53]= 0.160974
```

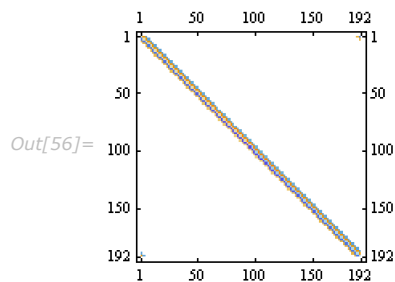
Jacobian Example

Collect data for the first few Jacobian matrices using the previously defined monitor function.

```
In[54]:= i = 0;
jacdata = Reap[sol = NDSolve[system, Method -> "StiffnessSwitching",
"MethodMonitor" -> (StiffnessJacobianMonitor[i, NDSolve`Self];)]];
[[[
-1,
1]]];
```

Graphical illustration of the initial Jacobian J_{t_0} .

```
In[56]:= MatrixPlot[First[jacdata]]
```



Compute and display the first few eigenvalues of $J_{t_k}, J_{t_{k+1}}, \dots$ and the norm bounds.

```
In[57]:= DisplayJacobianData[jacdata]
```

Out[57]=

	J_{t_1}	J_{t_2}	J_{t_3}	J_{t_4}	J_{t_5}
λ_1	$1.37916 \times 10^{-8} + 32\,608. \, i$	$5.3745 \times 10^{-6} + 32\,608. \, i$	$0.0000209094 + 32\,608. \, i$	$0.0000428279 + 32\,608. \, i$	$0.0000678117 + 32\,608.1 \, i$
λ_2	$1.37916 \times 10^{-8} - 32\,608. \, i$	$5.3745 \times 10^{-6} - 32\,608. \, i$	$0.0000209094 - 32\,608. \, i$	$0.0000428279 - 32\,608. \, i$	$0.0000678117 - 32\,608.1 \, i$
λ_3	$5.90398 \times 10^{-8} + 32\,575.5 \, i$	$0.0000103621 + 32\,575.5 \, i$	$0.0000406475 + 32\,575.5 \, i$	$0.0000817789 + 32\,575.5 \, i$	$0.000125286 + 32\,575.6 \, i$
λ_4	$5.90398 \times 10^{-8} - 32\,575.5 \, i$	$0.0000103621 - 32\,575.5 \, i$	$0.0000406475 - 32\,575.5 \, i$	$0.0000817789 - 32\,575.5 \, i$	$0.000125286 - 32\,575.6 \, i$
$\ J_{t_k}\ _1$	38 928.4	38 928.4	38 928.4	38 930.	38 932.9
$\ J_{t_k}\ _\infty$	38 928.4	38 928.4	38 928.4	38 930.1	38 933.

Norm bounds overestimate slightly, but more importantly they give no indication of the relative size of real and imaginary parts.

Option Summary

StiffnessTest

option name	default value	
"MaxRepetitions"	{3, 5}	specify the maximum number of successive and total times that the stiffness test (15) is allowed to fail
"SafetyFactor"	$\frac{4}{5}$	specify the safety factor to use in the right-hand side of the stiffness test (15)

Options of the method option "StiffnessTest".

NonstiffTest

option name	default value	
"MaxRepetitions"	{2,∞}	specify the maximum number of successive and total times that the stiffness test (15) is allowed to fail
"SafetyFactor"	$\frac{4}{5}$	specify the safety factor to use in the right-hand side of the stiffness test (15)

Options of the method option "NonstiffTest".

Structured Systems

Numerical Methods for Solving the Lotka-Volterra Equations

Introduction

The Lotka-Volterra system arises in mathematical biology and models the growth of animal species. Consider two species where $Y_1(T)$ denotes the number of predators and $Y_2(T)$ denotes the number of prey. A particular case of the Lotka-Volterra differential system is:

$$\dot{Y}_1 = Y_1(Y_2 - 1), \quad \dot{Y}_2 = Y_2(2 - Y_1),$$

(1)

where the dot denotes differentiation with respect to time T .

The Lotka-Volterra system (9) has an invariant H , which is constant for all T :

$$H(Y_1, Y_2) = 2 \ln Y_1 - Y_1 + \ln Y_2 - Y_2.$$

(2)

The level curves of the invariant (2) are closed so that the solution is periodic. It is desirable that the numerical solution of (9) is also periodic, but this is not always the case. Note that (9) is a Poisson system:

$$\dot{Y} = B(Y) \nabla H(Y) = \begin{pmatrix} 0 & -Y_1 Y_2 \\ Y_1 Y_2 & 0 \end{pmatrix} \begin{pmatrix} \frac{2}{Y_1} - 1 \\ \frac{1}{Y_2} - 1 \end{pmatrix} \quad (3)$$

where $H(Y)$ is defined in (2).

Poisson systems and Poisson integrators are discussed in Chapter VII.2 of [HLW02] and [MQ02].

Load a package with some predefined problems and select the Lotka-Volterra system.

```
In[10]:= Needs["DifferentialEquations`NDSolveProblems`"];
Needs["DifferentialEquations`NDSolveUtilities`"];
Needs["DifferentialEquations`InterpolatingFunctionAnatomy`"];

system = GetNDSolveProblem["LotkaVolterra"];
invts = system["Invariants"];
time = system["TimeData"];
vars = system["DependentVariables"];
step = 3 / 25;
```

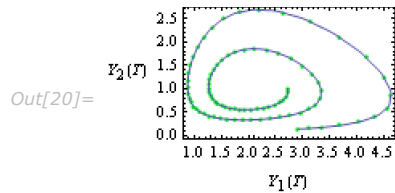
Define a utility function for visualizing solutions.

```
In[18]:= LotkaVolterraPlot[sol_, vars_, time_, opts___?OptionQ] :=
Module[{data, data1, data2, ifuns, lplot, pplot},
  ifuns = First[vars /. sol];
  data1 = Part[ifuns, 1, 0] ["ValuesOnGrid"];
  data2 = Part[ifuns, 2, 0] ["ValuesOnGrid"];
  data = Transpose[{data1, data2}];
  commonopts = Sequence[Axes → False, Frame → True, FrameLabel →
    Join[Map[TraditionalForm, vars], {None, None}], RotateLabel → False];
  lplot = ListPlot[data, Evaluate[FilterRules[{opts}, Options[ListPlot]]],
    PlotStyle → {PointSize[0.02], RGBColor[0, 1, 0]}, Evaluate[commonopts]];
  pplot = ParametricPlot[Evaluate[ifuns], time, Evaluate[
    FilterRules[{opts}, Options[ParametricPlot]]], Evaluate[commonopts]];
  Show[lplot, pplot]
];
```


Explicit Euler

Use the explicit or forward Euler method to solve the system (9).

```
In[19]:= fesol = NDSolve[system, Method → "ExplicitEuler", StartingStepSize → step];
LotkaVolterraPlot[fesol, vars, time]
```



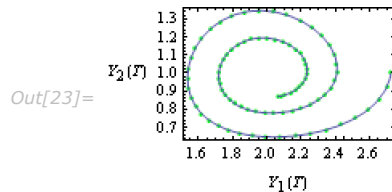
Backward Euler

Define the backward or implicit Euler method in terms of the RadauIIA implicit Runge-Kutta method and use it to solve (9). The resulting trajectory spirals from the initial conditions toward a fixed point at (2, 1) in a clockwise direction.

```
In[21]:= BackwardEuler = {"FixedStep", Method → {"ImplicitRungeKutta", "Coefficients" →
  "ImplicitRungeKuttaRadauIIACoefficients", "DifferenceOrder" → 1,
  "ImplicitSolver" → {"FixedPoint", AccuracyGoal → MachinePrecision,
    PrecisionGoal → MachinePrecision, "IterationSafetyFactor" → 1}}};

besol = NDSolve[system, Method → BackwardEuler, StartingStepSize → step];

LotkaVolterraPlot[besol, vars, time]
```

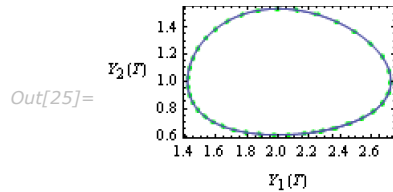


Projection

Projection of the forward Euler method using the invariant (2) of the Lotka-Volterra equations gives a periodic solution.

```
In[24]:= pfesol = NDSolve[system,
  Method → {Projection, Method → "ExplicitEuler", Invariants → invts},
  StartingStepSize → step];
```

```
LotkaVolterraPlot[pfesol, vars, time]
```



Splitting

Another approach for obtaining the correct qualitative behavior is to additively split (9) into two systems:

$$\begin{aligned} \dot{Y}_1 &= Y_1(Y_2 - 1) & \dot{Y}_2 &= 0 \\ \dot{Y}_1 &= 0 & \dot{Y}_2 &= Y_2(2 - Y_1). \end{aligned} \quad (4)$$

By appropriately solving (4) it is possible to construct Poisson integrators.

Define the equations for splitting of the Lotka-Volterra equations.

```
In[26]:= eqs = system["System"];
Y1 = eqs;
Part[Y1, 2, 2] = 0;
Y2 = eqs;
Part[Y2, 1, 2] = 0;
```

Symplectic Euler

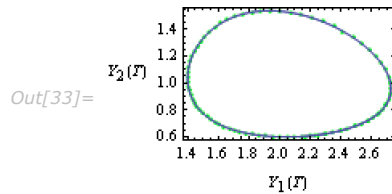
Define the symplectic Euler method in terms of a splitting method using the backward and forward Euler methods for each system in (4).

```
In[31]:= SymplecticEuler = {"Splitting",
  "DifferenceOrder" → 1, "Equations" → {Y1, Y2},
  "Method" → {BackwardEuler, "ExplicitEuler"}};

sesol = NDSolve[system, Method → SymplecticEuler, StartingStepSize → step];
```

The numerical solution using the symplectic Euler method is periodic.

```
In[33]:= LotkaVolterraPlot[sesol, vars, time]
```



Flows

Consider splitting the Lotka-Volterra equations and computing the flow (or exact solution) of each system in (4). The solutions can be found as follows, where the constants should be related to the initial conditions at each step.

```
In[34]:= DSolve[Y1, vars, T]
```

```
Out[34]= {{Y2[T] → C[1], Y1[T] → eT (-1+C[1]) C[2]}}
```

```
In[35]:= DSolve[Y2, vars, T]
```

```
Out[35]= {{Y1[T] → C[1], Y2[T] → eT (2-C[1]) C[2]}}
```

An advantage of locally computing the flow is that it yields an *explicit*, and hence very efficient, integration procedure. The "LocallyExact" method provides a general way of computing the flow of each splitting using DSolve only during the initialization phase.

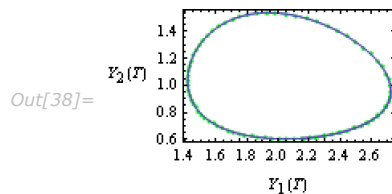
Set up a hybrid symbolic-numeric splitting method and use it to solve the Lotka-Volterra system.

```
In[36]:= SplittingLotkaVolterra = {"Splitting",
  "DifferenceOrder" → 1, "Equations" → {Y1, Y2},
  "Method" → {"LocallyExact", "LocallyExact"}};
```

```
spsol = NDSolve[system, Method → SplittingLotkaVolterra, StartingStepSize → step];
```

The numerical solution using the splitting method is periodic.

```
In[38]:= LotkaVolterraPlot[spsol, vars, time]
```



Rigid Body Solvers

Introduction

The equations of motion for a free rigid body whose center of mass is at the origin are given by the following Euler equations (see [MR99]).

$$\begin{pmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \end{pmatrix} = \begin{pmatrix} 0 & y_3/I_3 & -y_2/I_2 \\ -y_3/I_3 & 0 & y_1/I_1 \\ y_2/I_2 & -y_1/I_1 & 0 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Two quadratic first integrals of the system are:

$$\begin{aligned} I(y) &= y_1^2 + y_2^2 + y_3^2 \\ H(y) &= \frac{1}{2} \left(\frac{y_1^2}{I_1} + \frac{y_2^2}{I_2} + \frac{y_3^2}{I_3} \right). \end{aligned}$$

The first constraint effectively confines the motion from $\mathbb{R}^3$ to a sphere. The second constraint represents the kinetic energy of the system and, in conjunction with the first invariant, effectively confines the motion to ellipsoids on the sphere.

Numerical experiments for various methods are given in [HLW02] and a variety of `NDSolve` methods will now be compared.

Manifold Generation and Utility Functions

Load some useful packages.

```
In[6]:= Needs["DifferentialEquations`NDSolveProblems`"];
Needs["DifferentialEquations`NDSolveUtilities`"];
```

Define Euler's equations for rigid body motion together with the invariants of the system.

```
In[8]:= system = GetNDSolveProblem["RigidBody"];
eqs = system["System"];
vars = system["DependentVariables"];
time = system["TimeData"];
invariants = system["Invariants"];
```

The equations of motion evolve as closed curves on the unit sphere. This generates a three-dimensional graphics object to represent the unit sphere.

```
In[13]:= UnitSphere = Graphics3D[{EdgeForm[], Sphere[]}, Boxed -> False];
```

This function superimposes a solution from NDSolve on a given manifold.

```
In[14]:= PlotSolutionOnManifold[sol_, vars_, time_, manifold_, opts___?OptionQ] :=
Module[{solplot},
  solplot = ParametricPlot3D[
    Evaluate[vars /. sol], time, opts, Boxed → False, Axes → False];
  Show[solplot, manifold, opts]
]
```

This function plots the various solution components.

```
In[15]:= PlotSolutionComponents[sols_, vars_, time_, opts___?OptionQ] :=
Module[{ifuns, plotopts},
  ifuns = vars /. First[sols];
  Table[plotopts = Sequence[PlotLabel →
    StringForm["`1` vs time", Part[vars, i]], Frame → True, Axes → False];
    Plot[Evaluate[Part[ifuns, i]], time, opts, Evaluate[plotopts]],
    {i, Length[vars]}]
];
```

Method Comparison

Various integration methods can be used to solve Euler's equations and they each have different associated costs and different dynamical properties.

Adams Multistep Method

Here an Adams method is used to solve the equations of motion.

```
In[21]:= AdamsSolution = NDSolve[system, Method → "Adams"];
```

This shows the solution trajectory by superimposing it on the unit sphere.

```
In[22]:= PlotSolutionOnManifold[AdamsSolution, vars, time, UnitSphere, PlotRange -> All]
```

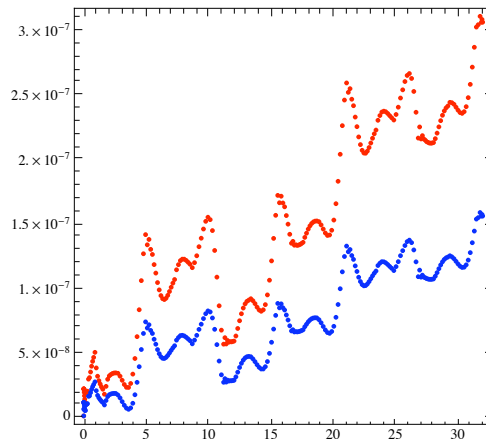
Out[22]=



The solution appears visually to give a closed curve on the sphere. However, a plot of the error reveals that neither constraint is conserved particularly well.

```
In[23]:= InvariantErrorPlot[invariants, vars, T, AdamsSolution, PlotStyle -> {Red, Blue}]
```

Out[23]=



Euler and Implicit Midpoint Methods

This solves the equations of motion using Euler's method with a specified fixed step size.

```
In[16]:= EulerSolution = NDSolve[system,
    Method -> {"FixedStep", Method -> "ExplicitEuler"}, StartingStepSize -> 1 / 20];
```

This solves the equations of motion using the implicit midpoint method with a specified fixed step size.

```
In[17]:= ImplicitMidpoint = {"FixedStep", Method → {"ImplicitRungeKutta",
  "Coefficients" → "ImplicitRungeKuttaGaussCoefficients", DifferenceOrder → 2,
  "ImplicitSolver" → {FixedPoint, "AccuracyGoal" → MachinePrecision,
    "PrecisionGoal" → MachinePrecision, "IterationSafetyFactor" → 1}}};

IMPSolution =
  NDSolve[system, Method → ImplicitMidpoint, StartingStepSize → 3 / 10];
```

This shows the superimposition on the unit sphere of the numerical solution of the equations of motion for Euler's method (left) and the implicit midpoint rule (right).

```
In[19]:= EulerPlotOnSphere =
  PlotSolutionOnManifold[EulerSolution, vars, time, UnitSphere, PlotRange → All];

IMPPlotOnSphere =
  PlotSolutionOnManifold[IMPSolution, vars, time, UnitSphere, PlotRange → All];

GraphicsArray[{EulerPlotOnSphere, IMPPlotOnSphere}]
```

Out[21]=



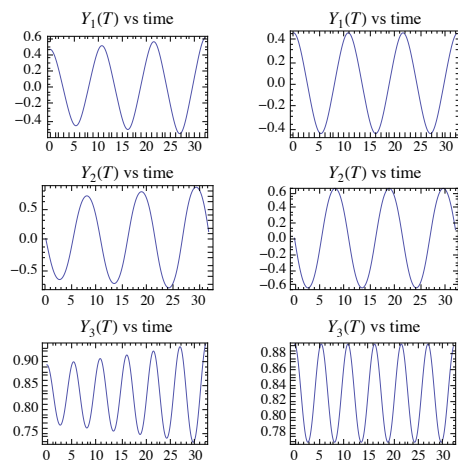
This shows the components of the numerical solution using Euler's method (left) and the implicit midpoint rule (right).

```
In[30]:= EulerSolutionPlots = PlotSolutionComponents[EulerSolution, vars, time];

IMPSolutionPlots = PlotSolutionComponents[IMPSolution, vars, time];

GraphicsArray[Transpose[{EulerSolutionPlots, IMPSolutionPlots}]]
```

Out[32]=



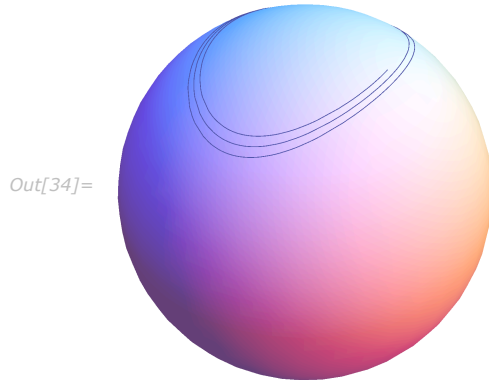
Orthogonal Projection Method

Here the “OrthogonalProjection” method is used to solve the equations.

```
In[33]:= OPSolution = NDSolve[system, Method → {"OrthogonalProjection",  
Dimensions → {3, 1}, Method → "ExplicitEuler"}, StartingStepSize → 1 / 20];
```

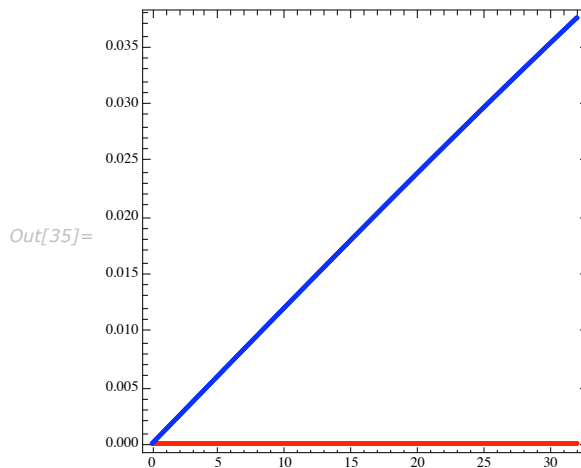
Only the orthogonal constraint is conserved so the curve is not closed.

```
In[34]:= PlotSolutionOnManifold[OPSolution, vars, time, UnitSphere, PlotRange → All]
```



Plotting the error in the invariants against time, it can be seen that the orthogonal projection method conserves only one of the two invariants.

```
In[35]:= InvariantErrorPlot[invariants, vars, T, OPSolution, PlotStyle → {Red, Blue}]
```



Projection Method

The method “Projection” takes a set of constraints and projects the solution onto a manifold at the end of each integration step.

Generally all the invariants of the problem should be used in the projection; otherwise the numerical solution may actually be qualitatively worse than the unprojected solution.

The following specifies the integration method and defers determination of the constraints until the invocation of `NDSolve`.

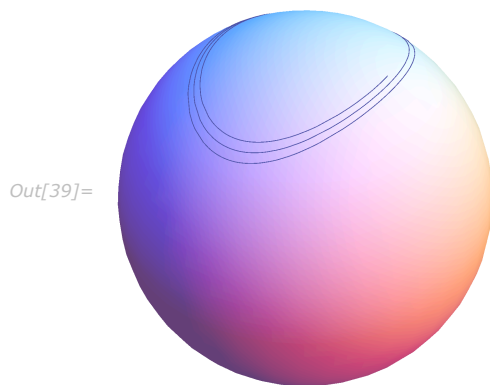
```
In[36]:= ProjectionMethod = {Projection,
    Method → {"FixedStep", Method → "ExplicitEuler"}, "Invariants" → invts};
```

Projecting One Constraint

This projects the first constraint onto the manifold.

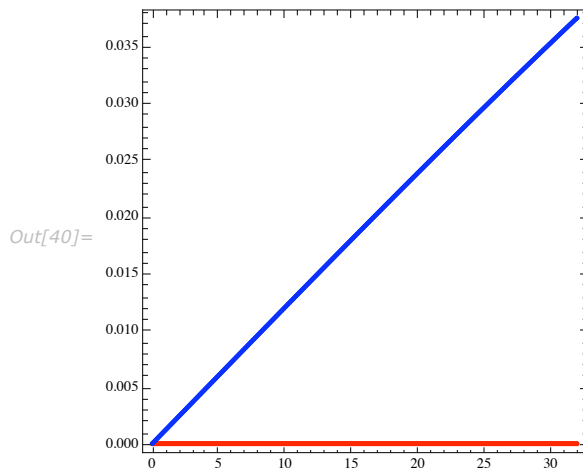
```
In[37]:= invts = First[invariants];

projsol1 = NDSolve[system, Method → ProjectionMethod, StartingStepSize → 1 / 20];
PlotSolutionOnManifold[projsol1, vars, time, UnitSphere, PlotRange → All]
```



Only the first invariant is conserved.

```
In[40]:= InvariantErrorPlot[invariants, vars, T, projsol1, PlotStyle → {Red, Blue}]
```



This projects the second constraint onto the manifold.

```
In[41]:= invts = Last[invariants];  
projsol2 = NDSolve[system, Method -> ProjectionMethod, StartingStepSize -> 1 / 20];  
PlotSolutionOnManifold[projsol2, vars, time, UnitSphere, PlotRange -> All]
```

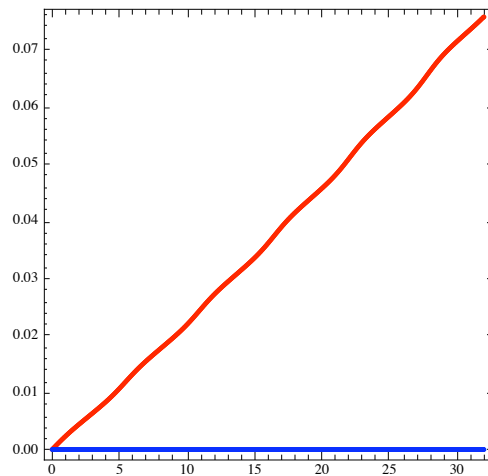
Out[43]=



Only the second invariant is conserved.

```
In[44]:= InvariantErrorPlot[invariants, vars, T, projsol2, PlotStyle -> {Red, Blue}]
```

Out[44]=



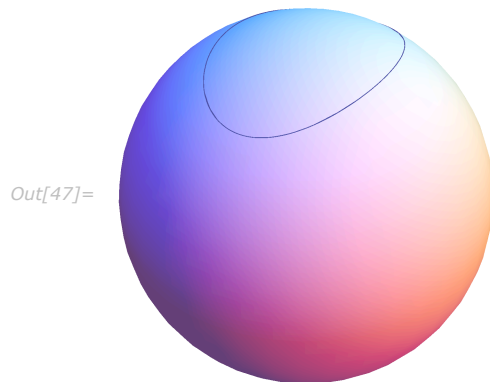
Projecting Multiple Constraints

This projects both constraints onto the manifold.

```
In[45]:= invts = invariants;

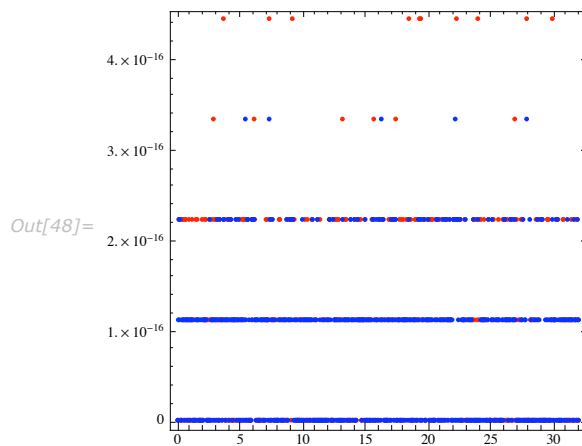
projsol = NDSolve[system, Method -> ProjectionMethod, StartingStepSize -> 1 / 20];

PlotSolutionOnManifold[projsol, vars, time, UnitSphere, PlotRange -> All]
```



Now both invariants are conserved.

```
In[48]:= InvariantErrorPlot[invariants, vars, T, projsol, PlotStyle -> {Red, Blue}]
```



"Splitting" Method

A splitting that yields an efficient explicit integration method was derived independently by McLachlan [M93] and Reich [R93].

Write the flow of an ODE $\dot{y} = Y$ as $y(t) = \exp(t Y)(y(0))$.

The differential system is split into three components, $YH1$, $YH2$, and $YH3$, each of which is Hamiltonian and can be solved exactly.

The Hamiltonian systems are solved and recombined at each integration step as:

$$\exp(tY) \approx \exp(1/2 t YH1) \exp(1/2 t YH2) \exp(t YH3) \exp(1/2 t YH2) \exp(1/2 t YH1).$$

This defines an appropriate splitting into Hamiltonian vector fields.

```
In[49]:= Grad[H_, x_?VectorQ] := Map[D[H, #] &, x];
isub = {I1 -> 2, I2 -> 1, I3 -> 2/3};
H1 = Y1[T]^2 / (2 I1) /. isub;
H2 = Y2[T]^2 / (2 I2) /. isub;
H3 = Y3[T]^2 / (2 I3) /. isub;
JX = {{0, -Y3[T], Y2[T]}, {Y3[T], 0, -Y1[T]}, {-Y2[T], Y1[T], 0}};
YH1 = Thread[D[vars, T] == JX.Grad[H1, vars]];
YH2 = Thread[D[vars, T] == JX.Grad[H2, vars]];
YH3 = Thread[D[vars, T] == JX.Grad[H3, vars]];
```

Here is the differential system for Euler's equations.

```
In[58]:= eqs
```

$$\text{Out[58]} = \left\{ Y_1'[T] = -\frac{1}{2} Y_2[T] Y_3[T], Y_2'[T] = -Y_1[T] Y_3[T], Y_3'[T] = -\frac{1}{2} Y_1[T] Y_2[T] \right\}$$

Here are the three split vector fields.

```
In[59]:= YH1
```

$$\text{Out[59]} = \left\{ Y_1'[T] = 0, Y_2'[T] = \frac{1}{2} Y_1[T] Y_3[T], Y_3'[T] = -\frac{1}{2} Y_1[T] Y_2[T] \right\}$$

```
In[60]:= YH2
```

$$\text{Out[60]} = \{ Y_1'[T] = -Y_2[T] Y_3[T], Y_2'[T] = 0, Y_3'[T] = Y_1[T] Y_2[T] \}$$

```
In[61]:= YH3
```

$$\text{Out[61]} = \left\{ Y_1'[T] = \frac{3}{2} Y_2[T] Y_3[T], Y_2'[T] = -\frac{3}{2} Y_1[T] Y_3[T], Y_3'[T] = 0 \right\}$$

Solution

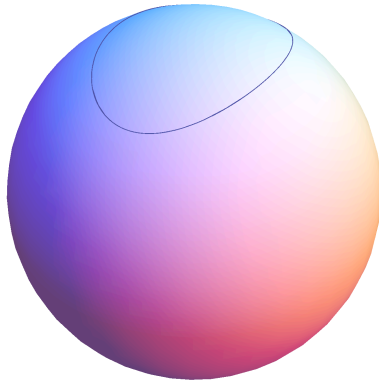
This defines a symmetric second-order splitting method. The coefficients are automatically determined from the structure of the equations and are an extension of the Strang splitting.

```
In[62]:= SplittingMethod =
{ "Splitting",
  "DifferenceOrder" -> 2,
  "Equations" -> {YH1, YH2, YH3, YH2, YH1},
  "Method" -> {"LocallyExact"};}
```

This solves the system and graphically displays the solution.

```
In[63]:= splitsol = NDSolve[system, Method -> SplittingMethod, StartingStepSize -> 1 / 20];
PlotSolutionOnManifold[splitsol, vars, time, UnitSphere, PlotRange -> All]
```

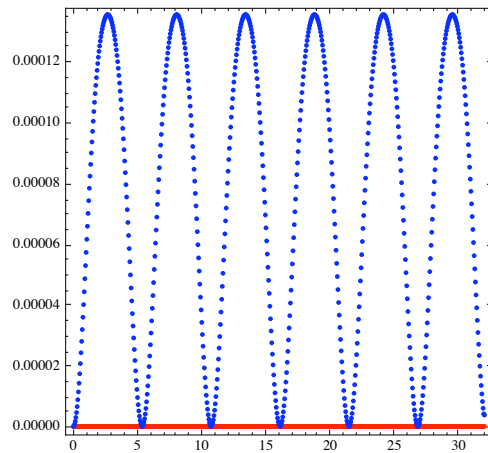
Out[64]=



One of the invariants is preserved up to roundoff while the error in the second invariant remains bounded.

```
In[65]:= InvariantErrorPlot[invariants, vars, T, splitsol, PlotStyle -> {Red, Blue}]
```

Out[65]=



Components and Data Structures in NDSolve

Introduction

`NDSolve` is broken up into several basic steps. For advanced usage, it can sometimes be advantageous to access components to carry out each of these steps separately.

- Equation processing and method selection
- Method initialization
- Numerical solution
- Solution processing

`NDSolve` performs each of these steps internally, hiding the details from a casual user. However, for advanced usage it can sometimes be advantageous to access components to carry out each of these steps separately.

Here are the low-level functions that are used to break up these steps.

- `NDSolve`ProcessEquations`
- `NDSolve`Iterate`
- `NDSolve`ProcessSolutions`

`NDSolve`ProcessEquations` classifies the differential system into initial value problem, boundary value problem, differential-algebraic problem, partial differential problem, etc. It also chooses appropriate default integration methods and constructs the main `NDSolve`StateData` data structure.

`NDSolve`Iterate` advances the numerical solution. The first invocation (there can be several) initializes the numerical integration methods.

`NDSolve`ProcessSolutions` converts numerical data into an `InterpolatingFunction` to represent each solution.

Note that `NDSolve`ProcessEquations` can take a significant portion of the overall time to solve a differential system. In such cases, it can be useful to perform this step only once and use `NDSolve`Reinitialize` to repeatedly solve for different options or initial conditions.

Example

Process equations and set up data structures for solving the differential system.

```
In[1]:= ndssdata =
  First[NDSolve`ProcessEquations[{y'[t] + y[t] == 0, y[0] == 1, y'[0] == 0},
    {y, y'}, t, Method -> "ExplicitRungeKutta"]]
Out[1]= NDSolve`StateData[<0.>]
```

Initialize the method "ExplicitRungeKutta" and integrate the system up to time 10. The return value of `NDSolve`Iterate` is `Null` in order to avoid extra references, which would lead to undesirable copying.

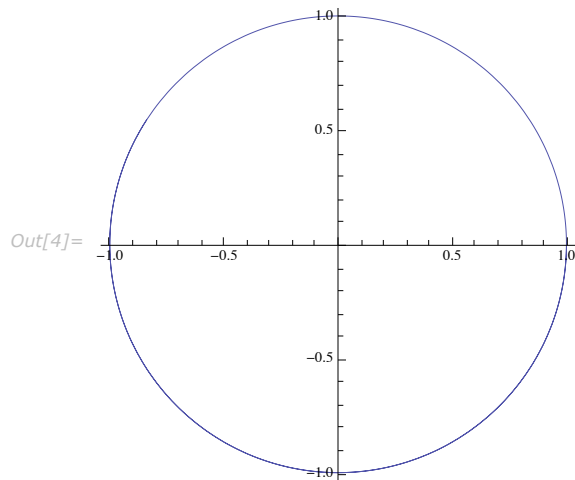
```
In[2]:= NDSolve`Iterate[ndssdata, 10]
```

Convert each set of solution data into an `InterpolatingFunction`.

```
In[3]:= ndsol = NDSolve`ProcessSolutions[ndssdata]
Out[3]= {y -> InterpolatingFunction[{{0., 10.}}, <>], y' -> InterpolatingFunction[{{0., 10.}}, <>]}
```

Representing the solution as an `InterpolatingFunction` allows continuous output even for points that are not part of the numerical solution grid.

```
In[4]:= ParametricPlot[{y[t], y'[t]} /. ndsol, {t, 0, 10}]
```



Creating NDSolve`StateData Objects

ProcessEquations

The first stage of any solution using `NDSolve` is processing the equations specified into a form that can be efficiently accessed by the actual integration algorithms. This stage minimally involves determining the differential order of each variable, making substitutions needed to get a first-order system, solving for the time derivatives of the functions in terms of the functions, and forming the result into a “`NumericalFunction`” object. If you want to save the time of repeating this process for the same set of equations or if you want more control over the numerical integration process, the processing stage can be executed separately with `NDSolve`ProcessEquations`.

```
NDSolve`ProcessEquations[{eqn1, eqn2, ...}, {u1, u2, ...}, t]
```

process the differential equations $\{eqn_1, eqn_2, \dots\}$ for the functions $\{u_1, u_2, \dots\}$ into a normal form; return a list of `NDSolve`StateData` objects containing the solution and data associated with each solution for the time derivatives of the functions in terms of the functions;  $t$  may be specified in a list with a range of values as in `NDSolve`

```
NDSolve`ProcessEquations[{eqn1, eqn2, ...}, {u1, u2, ...}, {x1, x1min, x1max}, {x2, x2min, x2max}, ...]
```

process the partial differential equations $\{eqn_1, eqn_2, \dots\}$ for the functions $\{u_1, u_2, \dots\}$ into a normal form; return a list of `NDSolve`StateData` objects containing the solution and data associated with each solution for the time derivatives of the functions in terms of the functions; if x_j is the temporal variable, it need not be specified with the boundaries x_{jmin}, x_{jmax}

Processing equations for `NDSolve`.

This creates a list of two `NDSolve`StateData` objects because there are two possible solutions for the y' in terms of y .

```
In[1]:= NDSolve`ProcessEquations[{y'[x]^2 == y[x] + x, y[0] == 1}, y, x]
```

```
Out[1]= {NDSolve`StateData[<0.>], NDSolve`StateData[<0.>]}
```


Reinitialize

It is not uncommon that the solution to a more sophisticated problem involves solving the same differential equation repeatedly, but with different initial conditions. In some cases, processing equations may be as time-consuming as numerically integrating the differential equations. In these situations, it is a significant advantage to be able to simply give new initial values.

`NDSolve`Reinitialize[
state, conditions]`

assuming the equations and variables are the same as the ones used to create the `NDSolve`StateData` object *state*, form a list of new `NDSolve`StateData` objects, one for each of the possible solutions for the initial values of the functions of the equations *conditions*

Reusing processed equations.

This creates an `NDSolve`StateData` object for the harmonic oscillator.

```
In[2]:= state =  
First[NDSolve`ProcessEquations[{x'[t] + x[t] == 0, x[0] == 0, x'[0] == 1}, x, t]]  
Out[2]= NDSolve`StateData[<0.>]
```

This creates three new `NDSolve`StateData` objects, each with a different initial condition.

```
In[3]:= newstate = NDSolve`Reinitialize[state, {x[1]^3 == 1, x'[1] == 0}]  
Out[3]= {NDSolve`StateData[<1.>], NDSolve`StateData[<1.>], NDSolve`StateData[<1.>]}
```

Using `NDSolve`Reinitialize` may save computation time when you need to solve the same differential equation for many different initial conditions, as you might in a shooting method for boundary value problems.

A subset of `NDSolve` options can be specified as options to `NDSolve`Reinitialize`.

This creates a new `NDSolve`StateData` object, specifying a starting step size.

```
In[3]:= newstate =  
NDSolve`Reinitialize[state, {x[0] == 0, x'[0] == 1}, StartingStepSize -> 1/10]  
Out[3]= {NDSolve`StateData[<0.>]}
```

Iterating Solutions

One important use of `NDSolve`StateData` objects is to have more control of the integration. For some problems, it is appropriate to check the solution and start over or change parameters, depending on certain conditions.

<code>NDSolve`Iterate[state, t]</code>	compute the solution of the differential equation in an <code>NDSolve`StateData</code> object that has been assigned as the value of the variable <i>state</i> from the current time up to time <i>t</i>
--	--

Iterating solutions to differential equations.

This creates an `NDSolve`StateData` object that contains the information needed to solve the equation for an oscillator with a varying coefficient using an explicit Runge-Kutta method.

```
In[4]:= state =
  First[NDSolve`ProcessEquations[{x'[t] + (1 + 4 UnitStep[Sin[t]]) x[t] == 0,
    x[0] == 1, x'[0] == 0}, x, t, Method -> "ExplicitRungeKutta"]]
Out[4]= NDSolve`StateData[<0.>]
```

Note that when you use `NDSolve`ProcessEquations`, you do not need to give the range of the *t* variable explicitly because that information is not needed to set up the equations in a form ready to solve. (For PDEs, you do have to give the ranges of all spatial variables, however, since that information is essential for determining an appropriate discretization.)

This computes the solution out to time $t = 1$.

```
In[5]:= NDSolve`Iterate[state, 1]
```

`NDSolve`Iterate` does not return a value because it modifies the `NDSolve`StateData` object assigned to the variable *state*. Thus, the command affects the value of the variable in a manner similar to setting parts of a list, as described in "Manipulating Lists by Their Indices". You can see that the value of *state* has changed since it now displays the current time to which it is integrated.

The output form of *state* shows the range of times over which the solution has been integrated.

```
In[6]:= state
Out[6]= NDSolve`StateData[<0.,1.>]
```

If you want to integrate further, you can call `NDSolve`Iterate` again, but with a larger value for time.

This computes the solution out to time $t = 3$.

```
In[7]:= NDSolve`Iterate[state, 3]
```

You can specify a time that is earlier than the first current time, in which case the integration proceeds backwards with respect to time.

This computes the solution from the initial condition backwards to $t = -\pi/2$.

```
In[8]:= NDSolve`Iterate[state, -Pi / 2]
```

`NDSolve`Iterate` allows you to specify intermediate times at which to stop. This can be useful, for example, to avoid discontinuities. Typically, this strategy is more effective with so-called one-step methods, such as the explicit Runge-Kutta method used in this example. However, it generally works with the default `NDSolve` method as well.

This computes the solution out to $t = 10\pi$, making sure that the solution does not have problems with the points of discontinuity in the coefficients at $t = \pi, 2\pi, \dots$

```
In[9]:= NDSolve`Iterate[state, Pi Range[10]]
```

Getting Solution Functions

Once you have integrated a system up to a certain time, typically you want to be able to look at the current solution values and to generate an approximate function representing the solution computed so far. The command `NDSolve`ProcessSolutions` allows you to do both.

`NDSolve`ProcessSolutions[state]` give the solutions that have been computed in *state* as a list of rules with `InterpolatingFunction` objects

Getting solutions as `InterpolatingFunction` objects.

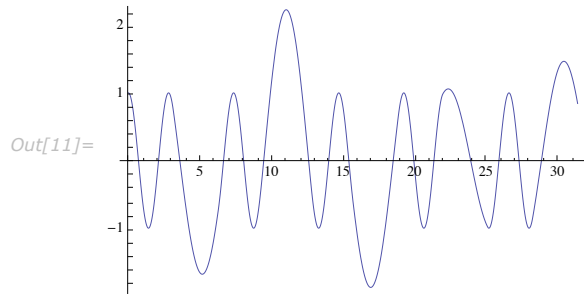
This extracts the solution computed in the previous section as an `InterpolatingFunction` object.

```
In[10]:= sol = NDSolve`ProcessSolutions[state]
```

```
Out[10]= {x -> InterpolatingFunction[{{-1.5708, 31.4159}}, <>]}
```

This plots the solution.

```
In[11]:= Plot[Evaluate[x[t] /. sol], {t, 0, 10 Pi}]
```



Just as when using `NDSolve` directly, there will be a rule for each function you specified in the second argument to `NDSolve`ProcessEquations`. Only the specified components of the solutions are saved in such a way that an `InterpolatingFunction` object can be created.

`NDSolve`ProcessSolutions[
state, dir]`

give the solutions that have been most recently computed in direction *dir* in *state* as a list of rules with values for both the functions and their derivatives

Obtaining the current solution values.

This gives the current solution values and derivatives in the forward direction.

```
In[12]:= sol = NDSolve`ProcessSolutions[state, "Forward"]
```

```
Out[12]= {x[31.4159] → 0.843755, x'[31.4159] → -1.20016, x''[31.4159] → -0.843755}
```

The choices you can give for the direction *dir* are `"Forward"` and `"Backward"`, which refer to the integration forward and backward from the initial condition.

`"Forward"`

integration in the direction of increasing values of the temporal variable

`"Backward"`

integration in the direction of decreasing values of the temporal variables

`"Active"`

integration in the direction that is currently being integrated; typically, this value should only be called from method initialization that is used during an active integration

Integration direction specifications.

The output given by `NDSolve`ProcessSolution` is always given in terms of the dependent variables, either at a specific value of the independent variable, or interpolated over all of the saved values. This means that when a partial differential equation is being integrated, you will get results representing the dependent variables over the spatial variables.

This computes the solution to the heat equation from time $t = -1/4$ to $t = 2$.

```
In[13]:= state = First[NDSolve`ProcessEquations[{D[u[t, x], t] == D[u[t, x], x, x],
      u[0, x] == Cos[ $\pi/2$  x], u[t, 0] == 1, u[t, 1] == 0}, u, t, {x, 0, 1}]];
NDSolve`Iterate[state, {-1/4, 2}]
```

This gives the solution at $t = 2$.

```
In[15]:= NDSolve`ProcessSolutions[state, "Forward"]
Out[15]= {u[2., x] → InterpolatingFunction[{{0., 1.}}, <>][x],
      u(1,0)[2., x] → InterpolatingFunction[{{0., 1.}}, <>][x]}
```

The solution is given as an `InterpolatingFunction` object that interpolates over the spatial variable x .

This gives the solution at $t = -1/4$.

```
In[16]:= NDSolve`ProcessSolutions[state, "Backward"]

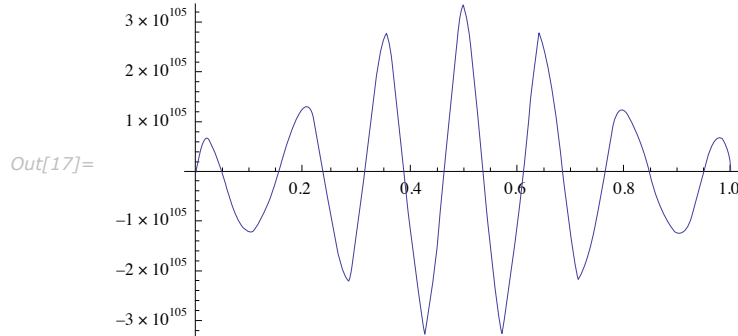
NDSolve::eerr: Warning: Scaled local spatial error estimate of 638.6378240455119`
      at t = -0.25 in the direction of independent variable x
      is much greater than prescribed error tolerance. Grid spacing with 15
      points may be too large to achieve the desired accuracy or precision. A
      singularity may have formed or you may want to specify a smaller
      grid spacing using the MaxStepSize or MinPoints method options. >>

Out[16]= {u[-0.25, x] → InterpolatingFunction[{{0., 1.}}, <>][x],
      u(1,0)[-0.25, x] → InterpolatingFunction[{{0., 1.}}, <>][x]}
```

When you process the current solution for partial differential equations, the spatial error estimate is checked. (It is not generally checked except when solutions are produced because doing so would be quite time consuming.) Since it is excessive, the `NDSolve::eerr` message is issued. The typical association of the word "backward" with the heat equation as implying instability gives a clue to what is wrong in this example.

Here is a plot of the solution at $t = 1/4$.

`In[17]:= Plot[Evaluate[u[-0.25, x] /. %], {x, 0, 1}]`



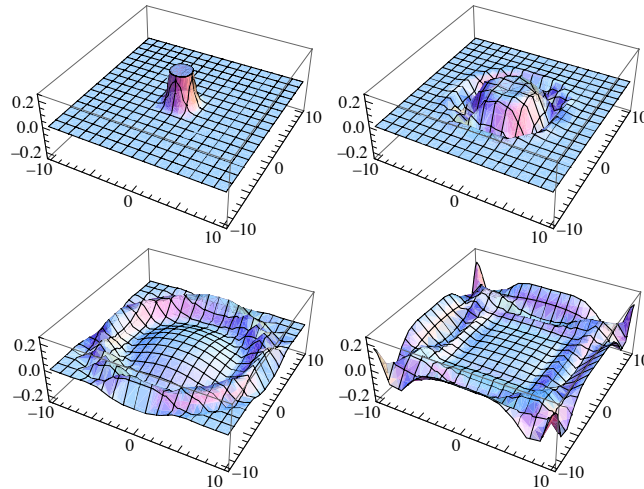
The plot of the solution shows that instability is indeed the problem.

Even though the heat equation example is simple enough to know that the solution backward in time is problematic, using `NDSolve`Iterate` and `NDSolve`ProcessSolutions` to monitor the solution of a PDE can be used to save computing a solution that turns out not to be as accurate as desired. Another simple form of monitoring follows.

Entering the following commands generates a sequence of plots showing the solution of a generalization of the sine-Gordon equation as it is being computed.

```
In[58]:= L = -10;
state = First[NDSolve`ProcessEquations[{D[u[t, x, y], t, t] ==
    D[u[t, x, y], x, x] + D[u[t, x, y], y, y] - Sin[u[t, x, y]],
    u[0, x, y] == Exp[-(x^2 + y^2)], Derivative[1, 0, 0][u][0, x, y] == 0,
    u[t, -L, y] == u[t, L, y], u[t, x, -L] == u[t, x, L]}, u, t, {x, -L, L},
    {y, -L, L}, Method -> {"MethodOfLines", "SpatialDiscretization" ->
    {"TensorProductGrid", "DifferenceOrder" -> "Pseudospectral"}}]];
GraphicsGrid[Partition[Table[
    NDSolve`Iterate[state,  $\tau$ ];
    Plot3D[Evaluate[u[ $\tau$ , x, y] /. NDSolve`ProcessSolutions[state, "Forward"]],
    {x, -L, L}, {y, -L, L}, PlotRange -> {-1/4, 1/4}],
    { $\tau$ , 0., 20., 5.}], 2]]
```

Out[60]=



When you monitor a solution in this way, it is usually possible to interrupt the computation if you see that the solution found is sufficient. You can still use the `NDSolve`StateData` object to get the solutions that have been computed.

NDSolve`StateData Methods

An `NDSolve`StateData` object contains a lot of information, but it is arranged in a manner which makes it easy to iterate solutions, and not in a manner which makes it easy to understand where the information is kept. However, sometimes you will want to get information from the state data object: for this reason several method functions have been defined to make accessing the information easy.

<code>state@"TemporalVariable"</code>	give the independent variable that the dependent variables (functions) depend on
<code>state@"DependentVariables"</code>	give a list of the dependent variables (functions) to be solved for
<code>state@"VariableDimensions"</code>	give the dimensions of each of the dependent variables (functions)
<code>state@"VariablePositions"</code>	give the positions in the solution vector for each of the dependent variables
<code>state@"VariableTransformation"</code>	give the transformation of variables from the original problem variables to the working variables
<code>state@"NumericalFunction"</code>	give the "NumericalFunction" object used to evaluate the derivatives of the solution vector with respect to the temporal variable t
<code>state@"ProcessExpression" [args, expr, dims]</code>	process the expression <i>expr</i> using the same variable transformations that NDSolve used to generate state to give a "NumericalFunction" object for numerically evaluating <i>expr</i> ; <i>args</i> are the arguments for the numerical function and should either be All or a list of arguments that are dependent variables of the system; <i>dims</i> should be Automatic or an explicit list giving the expected dimensions of the numerical function result
<code>state@"SystemSize"</code>	give the effective number of first-order ordinary differential equations being solved
<code>state@"MaxSteps"</code>	give the maximum number of steps allowed for iterating the differential equations
<code>state@"WorkingPrecision"</code>	give the working precision used to solve the equations
<code>state@"Norm"</code>	the scaled norm to use for gauging error

General method functions for an NDSolve`StateData object *state*.

Much of the available information depends on the current solution values. Each NDSolve`StateData object keeps solution information for solutions in both the forward and backward direction. At the initial condition these are the same, but once the problem has been iterated in either direction, these will be different.

<code>state@"CurrentTime" [dir]</code>	give the current value of the temporal variable in the integration direction <i>dir</i>
<code>state@"SolutionVector" [dir]</code>	give the current value of the solution vector in the integration direction <i>dir</i>
<code>state@"SolutionDerivativeVector" [dir]</code>	give the current value of the derivative with respect to the temporal variable of the solution vector in the integration direction <i>dir</i>
<code>state@"TimeStep" [dir]</code>	give the time step size for the next step in the integration direction <i>dir</i>
<code>state@"TimeStepsUsed" [dir]</code>	give the number of time steps used to get to the current time in the integration direction <i>dir</i>
<code>state@"MethodData" [dir]</code>	give the method data object used in the integration direction <i>dir</i>

Directional method functions for an `NDSolve`StateData` object *state*.

If the direction argument is omitted, the functions will return a list with the data for both directions (a list with a single element at the initial condition). Otherwise, the direction can be "Forward", "Backward", or "Active" as specified in the previous subsection.

Here is an `NDSolve`StateData` object for a solution of the nonlinear Schrodinger equation that has been computed up to $t = 1$.

```
In[24]:= state = First[NDSolve`ProcessEquations[
  {ID[u[t, x], t] == D[u[t, x], x, x] + Abs[u[t, x]]^2 u[t, x],
    u[0, x] == Sech[x] Exp[π I x], u[t, -15] == u[t, 15]},
  u, t, {x, -15, 15}, Method -> StiffnessSwitching];
NDSolve`Iterate[state, 1];
state
Out[24]= NDSolve`StateData[<0.,1.>]
```

"Current" refers to the most recent point reached in the integration.

This gives the current time in both the forward and backward directions.

```
In[27]:= state@"CurrentTime"
Out[27]= {0., 1.}
```

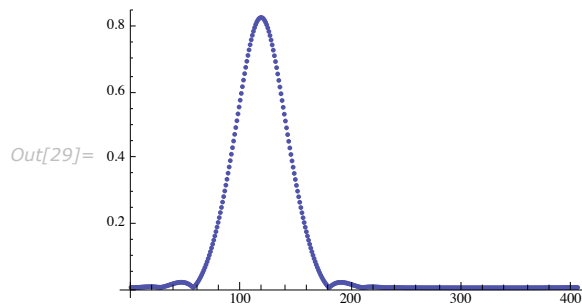
This gives the size of the system of ordinary differential equations being solved.

```
In[28]:= state@"SystemSize"
Out[28]= 400
```

The method functions are relatively low-level hooks into the data structure; they do little processing on the data returned to you. Thus, unlike `NDSolve`ProcessSolutions`, the solutions given are simply vectors of data points relating to the system of ordinary differential equations `NDSolve` is solving.

This makes a plot of the modulus of current solution in the forward direction.

```
In[29]:= ListPlot[Abs[state@SolutionVector["Forward"]]]
```



This plot does not show the correspondence with the x -grid values correctly. To get the correspondence with the spatial grid correctly, you must use `NDSolve`ProcessSolutions`.

There is a tremendous amount of control provided by these methods, but an exhaustive set of examples is beyond the scope of this documentation.

One of the most important uses of the information from an `NDSolve`StateData` object is to initialize integration methods. Examples are shown in "The `NDSolve` Method Plug-in Framework".

Utility Packages for Numerical Differential Equation Solving

InterpolatingFunctionAnatomy

`NDSolve` returns solutions as `InterpolatingFunction` objects. Most of the time, simply using these as functions does what is needed, but occasionally it is useful to access the data inside, which includes the actual values and points `NDSolve` computed when taking steps. The exact structure of an `InterpolatingFunction` object is arranged to make the data storage efficient

and evaluation at a given point fast. This structure may change between *Mathematica* versions, so code that is written in terms of accessing parts of `InterpolatingFunction` objects may not work with new versions of *Mathematica*. The `DifferentialEquations`InterpolatingFunctionAnatomy`` package provides an interface to the data in an `InterpolatingFunction` object that will be maintained for future *Mathematica* versions.

<code>InterpolatingFunctionDomain[ifun]</code>	return a list with the domain of definition for each of the dimensions of the <code>InterpolatingFunction</code> object <i>ifun</i>
<code>InterpolatingFunctionCoordinates[ifun]</code>	return a list with the coordinates at which data is specified in each of the dimensions for the <code>InterpolatingFunction</code> object <i>ifun</i>
<code>InterpolatingFunctionGrid[ifun]</code>	return the grid of points at which data is specified for the <code>InterpolatingFunction</code> object <i>ifun</i>
<code>InterpolatingFunctionValuesOnGrid[ifun]</code>	return the values that would be returned by evaluating the <code>InterpolatingFunction</code> object <i>ifun</i> at each of its grid points
<code>InterpolatingFunctionInterpolationOrder[ifun]</code>	return the interpolation order used for each of the dimensions for the <code>InterpolatingFunction</code> object <i>ifun</i>
<code>InterpolatingFunctionDerivativeOrder[ifun]</code>	return the order of the derivative of the base function for which values are specified when evaluating the <code>InterpolatingFunction</code> object <i>ifun</i>

Anatomy of `InterpolatingFunction` objects.

This loads the package.

```
In[21]:= Needs["DifferentialEquations`InterpolatingFunctionAnatomy`"];
```

One common situation where the `InterpolatingFunctionAnatomy` package is useful is when `NDSolve` cannot compute a solution over the full range of values that you specified, and you want to plot all of the solution that was computed to try to understand better what might have gone wrong.

Here is an example of a differential equation which cannot be computed up to the specified endpoint.

```
In[2]:= ifun = First[x /. NDSolve[{x'[t] == Exp[x[t]] - x[t], x[0] == 1}, x, {t, 0, 10}]]

NDSolve::ndsz:
At t == 0.5160191740198964`, step size is effectively zero; singularity or stiff system suspected. >>

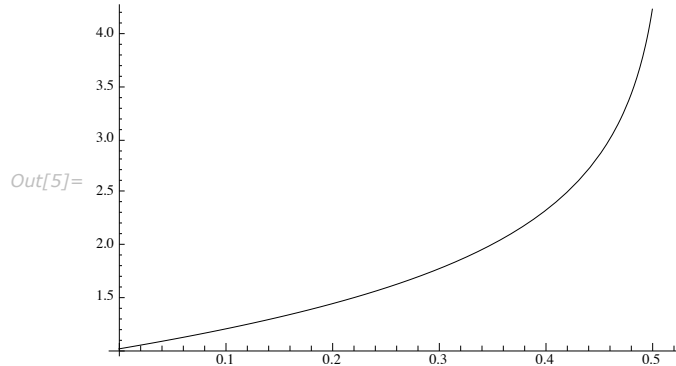
Out[2]= InterpolatingFunction[{{0., 0.516019}}, <>]
```

This gets the domain.

```
In[3]:= domain = InterpolatingFunctionDomain[ifun]
Out[3]= {{0., 0.516019}}
```

Once the domain has been returned in a list, it is easy to use `Part` to get the desired endpoints and make the plot.

```
In[4]:= {begin, end} = domain[[1]];
Plot[ifun[t], {t, begin, end}]
```

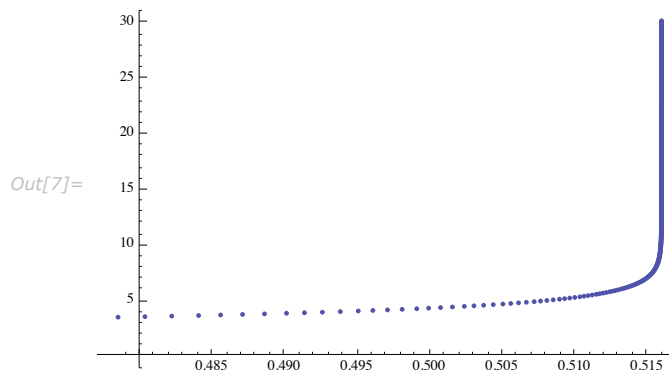


From the plot, it is quite apparent that a singularity has formed and it will not be possible to integrate the system any further.

Sometimes it is useful to see where `NDSolve` took steps. Getting the coordinates is useful for doing this.

This shows the values that `NDSolve` computed at each step it took. It is quite apparent from this that nearly all of the steps were used to try to resolve the singularity.

```
In[6]:= coords = First[InterpolatingFunctionCoordinates[ifun]];
ListPlot[Transpose[{coords, ifun[coords]}]]
```



The package is particularly useful for analyzing the computed solutions of PDEs.

With this initial condition, Burgers' equation forms a steep front.

```
In[8]:= mdfun =
  First[u /. NDSolve[{D[u[x, t], t] == 0.01 D[u[x, t], x, x] - u[x, t] D[u[x, t], x],
    u[0, t] == u[1, t], u[x, 0] == Sin[2 Pi x]}, u, {x, 0, 1}, {t, 0, 0.5}]]

NDSolve::ndsz:
  At t == 0.472151168326526`, step size is effectively zero; singularity or stiff system suspected. >>

NDSolve::eerr: Warning: Scaled local spatial error estimate of 9.135898727911074`*^12
  at t = 0.472151168326526` in the direction of independent variable x
  is much greater than prescribed error tolerance. Grid spacing with 27
  points may be too large to achieve the desired accuracy or precision. A
  singularity may have formed or you may want to specify a smaller
  grid spacing using the MaxStepSize or MinPoints method options. >>

Out[8]= InterpolatingFunction[{{..., 0., 1., ...}, {0., 0.472151}}, <>]
```

This shows the number of points used in each dimension.

```
In[9]:= Map[Length, InterpolatingFunctionCoordinates[mdfun]]
Out[9]= {27, 312}
```

This shows the interpolation order used in each dimension.

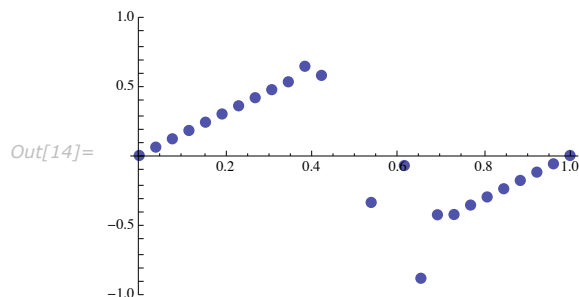
```
In[10]:= InterpolatingFunctionInterpolationOrder[mdfun]
Out[10]= {5, 3}
```

This shows that the inability to resolve the front has manifested itself as numerical instability.

```
In[11]:= Max[Abs[InterpolatingFunctionValuesOnGrid[mdfun]]]
Out[11]= 1.14928 × 1012
```

This shows the values computed at the spatial grid points at the endpoint of the temporal integration.

```
In[12]:= end = InterpolatingFunctionDomain[mdfun][[2, -1]];
X = InterpolatingFunctionCoordinates[mdfun][[1]];
ListPlot[Transpose[{X, mdfun[X, end]}],
  PlotStyle -> PointSize[.025], PlotRange -> {-1, 1}]
```

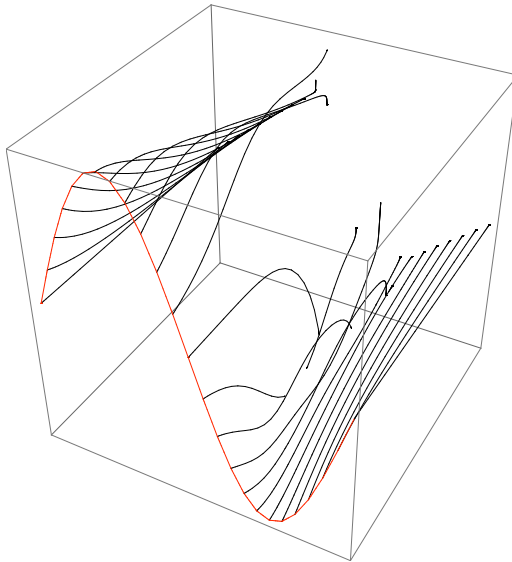


It is easily seen from the point plot that the front has not been resolved.

This makes a 3D plot showing the time evolution for each of the spatial grid points. The initial condition is shown in red.

```
In[15]:= Show[Graphics3D[{Map[Line, MapThread[Append, {InterpolatingFunctionGrid[mdfun],
    InterpolatingFunctionValuesOnGrid[mdfun]}, 2]],
    {RGBColor[1, 0, 0], Line[Transpose[{X, 0.X, mdfun[X, 0.]}}]}],
    BoxRatios -> {1, 1, 1}, PlotRange -> {All, All, {-1, 1}}]
```

Out[15]=



When a derivative of an `InterpolatingFunction` object is taken, a new `InterpolatingFunction` object is returned that gives the requested derivative when evaluated at a point. The `InterpolatingFunctionDerivativeOrder` is a way of determining what derivative will be evaluated.

The derivative returns a new `InterpolatingFunction` object.

```
In[16]:= dmdfun = Derivative[0, 1][mdfun]
```

```
Out[16]= InterpolatingFunction[{{..., 0., 1., ...}, {0., 0.472151}}, <>]
```

This shows what derivative will be evaluated.

```
In[17]:= InterpolatingFunctionDerivativeOrder[dmdfun]
```

```
Out[17]= Derivative[0, 1]
```

NDSolveUtilities

A number of utility routines have been written to facilitate the investigation and comparison of various `NDSolve` methods. These functions have been collected in the package `DifferentialEquations`NDSolveUtilities``.

<code>CompareMethods[sys, refsol, methods, opts]</code>	return statistics for various methods applied to the system <i>sys</i>
<code>FinalSolutions[sys, sols]</code>	return the solution values at the end of the numerical integration for various solutions <i>sols</i> corresponding to the system <i>sys</i>
<code>InvariantErrorPlot[invts, dvars, ivar, sol, opts]</code>	return a plot of the error in the invariants <i>invts</i> for the solution <i>sol</i>
<code>RungeKuttaLinearStabilityFunction[amat, bvec, var]</code>	return the linear stability function for the Runge-Kutta method with coefficient matrix <i>amat</i> and weight vector <i>bvec</i> using the variable <i>var</i>
<code>StepDataPlot[sols, opts]</code>	return plots of the step sizes taken for the solutions <i>sols</i> on a logarithmic scale

Functions provided in the `NDSolveUtilities` package.

This loads the package.

```
In[18]:= Needs["DifferentialEquations`NDSolveUtilities`"]
```

A useful means of analyzing Runge-Kutta methods is to study how they behave when applied to a scalar linear test problem (see the package `FunctionApproximations.m`).

This assigns the (exact or infinitely precise) coefficients for the 2-stage implicit Runge-Kutta Gauss method of order 4.

```
In[19]:= {amat, bvec, cvec} = NDSolve`ImplicitRungeKuttaGaussCoefficients[4, Infinity]
```

```
Out[19]= {{ {1/4, 1/12 (3 - 2 Sqrt[3])}, {1/12 (3 + 2 Sqrt[3]), 1/4}}, {1/2, 1/2}, {1/6 (3 - Sqrt[3]), 1/6 (3 + Sqrt[3])} }
```

This computes the linear stability function, which corresponds to the (2,2) Padé approximation to the exponential at the origin.

```
In[20]:= RungeKuttaLinearStabilityFunction[amat, bvec, z]
```

```
Out[20]= (1 + z/2 + z^2/12) / (1 - z/2 + z^2/12)
```

Examples of the functions `CompareMethods`, `FinalSolutions`, `RungeKuttaLinearStabilityFunction`, and `StepDataPlot` can be found within "ExplicitRungeKutta Method for `NDSolve`".

Examples of the function `InvariantErrorPlot` can be found within "Projection Method for `NDSolve`".

InvariantErrorPlot Options

The function `InvariantErrorPlot` has a number of options that can be used to control the form of the result.

<i>option name</i>	<i>default value</i>	
<code>InvariantDimensions</code>	<code>Automatic</code>	specify the dimensions of the invariants
<code>InvariantErrorFunction</code>	<code>Abs[Subtract[#1,#2]] &</code>	specify the function to use for comparing errors
<code>InvariantErrorSampleRate</code>	<code>Automatic</code>	specify how often errors are sampled

Options of the function `InvariantErrorPlot`.

The default value for `InvariantDimensions` is to determine the dimensions from the structure of the input, `Dimensions[invt]`.

The default value for `InvariantErrorFunction` is a function to compute the absolute error.

The default value for `InvariantErrorSampleRate` is to sample all points if there are less than 1000 steps taken. Above this threshold a logarithmic sample rate is used.

Advanced Numerical Differential Equation Solving in Mathematica: References

- [AP91] Ascher U. and L. Petzold. "Projected Implicit Runge-Kutta Methods for Differential Algebraic Equations" *SIAM J. Numer. Anal.* 28 (1991): 1097-1120
- [AP98] Ascher U. and L. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM Press (1998)
- [ARPACK98] Lehoucq, R. B., D. C. Sorensen, and C. Yang. ARPACK Users' Guide, Solution of Large-Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods, SIAM (1998)
- [ATLAS00] Whaley R. C., A. Petitet, and J. J. Dongarra. "Automated Empirical Optimization of Software and the ATLAS Project" Available electronically from <http://math-atlas.sourceforge.net/>
- [BD83] Bader G. and P. Deuflhard. "A Semi-Implicit Mid-Point Rule for Stiff Systems of Ordinary Differential Equations" *Numer. Math* 41 (1983): 373-398
- [BS97] Bai Z. and G. W. Stewart. "SRRIT: a Fortran Subroutine to Calculate the Dominant Invariant Subspace of a Nonsymmetric Matrix" *ACM Trans. Math. Soft.* 23 4 (1997): 494-513
- [BG94] Benettin G. and A. Giorgilli. "On the Hamiltonian Interpolation of Near to the Identity Symplectic Mappings with Application to Symplectic Integration Algorithms" *J. Stat. Phys.* 74 (1994): 1117-1143
- [BZ65] Berezin I. S. and N. P. Zhidkov. *Computing Methods, Volume 2*. Pergamon (1965)
- [BM02] Blanes S. and P. C. Moan. "Practical Symplectic Partitioned Runge-Kutta and Runge-Kutta-Nyström Methods" *J. Comput. Appl. Math.* 142 (2002): 313-330
- [BCR99a] Blanes S., F. Casas, and J. Ros. "Symplectic Integration with Processing: A General Study" *SIAM J. Sci. Comput.* 21 (1999): 711-727
- [BCR99b] Blanes S., F. Casas, and J. Ros. "Extrapolation of Symplectic Integrators" Report DAMTP NA09, Cambridge University (1999)
- [BS89a] Bogacki P. and L. F. Shampine. "A 3(2) Pair of Runge-Kutta Formulas" *Appl. Math. Letters* 2 (1989): 1-9

- [BS89b] Bogacki P. and L. F. Shampine. "An Efficient Runge-Kutta (4, 5) Pair" Report 89-20, Math. Dept. Southern Methodist University, Dallas, Texas (1989)
- [BGS93] Brankin R. W., I. Gladwell and L. F. Shampine. "RKSUITE: A Suite of Explicit Runge-Kutta Codes" In *Contributions to Numerical Mathematics*, R. P. Agarwal, ed., 41-53 (1993)
- [BCP89] Brenan K., S. Campbell, and L. Petzold. *Numerical Solutions of Initial-Value Problems in Differential-Algebraic Equations*. Elsevier Science Publishing (1989)
- [BHP94] Brown P. N., A. C. Hindmarsh, and L. R. Petzold. "Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems" *SIAM J. Sci. Comput.* 15 (1994): 1467-1488
- [BHP98] Brown P. N., A. C. Hindmarsh, and L. R. Petzold. "Consistent Initial Condition Calculation for Differential-Algebraic Systems" *SIAM J. Sci. Comput.* 19 (1998): 1495-1512
- [B87] Butcher J. C. *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. John Wiley (1987)
- [B90] Butcher J. C. "Order, Stepsize and Stiffness Switching" *Computing*. 44 3, (1990): 209-220
- [BS64] Bulirsch R. and J. Stoer. "Fehlerabschätzungen und Extrapolation mit Rationalen Funktionen bei Verfahren vom Richardson-Typus" *Numer. Math.* 6 (1964): 413-427
- [CIZ97] Calvo M. P., A. Iserles, and A. Zanna. "Numerical Solution of Isospectral Flows" *Math. Comp.* 66, no. 220 (1997): 1461-1486
- [CIZ99] Calvo M. P., A. Iserles, and A. Zanna. "Conservative Methods for the Toda Lattice Equations" *IMA J. Numer. Anal.* 19 (1999): 509-523
- [CR91] Candy J. and R. Rozmus. "A Symplectic Integration Algorithm for Separable Hamiltonian Functions" *J. Comput. Phys.* 92 (1991): 230-256
- [CH94] Cohen S. D. and A. C. Hindmarsh. *CVODE User Guide*. Lawrence Livermore National Laboratory report UCRL-MA-118618, September 1994
- [CH96] Cohen S. D. and A. C. Hindmarsh. "CVODE, a Stiff/Nonstiff ODE Solver in C" *Computers in Physics* 10, no. 2 (1996): 138-143
- [C87] Cooper G. J. "Stability of Runge-Kutta Methods for Trajectory Problems" *IMA J. Numer. Anal.* 7 (1987): 1-13
- [DP80] Dormand J. R. and P. J. Prince. "A Family of Embedded Runge-Kutta Formulae" *J. Comp. Appl. Math.* 6 (1980): 19-26

- [DL01] Del Buono N. and L. Lopez. "Runge-Kutta Type Methods Based on Geodesics for Systems of ODEs on the Stiefel Manifold" *BIT* 41 (5 (2001): 912-923
- [D83] Deuflhard P. "Order and Step Size Control in Extrapolation Methods" *Numer. Math.* 41 (1983): 399-422
- [D85] Deuflhard P. "Recent Progress in Extrapolation Methods for Ordinary Differential Equations" *SIAM Rev.* 27 (1985): 505-535
- [DN87] Deuflhard P. and U. Nowak. "Extrapolation Integrators for Quasilinear Implicit ODEs" In *Large-scale scientific computing*, (P. Deuflhard and B. Engquist eds.) Birkhäuser, (1987)
- [DS93] Duff I. S. and J. A. Scott. "Computing Selected Eigenvalues of Sparse Unsymmetric Matrices Using Subspace Iteration" *ACM Trans. Math. Soft.* 19 2, (1993): 137-159
- [DHZ87] Deuflhard P., E. Hairer, and J. Zugck. "One-Step and Extrapolation Methods for Differential-Algebraic Systems" *Numer. Math.* 51 (1987): 501-516
- [DRV94] Dieci L., R. D. Russel, and E. S. Van Vleck. "Unitary Integrators and Applications to Continuous Orthonormalization Techniques" *SIAM J. Num. Anal.* 31 (1994): 261-281
- [DV99] Dieci L. and E. S. Van Vleck. "Computation of Orthonormal Factors for Fundamental Solution Matrices" *Numer. Math.* 83 (1999): 599-620
- [DLP98a] Diele F., L. Lopez, and R. Peluso. "The Cayley Transform in the Numerical Solution of Unitary Differential Systems" *Adv. Comput. Math.* 8 (1998): 317-334
- [DLP98b] Diele F., L. Lopez, and T. Politi. "One Step Semi-Explicit Methods Based on the Cayley Transform for Solving Isospectral Flows" *J. Comput. Appl. Math.* 89 (1998): 219-223
- [ET92] Earn D. J. D. and S. Tremaine. "Exact Numerical Studies of Hamiltonian Maps: Iterating without Roundoff Error" *Physica D.* 56 (1992): 1-22
- [F69] Fehlberg E. "Low-Order Classical Runge-Kutta Formulas with Step Size Control and Their Application to Heat Transfer Problems" NASA Technical Report 315, 1969 (extract published in *Computing* 6 (1970): 61-71)
- [FR90] Forest E. and R. D. Ruth. "Fourth Order Symplectic Integration" *Physica D.* 43 (1990): 105-117
- [F92] Fornberg B. "Fast Generation of Weights in Finite Difference Formulas" In *Recent Developments in Numerical Methods and Software for ODEs/DAEs/PDEs* (G. D. Byrne and W. E. Schiesser eds.). World Scientific (1992)

- [F96a] Fornberg B. *A Practical Guide to Pseudospectral Methods*. Cambridge University Press (1996)
- [F98] Fornberg B. "Calculation of Weights in Finite Difference Formulas" *SIAM Review* 40, no. 3 (1998): 685-691 (Available in PDF)
- [F96b] Fukushima T. "Reduction of Round-off Errors in the Extrapolation Methods and its Application to the Integration of Orbital Motion" *Astron. J.* 112, no. 3 (1996): 1298-1301
- [G51] Gill S. "A Process for the Step-by-Step Integration of Differential Equations in an Automatic Digital Computing Machine" *Proc. Cambridge Philos. Soc.* 47 (1951): 96-108
- [G65] Gragg W. B. "On Extrapolation Algorithms for Ordinary Initial Value Problems" *SIAM J. Num. Anal.* 2 (1965): 384-403
- [GØ84] Gear C. W. and O. Østerby. "Solving Ordinary Differential Equations with Discontinuities" *ACM Trans. Math. Soft.* 10 (1984): 23-44
- [G91] Gustafsson K. "Control Theoretic Techniques for Step Size Selection in Explicit Runge-Kutta Methods" *ACM Trans. Math. Soft.* 17, (1991): 533-554
- [G94] Gustafsson K. "Control Theoretic Techniques for Step Size Selection in Implicit Runge-Kutta Methods" *ACM Trans. Math. Soft.* 20, (1994): 496-517
- [GMW81] Gill P., W. Murray, and M. Wright. *Practical Optimization*. Academic Press (1981)
- [GDC91] Gladman B., M. Duncan, and J. Candy. "Symplectic Integrators for Long-Term Integrations in Celestial Mechanics" *Celest. Mech.* 52 (1991): 221-240
- [GSB87] Gladwell I., L. F. Shampine and R. W. Brankin. "Automatic Selection of the Initial Step Size for an ODE Solver" *J. Comp. Appl. Math.* 18 (1987): 175-192
- [GVL96] Golub G. H. and C. F. Van Loan. *Matrix Computations*, 3rd ed. Johns Hopkins University Press (1996)
- [H83] Hindmarsh A. C. "ODEPACK, A Systematized Collection of ODE Solvers" In Scientific Computing (R. S. Stepleman et al. eds.) Vol. 1 of IMACS Transactions on Scientific Computation (1983): 55-64
- [H94] Hairer E. "Backward Analysis of Numerical Integrators and Symplectic Methods" *Annals of Numerical Mathematics* 1 (1984): 107-132
- [H97] Hairer E. "Variable Time Step integration with Symplectic Methods" *Appl. Numer. Math.* 25 (1997): 219-227

[H00] Hairer E. "Symmetric Projection Methods for Differential Equations on Manifolds" *BIT* 40, no. 4 (2000): 726-734

[HL97] Hairer E. and Ch. Lubich. "The Life-Span of Backward Error Analysis for Numerical Integrators" *Numer. Math.* 76 (1997): 441-462. Erratum:
<http://www.unige.ch/math/folks/hairer>

[HL88a] Hairer E. and Ch. Lubich. "Extrapolation at Stiff Differential Equations" *Numer. Math.* 52 (1988): 377-400

[HL88b] Hairer E. and Ch. Lubich. "On Extrapolation Methods for Stiff and Differential-Algebraic Equations" *Teubner Texte zur Mathematik* 104 (1988): 64-73

[HO90] Hairer E. and A. Ostermann. "Dense Output for Extrapolation Methods" *Numer. Math.* 58 (1990): 419-439

[HW96] Hairer E. and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, 2nd ed. Springer-Verlag (1996)

[HW99] Hairer E. and G. Wanner. "Stiff Differential Equations Solved by Radau Methods" *J. Comp. Appl. Math.* 111 (1999): 93-111

[HLW02] Hairer E., Ch. Lubich, and G. Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. Springer-Verlag (2002)

[HNW93] Hairer E., S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd ed. Springer-Verlag (1993)

[H97] Higham D. "Time-Stepping and Preserving Orthonormality" *BIT* 37, no. 1 (1997): 24-36

[H89] Higham N. J. "Matrix Nearness Problems and Applications" In *Applications of Matrix Theory* (M. J. C. Gover and S. Barnett eds.). Oxford University Press (1989): 1-27

[H96] Higham N. J. *Accuracy and Stability of Numerical Algorithms*. SIAM (1996)

[H83] Hindmarsh A. C. "ODEPACK, A Systematized Collection of ODE Solvers" In *Scientific Computing* (R. S. Stepleman et al. eds). North-Holland (1983): 55-64

[HT99] Hindmarsh A. and A. Taylor. *User Documentation for IDA: A Differential-Algebraic Equation Solver for Sequential and Parallel Computers*. Lawrence Livermore National Laboratory report, UCRL-MA-136910, December 1999.

- [KL97] Kahan W. H. and R. C. Li. "Composition Constants for Raising the Order of Unconventional Schemes for Ordinary Differential Equations" *Math. Comp.* 66 (1997): 1089-1099
- [K65] Kahan W. H. "Further Remarks on Reducing Truncation Errors" *Comm. ACM.* 8 (1965): 40
- [K93] Koren I. *Computer Arithmetic Algorithms*. Prentice Hall (1993)
- [L87] Lambert J. D. *Numerical Methods for Ordinary Differential Equations*. John Wiley (1987)
- [LS96] Lehoucq, R. B and J. A. Scott. "An Evaluation of Software for Computing Eigenvalues of Sparse Nonsymmetric Matrices." Preprint MCS-P547-1195, Argonne National Laboratory, (1996)
- [LAPACK99] Anderson E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorenson. *LAPACK Users' Guide*, 3rd ed. SIAM (1999)
- [M68] Marchuk G. "Some Applications of Splitting-Up Methods to the Solution of Mathematical Physics Problems" *Aplikace Matematiky* 13 (1968): 103-132
- [MR99] Marsden J. E. and T. Ratiu. *Introduction to Mechanics and Symmetry: Texts in Applied Mathematics, Vol. 17*, 2nd ed. Springer-Verlag (1999)
- [M93] McLachlan R. I. "Explicit Lie-Poisson Integration and the Euler Equations" *Phys. Rev. Lett.* 71 (1993): 3043-3046
- [M95a] McLachlan R. I. "On the Numerical Integration of Ordinary Differential Equations by Symmetric Composition Methods" *SIAM J. Sci. Comp.* 16 (1995): 151-168
- [M95b] McLachlan R. I. "Composition Methods in the Presence of Small Parameters" *BIT* 35 (1995): 258-268
- [M01] McLachlan R. I. "Families of High-Order Composition Methods" *Numerical Algorithms* 31 (2002): 233-246
- [MA92] McLachlan R. I. and P. Atela. "The Accuracy of Symplectic Integrators" *Nonlinearity* 5 (1992): 541-562
- [MQ02] McLachlan R. I. and G. R. W. Quispel. "Splitting Methods" *Acta Numerica* 11 (2002): 341-434
- [MG80] Mitchell A. and D. Griffiths. *The Finite Difference Method in Partial Differential Equations*. John Wiley and Sons (1980)

- [M65a] Møller O. "Quasi Double-Precision in Floating Point Addition" *BIT* 5 (1965): 37-50
- [M65b] Møller O. "Note on Quasi Double-Precision" *BIT* 5 (1965): 251-255
- [M97] Murua A. "On Order Conditions for Partitioned Symplectic Methods" *SIAM J. Numer. Anal.* 34, no. 6 (1997): 2204-2211
- [MS99] Murua A. and J. M. Sanz-Serna. "Order Conditions for Numerical Integrators Obtained by Composing Simpler Integrators" *Phil. Trans. Royal Soc. A* 357 (1999): 1079-1100
- [M04] Moler C. B. *Numerical Computing with MATLAB*. SIAM (2004)
- [Na79] Na T. Y. *Computational Methods in Engineering: Boundary Value Problems*. Academic Press (1979)
- [OS92] Okunbor D. I. and R. D. Skeel. "Explicit Canonical Methods for Hamiltonian Systems" *Math. Comp.* 59 (1992): 439-455
- [O95] Olsson H. "Practical Implementation of Runge-Kutta Methods for Initial Value Problems" Licentiate thesis, Department of Computer Science, Lund University, 1995
- [O98] Olsson H. "Runge-Kutta Solution of Initial Value Problems: Methods, Algorithms and Implementation" PhD Thesis, Department of Computer Science, Lund University, 1998
- [OS00] Olsson H. and G. Söderlind. "The Approximate Runge-Kutta Computational Process" *BIT* 40 (2 (2000): 351-373
- [P83] Petzold L. R. "Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations" *SIAM J. Sci. Stat. Comput.* 4 (1983): 136-148
- [QSS00] Quarteroni A., R. Sacco, and F. Saleri. *Numerical Mathematics*. Springer-Verlag (2000)
- [QV94] Quarteroni A. and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer-Verlag (1994)
- [QT90] Quinn T. and S. Tremaine. "Roundoff Error in Long-Term Planetary Orbit Integrations" *Astron. J.* 99, no. 3 (1990): 1016-1023
- [R93] Reich S. "Numerical Integration of the Generalized Euler Equations" *Tech. Rep.* 93-20, Dept. Comput. Sci. Univ. of British Columbia (1993)
- [R99] Reich S. "Backward Error Analysis for Numerical Integrators" *SIAM J. Num. Anal.* 36 (1999): 1549-1570

- [R98] Rubinstein B. "Numerical Solution of Linear Boundary Value Problems" *Mathematica MathSource* package, <http://library.wolfram.com/database/MathSource/2127/>
- [RM57] Richtmeyer R. and K. Morton. *Difference Methods for Initial Value Problems*. Krieger Publishing Company (1994) (original edition 1957)
- [R87] Robertson B. C. "Detecting Stiffness with Explicit Runge-Kutta Formulas" Report 193/87, Dept. Comp. Sci. University of Toronto (1987)
- [S84b] Saad Y. "Chebyshev Acceleration Techniques for Solving Nonsymmetric Eigenvalue Problems" *Math. Comp.* 42, (1984): 567-588
- [SC94] Sanz-Serna J. M. and M. P. Calvo. *Numerical Hamiltonian Problems: Applied Mathematics and Mathematical Computation*, no. 7. Chapman and Hall (1994)
- [S91] Schiesser W. *The Numerical Method of Lines*. Academic Press (1991)
- [S77] Shampine L. F. "Stiffness and Non-Stiff Differential Equation Solvers II: Detecting Stiffness with Runge-Kutta Methods" *ACM Trans. Math. Soft.* 3 1 (1977): 44-53
- [S83] Shampine L. F. "Type-Insensitive ODE Codes Based on Extrapolation Methods" *SIAM J. Sci. Stat. Comput.* 4 1 (1984): 635-644
- [S84a] Shampine L. F. "Stiffness and the Automatic Selection of ODE Code" *J. Comp. Phys.* 54 1 (1984): 74-86
- [S86] Shampine L. F. "Conservation Laws and the Numerical Solution of ODEs" *Comp. Maths. Appl.* 12B (1986): 1287-1296
- [S87] Shampine L. F. "Control of Step Size and Order in Extrapolation Codes" *J. Comp. Appl. Math.* 18 (1987): 3-16
- [S91] Shampine L. F. "Diagnosing Stiffness for Explicit Runge-Kutta Methods" *SIAM J. Sci. Stat. Comput.* 12 2 (1991): 260-272
- [S94] Shampine L. F. *Numerical Solution of Ordinary Differential Equations*. Chapman and Hall (1994)
- [SB83] Shampine L. F. and L. S. Baca. "Smoothing the Extrapolated Midpoint Rule" *Numer. Math.* 41 (1983): 165-175
- [SG75] Shampine L. F. and M. Gordon. *Computer Solutions of Ordinary Differential Equations*. W. H. Freeman (1975)

- [SR97] Shampine L. F. and M. W Reichelt. *Solving ODEs with MATLAB*. *SIAM J. Sci. Comp.* 18-1 (1997): 1-22
- [ST00] Shampine L. F. and S. Thompson. "Solving Delay Differential Equations with dde23" Available electronically from <http://www.runet.edu/~thompson/webddes/tutorial.pdf>
- [ST01] Shampine L. F. and S. Thompson. "Solving DDEs in MATLAB" *Appl. Numer. Math.* 37 (2001): 441-458
- [SGT03] Shampine L. F., I. Gladwell, and S. Thompson. *Solving ODEs with MATLAB*. Cambridge University Press (2003)
- [SBB83] Shampine L. F., L. S. Baca, and H. J. Bauer. "Output in Extrapolation Codes" *Comp. and Maths. with Appl.* 9 (1983): 245-255
- [SS03] Sofroniou M. and G. Spaletta. "Increment Formulations for Rounding Error Reduction in the Numerical Solution of Structured Differential Systems" *Future Generation Computer Systems* 19, no. 3 (2003): 375-383
- [SS04] Sofroniou M. and G. Spaletta. "Construction of Explicit Runge-Kutta Pairs with Stiffness Detection" *Mathematical and Computer Modelling*, special issue on The Numerical Analysis of Ordinary Differential Equations, 40, no. 11-12 (2004): 1157-1169
- [SS05] Sofroniou M. and G. Spaletta. "Derivation of Symmetric Composition Constants for Symmetric Integrators" *Optimization Methods and Software* 20, no. 4-5 (2005): 597-613
- [SS06] Sofroniou M. and G. Spaletta. "Hybrid Solvers for Splitting and Composition Methods" *J. Comp. Appl. Math.*, special issue from the International Workshop on the Technological Aspects of Mathematics, 185, no. 2 (2006): 278-291
- [S84c] Sottas G. "Dynamic Adaptive Selection Between Explicit and Implicit Methods When Solving ODEs" Report, Sect. de math, University of Genève, 1984
- [S07] Sprott J.C. "A Simple Chaotic Delay Differential Equation", *Phys. Lett. A.* 366 (2007): 397-402
- [S68] Strang G. "On the Construction of Difference Schemes" *SIAM J. Num. Anal.* 5 (1968): 506-517
- [S70] Stetter H. J. "Symmetric Two-Step Algorithms for Ordinary Differential Equations" *Computing* 5 (1970): 267-280

- [S01] Stewart G. W. "A Krylov-Schur Algorithm for Large Eigenproblems" *SIAM J. Matrix Anal. Appl.* 23 3, (2001): 601-614
- [SJ81] Stewart W. J. and A. Jennings. "LOPSI: A Simultaneous Iteration Method for Real Matrices" *ACM Trans. Math. Soft.* 7 2, (1981): 184-198
- [S90] Suzuki M. "Fractal Decomposition of Exponential Operators with Applications to Many-Body Theories and Monte Carlo Simulations" *Phys. Lett. A* 146 (1990): 319-323
- [SLEPc05] Hernandez V., J. E. Roman and V. Vidal "SRRIT: a Fortran Subroutine to Calculate the Dominant Invariant Subspace of a Nonsymmetric Matrix" *ACM Trans. Math. Soft.* 31 3, (2005): 351-362
- [T59] Trotter H. F. "On the Product of Semi-Group Operators" *Proc. Am. Math. Soc.* 10 (1959): 545-551
- [TZ08] Tang, Z. H. and Zou, X. "Global attractivity in a predator-prey System with Pure Delays", *Proc. Edinburgh Math. Soc.* 51 (2008): 495-508
- [V78] Verner J. H. "Explicit Runge-Kutta Methods with Estimates of The Local Truncation Error" *SIAM J. Num. Anal.* 15 (1978): 772-790.
- [V79] Vitasek E. "A-Stability and Numerical Solution of Evolution Problems" *IAC 'Mauro Picone', Series III* 186 (1979): 42
- [W76] Whitham G. B. *Linear and Nonlinear Waves*. John Wiley and Sons (1976)
- [WH91] Wisdom J. and M. Holman. "Symplectic Maps for the N-Body Problem" *Astron. J.* 102 (1991): 1528-1538
- [Y90] Yoshida H. "Construction of High Order Symplectic Integrators" *Phys. Lett. A.* 150 (1990): 262-268
- [Z98] Zanna A. "On the Numerical Solution of Isospectral Flows" Ph.D. Thesis, Cambridge University, 1998
- [Z72] Zeeman E. C. "Differential Equations for the Heartbeat and Nerve Impulse". In *Towards a Theoretical Biology* (C. H. Waddington, ed.). Edinburgh Univeristy Press, 4 (1972): 8-67
- [Z06] Zennaro M. "The numerical solution of delay differential equations", Lecture notes, Dobbiaco Summer Chool on Delay Differential Equations and Applications (2006)

