CUBO *a Mathematical Journal*
Vol. 6, Nº 1, (511-569). March 2004.

Numerical Solutions of Ordinary Differential Equations

Michael J. Mezzino, Jr.
University of Houston - Clear Lake
Houston, Texas 77058
U.S.A.

ABSTRACT

This paper explains the methods of **Runge-Kutta,Adams-Bashforth** and **Bulirsch-Stoer** for approximating the solution of a first order initial value problem, and the **Numerov** method for approximating the solution for a class of second-order linear initial value problems. *Mathematica* is used to derive some of the formulas, solve the example equations, and plot the solutions.

Most differential equations do not have explicit solutions. Even if a differential equation has an explicit solution, the solution may be so complicated that it is useless for practical purposes.

On the other hand, we can try to find an approximation to the solution of a single differential equation with an initial condition. Later, we will investigate numerical solutions to systems of ordinary differential equations. For example, suppose we are given a first order initial value problem

$$\begin{cases} y' = f(t,y), \\ y(a) = y_0, \end{cases} \tag{0.1}$$

on an interval $a \leq t \leq b$. Although we may not be able to obtain a formula for the solution of (0.1), we can subdivide the interval as

$$a = t_0 < t_1 < \cdots < t_N = b$$

and try to assign approximate values Y_n to t_n for $n = 1, \ldots, N$. Instead of a formula we will have an approximation to the solution of (0.1) expressed as a table of the Y_n's in terms of the t_n's. By graphing the table we can visualize the solution.

One of the main goals in this paper is to explain the methods of **Runge-Kutta**, **Adams-Bashforth** and **Bulirsch-Stoer** for approximating the solution of a first order initial value problem, and the **Numerov** method for approximating the solution for a class of second-order linear initial value problems. These complicated algorithms need some preparation, however. For that reason we begin by considering simpler methods of numerical approximations to solutions to differential equations. The simplest of all is the Euler method, which we describe in Section 1. The next simplest is the Heun method, which is derived from the Euler method in Section 2. Section 3 is devoted to the Runge-Kutta method. Section 4 discusses the error analysis, with a complete proof in the case of Euler's method. The development continues with the presentation of the multistep methods of Milne and Adams-Bashforth in Section 5. The extrapolation techniques of Bulirsch-Stoer are presented in Section 6. These and other solution methods available in `ODE` are discussed in Section 7. We compare them briefly, and then in Section 8 we explain the details of how the Euler, Heun and Runge-Kutta methods are implemented in `ODE`. In Section 9 we discuss *Mathematica*'s built-in numerical solver **`NDSolve`**. A brief discussion of implicit methods and stiff equations is presented in Section 10 with stability presented in Section 11. Adaptive step size is treated in Section 12. In Section 13 we present the theory and *Mathematica* implementation of the Numerov method for the numerical solution of second order differential equations of the form $y'' = f(t)y + g(t)$. Finally, in Sections 14 and 15 we show how these algorithms can be extended to systems of differential equations.

`ODE` is a comprehensive *Mathematica* package written for students and instructors of ordinary differential equations. It can be found at

http://math.cl.uh.edu/ode/htmldocs/newode.htm.

The complete documentation for this package can be found in two forms at

http://math.cl.uh.edu/ode/3x/ref/ref.htm.

First, as a hypertext linked document and second, as a set of postscript files suitable for printing. It is not necessary to understand *Mathematica* or `ODE` to read this paper. *Mathematica* together with `ODE` are used to derive some of the results and to perform all of the computations and produce all of the plots.

1 The Euler Method

The Euler[1] method is the simplest method used to find numerical solutions to differential equations. In spite of the fact that it is rarely used in practice, we need to study it because it serves as a model for more complicated methods such as the Runge-Kutta method that we shall study in Section 3.

[1] Leonhard Euler (1707–1783). Born in Basel, Switzerland. Worked most of his life in Berlin and Saint Petersburg. Was the most prolific mathematician of all time. Contributed greatly to the evolution and systematization of analysis, in particular to the founding of the calculus of variations and the theories of differential equations, functions of complex variables, and special functions, and also laid the foundations of number theory as a rigorous discipline. Concerned himself with applications of mathematics to fields as diverse as lotteries, hydraulic systems, shipbuilding and navigation, actuarial science, demography, fluid mechanics, astronomy, and ballistics.

The Euler method can be explained as follows. The objective is to construct an approximation to the solution of the initial value problem (0.1) for $a \leq t \leq b$. We divide the interval $a \leq t \leq b$ into equal subintervals:

$$a = t_0 < t_1 < \cdots < t_N = b,$$

where $h = t_{n+1} - t_n$ is called the **step size**. Let us first suppose that (0.1) has a solution $y(t)$, which we assume to be twice differentiable. We need the finite Taylor expansion

$$y(t_1) = y(t_0) + (t_1 - t_0)y'(t_0) + \frac{(t_1 - t_0)^2}{2} y''(\xi_0), \tag{1.2}$$

where $t_0 < \xi_0 < t_1$. Using (0.1), we can rewrite (1.2) as

$$y(t_1) = y(t_0) + h\,f(t_0, y_0) + \frac{h^2}{2} y''(\xi_0). \tag{1.3}$$

Now if h is a positive number less than 1, the quantity h^2 is even smaller. Therefore, we have the approximate equality

$$y(t_1) \approx y(t_0) + h\,f(t_0, y_0). \tag{1.4}$$

The initial condition of (0.1) is $y(t_0) = y_0$. We can use this as the starting point of a sequence of numerical approximations (Y_n), by taking $Y_0 = y_0$. However there may be roundoff errors or other inaccuracies which suggest a more general approach, beginning with a value Y_0, not necessarily equal to y_0. Let us now *define* Y_1 by

$$Y_1 = Y_0 + h\,f(t_0, Y_0). \tag{1.5}$$

Then Y_1 is an approximation to $y(t_1)$.

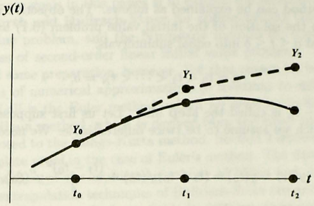
Suppose now that the solution of (0.1) cannot be easily or explicitly found. In fact this is the case with most differential equations. (The function f in (0.1) must be quite simple to allow us to find a symbolic solution of (0.1); even then ingenious methods are sometimes required to find the solution.) Although we do not know $y(t_1)$, we know its approximate value Y_1. Furthermore, if we put

$$Y_2 = Y_1 + h\,f(t_1, Y_1),$$

then we can expect Y_2 to be a reasonable approximation of $y(t_2)$. More generally, we define

$$Y_{n+1} = Y_n + h\,f(t_n, Y_n) \tag{1.6}$$

for $0 \leq n \leq N - 1$. The **Euler method** or **tangent method** consists of approximating the solution to (0.1) by means of (1.6), which we call the **Euler method** formula



The Euler method approximation

The above picture shows the geometry of the Euler method. From (0.1) we know the point (t_0, Y_0) and the slope of the tangent line to the solution curve at (t_0, Y_0), namely $f(t_0, Y_0)$. Thus the tangent line is the graph of

$$t \longmapsto Y_0 + (t - t_0) f(t_0, Y_0).$$

We can obtain an approximation Y_1 to $y(t_1)$ by moving along this tangent line until t reaches t_1; then $Y_0 + (t_1 - t_0) f(t_0, Y_0) = Y_1$, say. Once Y_1 is determined, we can compute $f(t_1, Y_1)$, which is an approximation to $f(t_1, y(t_1))$, which in turn approximates the slope of the tangent line to the actual solution at t_1. Continuing in this way, we obtain a broken line which approximates the solution curve.

To see how the Euler Method works, we use it to approximate the solution to a simple first order linear equation.

Example 1.1. *Use the Euler method to find an approximate solution to the initial value problem*

$$\begin{cases} y' = y + 1, \\ y(0) = 0, \end{cases} \tag{1.7}$$

for $0 \le t \le 1$. *Use step size* $h = 0.1$ *and compare the approximation with the exact solution.*

Solution. The Euler method formula (1.6) for the initial value problem (1.7) with step size $h = 0.1$ becomes

$$Y_{n+1} = Y_n + 0.1(Y_n + 1).$$

for $0 \le n \le N - 1$. The interval $0 \le t \le 1$ is divided into 10 equal pieces, so $N = 10$. We are given $Y_0 = 0$; then $Y_1 = Y_0 + 0.1(Y_0 + 1) = 0.1$ and

$$Y_2 = Y_1 + 0.1(Y_1 + 1) = 0.1 + 0.1(1.1) = 0.21.$$

Continuing in this way, we get the first three columns in the following table:

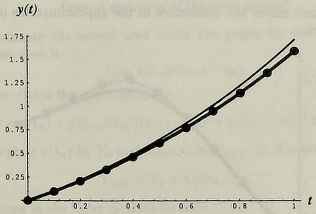
n	t_n	Y_n	$y_{\text{exact}}(t_n)$
0	0.0	0.0	0.0
1	0.1	0.1	0.10517
2	0.2	0.21	0.22140
3	0.3	0.331	0.34986
4	0.4	0.4641	0.49182
5	0.5	0.61050	0.64872
6	0.6	0.77156	0.82212
7	0.7	0.94871	1.01375
8	0.8	1.14359	1.22554
9	0.9	1.35795	1.45960
10	1.0	1.59370	1.71828

Since $y' = y + 1$ is a first order linear differential equation, the exact solution of (1.7) is easily found to be

$$y_{\text{exact}}(t) = e^t - 1.$$

Then $y_{\text{exact}}(0.1) = e^{0.1-1} \approx 0.10517$, $y_{\text{exact}}(0.2) = e^{0.2-1} \approx 0.22140$, and so forth. Values for the exact solution are shown in the fourth column.

In the following plot we compare the exact solution of (1.7) with the approximate solution obtained by the Euler method.



The exact solution of $y' = y + 1$, $y(0) = 0$ and an approximate solution found by the Euler method

The difference $|Y_{10} - y(t_{10})|$ is approximately 0.12. an error of less than ten percent, which can be considered reasonable. We shall see that more sophisticated methods such as the Heun method and the Runge-Kutta method give much better results ■

Next let us consider a differential equation for which it is impossible to find the solution, at least by elementary techniques.

Example 1.2. *Use the Euler method to find an approximate solution to the initial value problem*

$$\begin{cases} y' = 5 - t^2y^3, \\ y(0) = 0, \end{cases} \tag{1.8}$$

for $0 \le t \le 1$. *Use step sizes* $h = 0.1$ *and* $h = 0.01$ *and compare the results.*

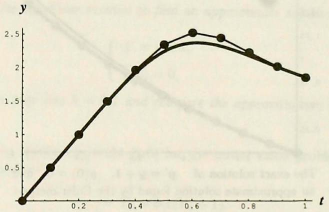
Solution. The Euler method formula (1.6) for the initial value problem (1.8) with step size h is

$$Y_{n+1} = Y_n + h(5 - t_n^2 Y_n^3). \tag{1.9}$$

Taking $h = 0.01$ and then $h = 0.001$ in (1.9) gives us the following table:

t	Y_n $(h = 0.1)$	Y_n $(h = 0.01)$
0.0	0.0	0.0
0.1	0.50000	0.49998
0.2	0.99987	0.99886
0.3	1.49588	1.48647
0.4	1.96575	1.92540
0.5	2.34422	2.24190
0.6	2.52216	2.36963
0.7	2.44457	2.32246
0.8	2.22875	2.18041
0.9	2.02021	2.01484
1.0	1.85237	1.86018

Here is a plot which shows the differences in the approximations obtained with the different step sizes:



Approximate solutions to $y' = 5 - t^2y^3$, $y(0) = 0$
found by the Euler method with step sizes 0.1 and 0.01

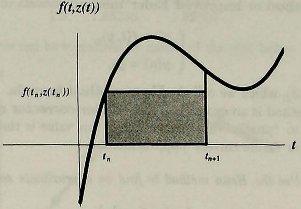
Finally, we give an alternate derivation of the Euler method formula (1.6). Suppose that we have an exact solution $y = z(t)$ of the initial value problem (0.1). Then if we integrate $z'(t)$ from t_n to t_{n+1} we obtain

$$\int_{t_n}^{t_{n+1}} z'(t)dt = \int_{t_n}^{t_{n+1}} f(t, z(t))dt,$$

which can be rewritten as

$$z(t_{n+1}) = z(t_n) + \int_{t_n}^{t_{n+1}} f(t, z(t))dt. \tag{1.10}$$

Consider the graph of the function $t \longmapsto f(t, z(t))$. The integral on the right hand side of (1.10) equals the area under this graph from t_n to t_{n+1}, as indicated by the following picture:



Integral derivation of the Euler method

We can approximate the actual area under the graph by the area of the shaded rectangle, whose area is

$$f(t_n, z(t_n))(t_{n+1} - t_n).$$

From (1.10) we obtain the approximation

$$z(t_{n+1}) \approx z(t_n) + f(t_n, z(t_n))(t_{n+1} - t_n) = z(t_n) + f(t_n, z(t_n))h. \tag{1.11}$$

In (1.11) we replace $z(t_n)$ by Y_n and $z(t_{n+1})$ by Y_{n+1}. In this way we obtain

$$Y_{n+1} = Y_n + h\, f(t_n, Y_n),$$

which is the same as the Euler method formula (1.6).

2 The Heun Method

As a first attempt to obtain an improvement of the Euler method formula (1.6), let us replace $f(t_n, Y_n)$ with the average of $f(t_n, Y_n)$ and $f(t_{n+1}, Y_{n+1})$; this leads to the formula

$$Y_{n+1} = Y_n + \frac{h}{2}\big(f(t_n, Y_n) + f(t_{n+1}, Y_{n+1})\big). \tag{2.12}$$

Unfortunately, Y_{n+1} occurs on both sides of (2.12), so that we cannot obtain it without solving the equation for Y_{n+1}; this may be difficult. Fortunately, we already have an approximation for Y_{n+1}, namely the value

$$Y_n + h\,f(t_n, Y_n)$$

from the Euler method formula (1.6). Let us substitute $Y_n + h\,f(t_n, Y_n)$ for the Y_{n+1} occurring on the right hand side of (2.12). The result is

$$Y_{n+1} = Y_n + \frac{h\Big(f(t_n, Y_n) + f\big(t_{n+1}, Y_n + h\,f(t_n, Y_n)\big)\Big)}{2}. \tag{2.13}$$

The **Heun**[2] **method** or **improved Euler method** consists of approximating the solution to

$$\begin{cases} y' = f(t, y), \\ y(a) = y_0, \end{cases} \tag{2.14}$$

by means of (2.13), which we call the **Heun method formula**.

The Heun method is an example of a **predictor-corrector method**. The Euler method is used to "predict" a value for Y_{n+1}; this value is then used in (2.13) to obtain a better (or "more correct") approximation.

Example 2.1. *Use the Heun method to find an approximate solution to the initial value problem*

$$\begin{cases} y' = 5 - t^2 y^3, \\ y(0) = 0, \end{cases} \tag{2.15}$$

for $0 \le t \le 1$. Use step size $h = 0.1$. Compare the Heun method approximation with the Euler method approximation.

Solution. Let $f(t, Y) = 5 - t^2 Y^3$. Since the Euler method approximation to Y_{n+1} is $Y_n + h\,f(t_n, Y_n) = Y_n + h(5 - t_n^2 Y_n^3)$, we compute

$$\begin{aligned} f\big(t_{n+1}, Y_n + h\,f(t_n, Y_n)\big) &= f\big(t_n + h, Y_n + h(5 - t_n^2 Y_n^3)\big) \\ &= 5 - (t_n + h)^2 \big(Y_n + h(5 - t_n^2 Y_n^3)\big)^3. \end{aligned}$$

We also have $f(t_n, Y_n) = 5 - t_n^2 Y_n^3$. Thus the Heun method formula (2.13) becomes

$$Y_{n+1} = Y_n + \frac{h}{2}\Big(10 - t_n^2 Y_n^3 - (t_n + h)^2 \big(Y_n + h(5 - t_n^2 Y_n^3)\big)^3\Big) \tag{2.16}$$

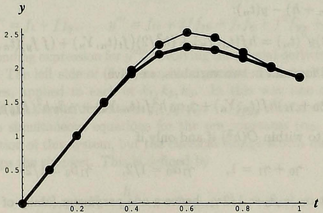
We obtain $Y_1 = (h/2)\big(10 - h^5(125)\big)$, and so forth. Taking $h = 0.1$ in (2.16) gives us the last column in the following table. The middle column is computed using the

[2]Karl Heun (1859-1929). German mathematician.

Euler method formula (1.6).

t	Y_n (Euler)	Y_n (Heun)
0.0	0.0	0.0
0.1	0.50000	0.49994
0.2	0.99987	0.99788
0.3	1.49588	1.48089
0.4	1.96575	1.90680
0.5	2.34422	2.20007
0.6	2.52216	2.30745
0.7	2.44457	2.26215
0.8	2.22875	2.14016
0.9	2.02021	1.99622
1.0	1.85236	1.85650

The data in this table can be visualized by means of the plot below.



Comparison of approximate solutions to $y' = 5 - t^2y^3 \quad y(0) = 0$,
found by the Euler and Heun methods with step size **0**.1

3 The Runge-Kutta Method

The Runge[3]-Kutta[4] method selected for this paper is an improvement of both the Euler method formula (1.6) and the Heun method formula (2.13) that involves a weighted average of four values of $f(t, y)$ taken at different points in the interval $t_n \le t \le t_{n+1}$.

In order to motivate this, we first reformulate the Euler and Heun methods in a common format which can be generalized. The notations $O(h^2)$, resp. $O(h^3)$, indicate terms which are proportional to h^2, resp. h^3, plus higher powers of h.

[3]Carl David Tolmé Runge (1856-1927). Applied Mathematics Professor at Göttingen. Runge devised his numerical method about 1895.

[4]Martin Wilhelm Kutta (1867-1944). German applied mathematician who made important contributions to aerodynamics. Kutta extended Runge's method in 1901.

The Euler method can be reformulated as the problem of writing

$$Y_{n+1} - Y_n = k_0$$

where k_0 is chosen so that so that $k_0 = hy'(t_n) + O(h^2)$, the first-order Taylor approximation of $y(t_n + h) - y(t_n)$. The simplest choice is $k_0 = hf(t_n, Y_n)$ as defined in the Euler method.

The Heun method can be reformulated as the problem of writing

$$Y_{n+1} - Y_n = \gamma_0 k_0 + \gamma_1 k_1$$

where $k_0 = h f(t_n, Y_n)$, $k_1 = h f(t_n + \alpha_0 h, Y_n + \beta_0 k_0)$ and where the constants $\alpha_0, \beta_0, \gamma_0, \gamma_1$ are chosen so that $\gamma_0 k_0 + \gamma_1 k_1 = hy'(t_n) + (h^2/2)y''(t_n) + O(h^3)$, the second-order Taylor approximation of $y(t_n + h) - y(t_n)$.

Example 3.1. *Find all possible choices of the constants* $\alpha_0, \beta_0, \gamma_0, \gamma_1$.

Solution Using the subscript notation for partial derivatives, we write the Taylor expansion of $y(t_n + h) - y(t_n)$:

$$hy'(t_n) + (h^2/2)y''(t_n) = hf(t_n, Y_n) + (h^2/2)(f_t(t_n, Y_n) + (f f_y)(t_n, Y_n)) + O(h^3).$$

Using the Taylor formula in two variables, we have

$$\gamma_0 k_0 + \gamma_1 k_1 = (\gamma_0 + \gamma_1)hf(t_n, Y_n) + \gamma_1\alpha_0 h^2 f_t(t_n, Y_n) + \gamma_1\beta_0 h^2(f f_y)(t_n, Y_n) + O(h^3).$$

These will agree to within $O(h^3)$ if and only if

$$\gamma_0 + \gamma_1 = 1, \qquad \gamma_1\alpha_0 = 1/2, \qquad \gamma_1\beta_0 = 1/2$$

This implies that $\alpha_0 = \beta_0 = 1/2\gamma_1$, hence a one-parameter family of solutions which can be written

$$\gamma_0 = 1 - \gamma_1, \alpha_0 = \beta_0 = 1/2\gamma_1 \qquad \blacksquare$$

In particular this is satisfied by the symmetric choice $\gamma_0 = \gamma_1 = 1/2, \alpha_0 = 1, \beta_0 = 1$ leading to the Heun method

$$Y_{n+1} - Y_n = (h/2)f(t_n, Y_n) + (h/2)f(t_n + h, Y_n + hf(t_n, Y_n)).$$

Another choice is $\gamma_0 = 0, \gamma_1 = 1, \alpha_0 = 1/2, \beta_0 = 1/2$ which leads to the second-order difference scheme

$$Y_{n+1} - Y_n = hf\big(t_n + (h/2), Y_n + (h/2)f(t_n, Y_n)\big).$$

Similarly one can define difference schemes based on the third-order or fourth-order Taylor approximations, and so forth. The Runge-Kutta scheme of order four is formulated as the problem of writing

$$Y_{n+1} - Y_n = \gamma_0 k_0 + \gamma_1 k_1 + \gamma_2 k_2 + \gamma_3 k_3,$$

where

$$\begin{aligned}
k_0 &= h\,f(t_n, Y_n)\\
k_1 &= h\,f(t_n + \alpha_0 h, Y_n + \beta_0 k_0)\\
k_2 &= h\,f(t_n + \alpha_1 h, Y_n + \beta_1 k_1)\\
k_3 &= hf(t_n + \alpha_2 h, Y_n + \beta_2 k_2)
\end{aligned}$$

so that we have

$$\gamma_0\,k_0 + \gamma_1\,k_1 + \gamma_2\,k_2 + \gamma_3\,k_3 = hy'(t_n) + \frac{h^2}{2}y''(t_n) + \frac{h^3}{6}y'''(t_n) + \frac{h^4}{24}y''''(t_n) + O(h^5). \tag{3.17}$$

In order to find the ten constants $(\alpha_0, \ldots, \gamma_3)$ we match the corresponding terms in the Taylor expansions of both sides of (3.17). Using the subscript notation for partial derivatives, the right side can be expanded by using the chain rule applied to the differential equation $y' = f(t, y)$. Thus

$$y'' = f_t + f f_y, \qquad y''' = f_{tt} + 2f f_{ty} + f_t f_y + f^2 f_{yy} + f f_y^2$$

with a corresponding expression for y'''', involving the partial derivatives of orders one, two and three. The left side of (3.17) can be expanded, by using the Taylor formula in two variables, applied to each of k_1, k_2, k_3. In this way one obtains a formula involving f and the partial derivatives $f_t, f_y, f_{tt}, \ldots, f_{yyy}$ a total of ten terms; (3.17) then yields ten simultaneous equations for the ten constants $(\alpha_0, \ldots, \gamma_3)$. There is no unique solution of this system, but the classical Runge-Kutta method chooses one which is perhaps the simplest. This is defined by

$$Y_{n+1} = Y_n + \frac{h}{6}\big(a_{1n} + 2a_{2n} + 2a_{3n} + a_{4n}\big), \tag{3.18}$$

where

$$\begin{aligned}
a_{1n} &= f(t_n, Y_n),\\
a_{2n} &= f\left(t_n + \frac{h}{2}, Y_n + \frac{h}{2}a_{1n}\right),\\
a_{3n} &= f\left(t_n + \frac{h}{2}, Y_n + \frac{h}{2}a_{2n}\right),\\
a_{4n} &= f(t_n + h, Y_n + ha_{3n}).
\end{aligned}$$

We call (3.18) the **Runge-Kutta method formula**.. It may be shown that in case $f(t, y)$ depends only on t, then the Runge-Kutta formula (3.18) reduces to the familiar Simpson's rule for approximating an integral by the integral of a quadratic polynomial. Therefore we can assert that (3.18) is a generalization of Simpson's rule.

The **Runge-Kutta method** is significantly more complicated than the Euler and Heun methods and is considerably more accurate. Computers can easily handle this increased complexity.

Mathematica contains a package which will produce the systems of equations for any Runge-Kutta scheme. It is called "Butcher.m" and it can be found in the "NumericalMath" directory . For example, to produce the system of 10 equations referenced above (see [Butcher]), evaluate

```
<<NumericalMath`Butcher`
RungeKuttaOrderConditions[4,4]
```

Example 3.2. *Use the Runge-Kutta method to find an approximate solution to the initial value problem*

$$\begin{cases} y' = 10(1 - t^2y^3 + 2t^4y^3), \\ y(0) = 0, \end{cases} \tag{3.19}$$

for $0 \le t \le 1$. *Use step size* $h = 0.1$. *Compare the Runge-Kutta method approximation with the Euler and Heun method approximations.*

Solution. Let $f(t, y) = 10(1 - t^2y^3 + 2t^4y^3)$. We compute

$$\begin{aligned} a_{10} &= f(0,0) = 10, \\ a_{20} &= f(0.05, 0.05 \times 10) = 9.99688, \\ a_{30} &= f(0.05, 0.05 \times 9.99688) = 9.99688, \\ a_{40} &= f(0.1, 0.1 \times 9.99688) = 9.90109. \end{aligned}$$

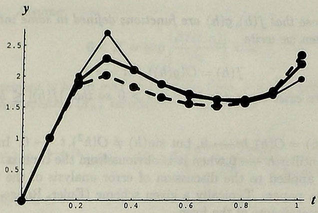
Hence

$$Y_1 = 0.0166667(a_{10} + 2a_{20} + 2a_{30} + a_{40}) = 0.998144.$$

Continuing in this way we get the last column in the following table. Similar calculations using the Euler and Heun method formulas yield the second and third columns.

n	t_n	Y_n (Euler)	Y_n (Heun)	Y_n (Runge-Kutta)
0	0.0	0.0	0.0	0.0
1	0.1	1.0	0.99505	0.99814
2	0.2	1.99010	1.83994	1.90323
3	0.3	2.68744	1.99998	2.27305
4	0.4	2.09780	1.80606	2.08513
5	0.5	1.85703	1.64398	1.85736
6	0.6	1.65626	1.54332	1.70104
7	0.7	1.60945	1.50988	1.61893
8	0.8	1.56762	1.56258	1.61459
9	0.9	1.68005	1.75588	1.72850
10	1.0	1.95025	2.33931	2.19565

Visualization of the data in this table is provided by the plot below.



Approximating solutions to
$y' = 10(1 - t^2y^3 + 2t^4y^3, \quad y(0) = 0$
Thin=Euler, Dashed=Heun, Thick=Runge-Kutta

4 Numerical Errors and Stability

Numerical Errors

A common way to classify methods is to give their order of accuracy. This order is associated with truncation error as defined by the particular method and the Taylor expansion of the solution $y(t)$. Taylor's theorem states that if $y(t)$ has $k+1$ continuous derivatives on the interval $t_0 - \delta < t < t_0 + \delta$, then

$$y(t) = y(t_0) + y'(t_0)(t - t_0) + \frac{y''(t_0)}{2!}(t - t_0)^2 + \cdots + \frac{y^{(k)}(t_0)}{k!}(t - t_0)^k + \boldsymbol{TE},$$

where $t_0 - \delta < t < t_0 + \delta$. Here the Taylor remainder $\boldsymbol{TE}$ is called the **local truncation error**; it is defined by

$$\boldsymbol{TE} = \frac{y^{(k+1)}(\eta)}{(k+1)!}(t - t_0)^{k+1},$$

where $t_0 - \delta \le \eta \le t_0 + \delta$. If $t_1 = t_0 + h$, then we may write

$$\boldsymbol{TE} = \frac{y^{(k+1)}(\eta)}{(k+1)!}h^{k+1},$$

and we say that the local truncation error is proportional to h^{k+1}. When this occurs, we say that the method is of order k. The reason for defining it this way is because the **global truncation error**

$$\boldsymbol{GTE} = |y(t_i) - Y(t_i)|$$

is asymptotically proportional to one lower power of h when h tends to zero. Here we use y for the true solution and Y for the approximate solution. In order to discuss the error analysis, we introduce the **big-O notation**.

Definition. *Suppose that $f(h), g(h)$ are functions defined in some interval $0 < h < a$ with $g(h) > 0$. Then we write*

$$f(h) = O(g(h)), \quad t \longrightarrow 0$$

provided that there exist constants $M > 0, \delta > 0$ so that $|f(h)| \leq Mg(h)$ whenever $0 < h < \delta$.

For example $\sin(h) = O(h), h \longrightarrow 0$, but $\sin(h) \neq O(h^2), t \longrightarrow 0$. In many cases we will omit the quantifier $h \longrightarrow 0$ when it is obvious from the context.

This may be applied to the discussion of error analysis of the various schemes introduced in this paper. Typically a given scheme (Euler, Runge-Kutta, etc) will satisfy a pair of statements of the form

$$\boldsymbol{TE} = O(h^{k+1}) \quad \text{and} \quad \boldsymbol{GTE} = O(h^k) \quad h \longrightarrow 0$$

for a certain value of k.

To see how this analysis is done, let us consider the Euler method. Using Taylor's theorem,

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(\eta) \tag{4.20}$$

for some $\eta \in [t_n, t_{n+1}]$. To analyze the error in the Euler method, we state the Euler method in terms of the approximate solution Y by

$$Y_{n+1} = Y_n + hf(t_n, Y_n) \quad (n = 0, 1, \ldots, N-1)$$

and subtract this equation from equation (4.20) to obtain

$$y(t_{n+1}) - Y_{n+1} = y(t_n) - Y_n + h\big(f(t_n, y(t_n)) - f(t_n, Y_n)\big) + \frac{h^2}{2}y''(\eta). \tag{4.21}$$

The error in Y_{n+1} consists of two parts: (1) the local truncation error $\boldsymbol{TE}$ introduced at step t_{n+1} and (2) the **propagated error**

$$y(t_n) - Y_n + h\big(f(t_n, y(t_n)) - f(t_n, Y_n)\big).$$

The propagated error can be simplified by applying the mean value theorem to $f(t, z)$, considered as a function of z:

$$f(t_n, y(t_n)) - f(t_n, Y_n) = \frac{\partial f(t_n, \xi)}{\partial z}\big(y(t_n) - Y_n\big).$$

where ξ is between $y(t_n)$ and Y_n. We let $\epsilon_n = y(t_n) - Y_n$, and use the above to rewrite equation (4.21), obtaining

$$\epsilon_{n+1} = \left(1 + h\frac{\partial f(t_n, \xi)}{\partial z}\right)\epsilon_n + \frac{h^2 y''(\eta)}{2}. \tag{4.22}$$

These computations yield a general error analysis for the Euler method for the initial value problem as stated in the following theorem.

Theorem 4.1. *Assume*

$$0 < K = \sup \left| \frac{\partial f(t,z)}{\partial z} \right| < \infty. \qquad (4.23)$$

where the supremum is taken over (t, z) *of the form* $z = y(t)$ *with* $t_0 \le t \le b$. *Then the Euler method solution* (Y_n) *satisfies the error bound*

$$|y(t_n) - Y_n| \le e^{(b-t_0)K}|y(t_0) - Y_0| + h\left(\frac{e^{(b-t_0)K} - 1}{2K}\right) \sup_{t_0 \le t \le b} |y''(t)|, \qquad (4.24)$$

for all n *with* $t_0 \le t_n \le b$.

Proof. The proof can be accomplished by mathematical induction on the sequence of numbers $\epsilon_n, n = 0, 1 \ldots, N$. From equation (4.22) we have

$$|\epsilon_{n+1}| \le A\,|\epsilon_n| + B \qquad (n = 0, 1, \ldots, N-1), \qquad (4.25)$$

where $A = 1 + hK, B = h^2M/2$ and $M = \sup_{t_0 \le t \le b} |y''(t)|$. We propose to show that

$$|\epsilon_n| \le |\epsilon_0|\,A^n + \frac{B}{A-1}(A^n - 1) \qquad (n = 0, 1, \ldots, N). \qquad (4.26)$$

Clearly (4.26) holds for the value $n = 0$. Assuming its truth for the value $n = m$, we have

$$\begin{aligned} |\epsilon_{m+1}| &\le A|\epsilon_m| + B \\ &\le A\left(|\epsilon_0|A^m + \frac{B}{A-1}(A^m - 1)\right) + B \\ &= |\epsilon_0|A^{m+1} + \frac{B}{A-1}(A^{m+1} - 1), \end{aligned}$$

which proves (4.26) for the value $n = m + 1$, and hence for all n by mathematical induction. To obtain the conclusion (4.23) we apply this with $A = 1+hK, B = h^2M/2$ and taking note of the inequality $1 + x \le e^x$ to write

$$\begin{aligned} |\epsilon_n| &\le |\epsilon_0|(1 + hK)^n + \frac{h^2M/2}{hK}((1 + hK)^n - 1) \\ &\le |\epsilon_0|e^{hnK} + \frac{hM}{2K}(e^{nhK} - 1) \\ &\le |\epsilon_0|e^{(b-t_0)K} + \frac{hM}{2K}(e^{(b-t_0)K} - 1). \end{aligned}$$

Making the identification $\epsilon_n = y(t_n) - Y_n$ completes the proof ■

When $Y_0 = y(t_0)$ (as is commonly the case), (4.24) can be written

$$|y(t_n) - Y_n| \le c\,h, \qquad t_0 \le t_n \le b,$$

where c is a constant. Therefore we say that the Euler method is an order one or first order method. When h is halved, the error is halved. Also, the Euler method is said to converge with order 1. In general, if we have

$$|y(t_n) - Y_n| \leq c\,h^k, \qquad t_0 \leq t_n \leq b,$$

then we say that the method is an order k method or is convergent with order k. To see what this means, let us consider an example.

Example 4.1. *Illustrate the error bound (4.23) for the equation*

$$y' = 4t, \qquad y(0) = 0$$

whose exact solution is $y(t) = 2t^2$.

Solution. The error formula (4.22) becomes

$$\epsilon_{n+1} = \epsilon_n + h^2, \qquad \epsilon_0 = 0.$$

Using induction, we get

$$\epsilon_n = n\,h^2, \qquad n \geq 0.$$

Since $n\,h = t_n$,

$$\epsilon_n = t_n h$$

■

In the above example we see that for each t_n, the error of approximation in the Euler method at t_n is proportional to h. The local truncation error TE is proportional to h^2, but the cumulative effect of these errors is a total error proportional to h. The following table summarizes the orders of the methods in `ODE` for the other approximation methods which occur in this paper. The methods of Milne, Adams-Bashforth and Bulirsch-Stoer in addition to an implicit Runge-Kutta method will be discussed in later sections. In `ODE`, `RungeKutta4` is the classical Runge-Kutta method defined in Section (3) and `RungeKutta45` is the Runge-Kutta-Fehlberg method.

Method	Order
`Euler`	1
`SecondOrderEuler`	2
`Heun`	2
`ImplicitRungeKutta`	3
`RungeKutta4`	4
`RungeKutta45`	4(5)
`Milne`	4
`AdamsBashforth`	4
`BulirschStoer`	variable
`NDSolve`	unknown

Example 4.2. *Consider the initial value problem*

$$\begin{cases} y' = t^2 y, \\ y(0) = 1 \end{cases}$$

Find the exact solution at $t = 1$. *Then use the Euler method with*

$$h \in \{1/8, 1/16, 1/32, 1/64, 1/128, 1/256, 1/512\}.$$

Create a table showing the absolute errors corresponding to the various step sizes.

Solution. First we compute the exact solution at 1.0 and call it **exact**.

```
exact = ODE[{y'==t^2 y,y[0]==1},y,t,
Form->Explicit,Method->Separable] /. t -> 1.0
```

Then we use

```
TableForm[Table[{1/2^k,diff = exact-Last[Last[
ODE[{y'==t^2 y,y[0]==1},y,{t,0,1},Method->Euler,
NumericalOutput->True,StepSize->1/2^k]]],
diff/(1/2^k)},{k,3,9}],TableHeadings->
{None,{"Stepsize","Error","Error/Stepsize"}}]
```

and obtain

```
Stepsize    Error         Error/Stepsize
1
-
8           0.0922455     0.737964

1
--
16          0.0490388     0.784622

1
--
32          0.0253175     0.810159

1
--
64          0.0128679     0.823544

 1
---
128         0.0064875     0.8304

 1
---
256         0.0032573     0.833869

 1
---
512         0.00163206    0.835615
```

Note that the last column is approximately a constant, which is what the theory predicts for this method.

Stability

The result of the previous subsection may be paraphrased in the language of stability. A numerical method is said to be **stable** if for any first-order differential equation $y' = f(t, y)$ and for any $\epsilon > 0$, there exists a $\delta > 0$ so that the sequence of numerical approximations Y_n satisfy $|Y_n - y(t_n)| < \epsilon$ for $n = 1, 2, \ldots, N$ whenever the step size $h < \delta$. To illustrate this, if a numerical method satisfies an inequality of the form $|Y_n - y(t_n)| < c\,h^k$ then we may take $\delta = (\epsilon/c)^{1/k}$ in the definition of stability. For example in the Euler method we have $k = 1$, thus $\delta = \epsilon/c$

This notion of stability is qualitative, and does not provide a numerical criterion to differentiate between the various numerical methods. Later in section 11 we will develop the notion of **absolute stability** in order to study stability in more detail.

5 Multistep Methods

The Euler, Heun and Runge-Kutta methods belong to the class of *single step* numerical integration methods. This means that only the information at step n is used to compute the estimate at step $n + 1$. There are other methods which use two or more previous data points to compute the estimate at the next data point. These are called **multistep methods** and they are frequently used to introduce the class of **predictor-corrector methods**. **ODE** includes two of the most familiar multistep methods, the Milne[5] and the Adams[6]-Bashforth/Adams-Moulton methods, where the latter is simply called the Adams-Bashforth method. Here we discuss the details of both the Milne method and the Adams-Bashforth method. Like the Runge-Kutta schemes, there exists a complete sequence of multistep methods depending on the number of points used to generate the interpolating polynomials. In this paper, we have chosen the fourth-order versions of the Milne and Adams-Bashforth methods so that comparisons can be made with other methods. Others can be derived in a similar fashion.

As in the previous sections, the problem is to find a numerical approximation to the solution of the initial-value problem

$$y' = f(t, y), \qquad y(t_0) = Y_0. \tag{5.27}$$

The Milne Method

The fourth-order Milne method needs four consecutive uniformly spaced data points to estimate the next point on the solution graph. We begin by computing the approximate values of the solution at equally spaced points t_1, t_2, t_3, for example by the Runge-Kutta method. The Milne method can then be used to find an approximate

[5]Edward Arthur Milne (1896-1950). English Astronomer. Studied the atmospheres of stars. Developed a new form of relativity called kinematic relativity, an alternative to Einstein's general theory of relativity.

[6]John Couch Adams (1819-1892). English Astronomer. Discover of the planet Neptune.

value for $y(t_4)$ as follows. In order to describe the Milne method for computing Y_{n+1} in general, we assume given the points

$$\{(t_{n-3}, y_{n-3}), (t_{n-2}, y_{n-2}), (t_{n-1}, y_{n-1}), (t_n, y_n)\},$$

where $t_{n-k} = t_{n-k-1} + h$.

Now if we had an exact solution of (5.27), we could write

$$y(t_{n+1}) = y(t_{n-3}) + \int_{t_{n-3}}^{t_{n+1}} y'(s)\, ds \tag{5.28}$$

Since $y(t_{n-2}), y(t_{n-1}), y(t_n)$ are known (approximately) , we can compute the derivatives y'_{n-2}, y'_{n-1}, y'_n (approximately) from (5.27). Now we approximate the function $t \longrightarrow y'(t)$ by a quadratic polynomial passing through the three points (t_{n-2}, y'_{n-2}), $(t_{n-1}, y'_{n-1}), (t_n, y'_n)$. This can be done in a unique way. We substitute this polynomial into the integral in (5.28) and substitute the approximate value ofy_{n-3} into (5.28) to obtain an approximation for $y(t_{n+1})$.

Instead of giving a symbolic derivation of the necessary formula (5.29) below, we will use *Mathematica* to obtain this. (The letter A below denotes the approximate value of the integral in (5.28) when we use the quadratic polynomial approximation).

Mathematica allows us to perform this derivation using the following commands, where we set *ypn* $= y'_n$, and *ypnmk* $= y'_{n-k}$.

First we define the system of equations which leads to the quadratic passing through the last three points.

```
system1 = {a1 (tn - 2h)^2 + b1 (tn - 2h) + c1 == ypnm2,
           a1 (tn -  h)^2 + b1 (tn -  h) + c1 == ypnm1,
           a1  tn^2       + b1 tn        + c1 == ypn}
coeffs1 = Flatten[Solve[system1,{a1,b1,c1}]]
parabola1 = a1 t^2 + b1 t + c1 /. coeffs1 // Simplify
```

(We suppress the output.) Next we integrate this quadratic on the interval $t_n - 3h \leq t \leq t_n + h$:

$$A = \int_{t_{n-3}}^{t_{n+1}} (a_1 t^2 + b_1 t + c_1)\, dt = \frac{4\,h}{3}(2y'_{n-2} - y'_{n-1} + 2y'_n).$$

```
A = Integrate[parabola1,{t,tn - 3h,tn + h}] // Simplify

4 h (2 ypn - ypnm1 + 2 ypnm2)
-----------------------------
             3
```

Therefore, the predicted value is

$$y_{n+1,p} = y_{n-3} + A = y_{n-3} + \frac{4\,h}{3}(2y'_{n-2} - y'_{n-1} + 2y'_n). \tag{5.29}$$

Next we insert this value into the differential equation (5.27) to produce a better approximation to the derivative at t_{n+1}.

$$y'_{n+1} = f(t_{n+1}, y_{n+1,p}).$$

Finally, we fit another quadratic to the points $(t_{n-1}, y'_{n-1}), (t_n, y'_n), (t_{n+1}, y'_{n+1})$, integrate to get B and then add B to y_{n-1} to give y_{n+1} (corrected). Again, *Mathematica* allows us to perform this derivation using the following commands.

```
system2 = {a2 (tn - h)^2 + b2 (tn - h) + c2 == ypnm1,
           a2  tn^2      + b2 tn       + c2 == ypn,
           a2 (tn + h)^2 + b2 (tn + h) + c2 == ypnp1}
coeffs2 = Flatten[Solve[system2,{a2,b2,c2}]]
parabola2 = a2 t^2 + b2 t + c2 /. coeffs2 // Simplify
```

Then we integrate it on the interval $[t_n - h, t_n + h]$.

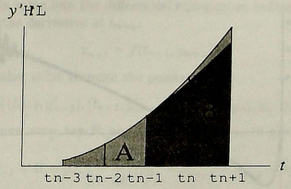
$$B = \int_{t_{n-1}}^{t_{n+1}} (a_2 t^2 + b_2 t + c_2)\, dt = \frac{h}{3}(y'_{n-1} + 4\, y'_n + y'_{n+1}).$$

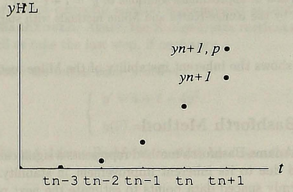
```
B = Integrate[parabola2,{t,tn - h,tn + h}] // Simplify

h (4 ypn + ypnm1 + ypnp1)
-------------------------
            3
```

Therefore, the **corrected value** is

$$y_{n+1} = y_{n-1} + B = y_{n-1} + \frac{h}{3}(y'_{n-1} + 4\, y'_n + y'_{n+1}). \tag{5.30}$$

Equations 5.29 and 5.30 constitute the fourth-order Milne predictor-corrector method implemented in **ODE** as **Method->Milne**. Normally, a multistep method requires a different technique to terminate the integration, since the final step size is frequently different from h. Again, the Runge-Kutta method can be used to take the last step.





The graph of $y' = f(t, y)$ on the interval $[t_{n-3}, t_{n+1}]$,
and the approximate solution found by the
Milne method

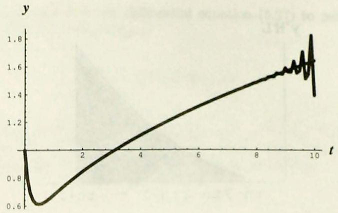
Example 5.1. *Solve the initial value problem*

$$\begin{cases} y' = 1 + t - 4y^2, \\ y(0) = 1 \end{cases}$$

by the Runge-Kutta and Milne methods and simultaneously plot the two approximations.

Solution. We use

```
Show[
{ListPlot[RungeKutta4[1 + t -  4y^2,{0,1},0.1,100][t,y],
PlotJoined->True,DisplayFunction->Identity]},
ListPlot[Milne[1 + t -  4y^2,{0,1},0.1,100][t,y],
PlotJoined->True,DisplayFunction->Identity],
DisplayFunction->$DisplayFunction];
```



Comparison of approximate solutions to $y' = 1 + t - 4y^2, \quad y(0) = 1$ found by the Runge-Kutta and Milne methods with step size 0.1

This example shows the inherent instability of the Milne method.

The Adams-Bashforth Method

The fourth-order Adams-Bashforth method represents a significant enhancement over the Milne method due to its inherent improvement in stability. It also needs four consecutive uniformly spaced data points to estimate the next point on the solution graph but these points are used to perform a different interpolation. Again we assume that the initial value is given by $y(t_0) = Y_0$ and that we obtain three additional points, (for example by the Runge-Kutta method) before the Adams-Bashforth method can be used. So we assume given the four points

$$\{(t_{n-3}, y_{n-3}), (t_{n-2}, y_{n-2}), (t_{n-1}, y_{n-1}), (t_n, y_n)\},$$

where $t_{n-k} = t_{n-k-1} + h$. We begin by writing

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} y'(s)\,ds \tag{5.31}$$

Since $y(t_{n-3}), y(t_{n-2}), y(t_{n-1}), y(t_n)$ are known (approximately), we can compute the derivatives $y'(t_{n-3}), y'(t_{n-2}), y'(t_{n-1}), y'(t_n)$ (approximately) from (5.27). Now we fit a cubic polynomial through the four points

$$\{(t_{n-3}, y'_{n-3}), (t_{n-2}, y'_{n-2}), (t_{n-1}, y'_{n-1}), (t_n, y'_n)\}.$$

This can be done in a unique way. Having obtained this polynomial, we integrate to get an area, say A, and then add A to y_n to give $y_{n+1,p}$ (predicted)

$$y_{n+1,p} = y_n + A = y_n + \frac{h}{24}(55y'_n - 59y'_{n-1} + 37y'_{n-2} - 9y'_{n-3}). \tag{5.32}$$

Next we insert this value into the differential equation as before to produce a better approximation to the derivative at t_{n+1}.

$$y'_{n+1} = f(t_{n+1}, y_{n+1,p}).$$

Finally, we fit another cubic through the points

$$\{(t_{n-2}, y'_{n-2}), (t_{n-1}, y'_{n-1}), (t_n, y'_n), (t_{n+1}, y'_{n+1})\},$$

integrate to get a new area, say B, and then add B to y_n to give y_{n+1} (corrected)

$$y_{n+1} = y_{n-1} + B = y_{n-1} + \frac{h}{24}(19y'_n - 5y'_{n-1} + y'_{n-2} + 9y'_{n-3}). \tag{5.33}$$

Equations 5.32 and 5.33 define the well known fourth-order Adams-Bashforth/Adams-Moulton predictor-corrector method implemented in **ODE** as **Method->AdamsBashforth**. Again, the Runge-Kutta method can be used to start this method as well as take the last step, if necessary.

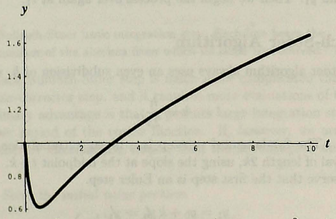
Example 5.2. *Solve the initial value problem*

$$\begin{cases} y' = 1 + t - 4y^2, \\ y(0) = 1 \end{cases}$$

by the Runge-Kutta and Adams-Bashforth methods and simultaneously plot the two approximations.

Solution. We use

```
Show[
{ListPlot[RungeKutta4[1 + t -  4y^2,{0,1},0.1,100][t,y],
PlotJoined->True,DisplayFunction->Identity]},
ListPlot[AdamsBashforth[1 + t -  4y^2,{0,1},0.1,100][t,y],
PlotJoined->True,DisplayFunction->Identity],
DisplayFunction->$DisplayFunction];
```



Comparison of approximate solutions to $y' = 1 + t - 4y^2$, $y(0) = 1$, found by the Runge-Kutta and the Adams-Bashforth methods with step size 0.1

In general, when comparing methods of comparable order, the multistep methods are more efficient than the single step methods, especially when high accuracy is desired. For example, for the fourth-order methods previously discussed, the Adams-Bashforth method requires only two evaluations of the derivative $y' = f(t, y)$ to compute the next solution estimate whereas the Runge-Kutta method needs four evaluations. This assumes that the previous evaluations have been stored for future use. Therefore the Adams-Bashforth method achieves the same accuracy as the Runge-Kutta method in roughly one half the number of function evaluations. Also, y_{n+1} in the predictor-corrector process just described receives its major contribution from the corrector equation. However, there is a strong desire to continue iterating with the corrector equation inside each marching step, thereby completely eliminating the effect of the predictor. Experience shows that very little is gained by doing this and in fact, although the corrected sequence converges, it does not converge to the true solution. Consult any numerical analysis book for additional details concerning multistep methods (see [Acton] or [Ham]).

6 Extrapolation Techniques

In both the Runge-Kutta and predictor-corrector schemes a basic step size h is chosen and used for as long as it gives satisfactory results. It may be adjusted from time to time, but it usually remains constant for long runs. In theory, the solution to the approximate difference equation converges to the true solution as h goes to zero, but h is never allowed to go to zero. Extrapolation is based on computing a limit on h at each step. Simply put, we do a quadrature on y' between t_0 and t_1 (a distance of h) first using a one-step quadrature to get $y_{1,1}$. Then we do this quadrature again using the rule twice, after dividing h into two equal regions, producing a second estimate $y_{1,2}$. Then we compute a third estimate $y_{1,3}$ by using quadrature four times on a step size of $h/4$. Then, when enough estimates are available, we extrapolate to estimate what $y_{1,\infty}$ would have been if h had been permitted to go to zero and we use this value as our final y_1. Then we begin the process over again at t_1.

The Bulirsch-Stoer Algorithm

The Bulirsch-Stoer algorithm always uses an even subdivision of h, which we call k. Therefore

$$k = \frac{h}{2i},$$

for $i = 1, 2 \ldots$. The fundamental substep is a linear extrapolation of y from t to $t + 2k$, an interval of length $2k$, using the slope at the midpoint $t + k$. From the figure on page ??, observe that the first step is an Euler step.

$$y_1 = y_0 + k\,y_0' \Rightarrow y_1'.$$

Next we insert this estimate into the differential equation to get a slope and then use this slope together with the previous value of y to compute a new estimate.

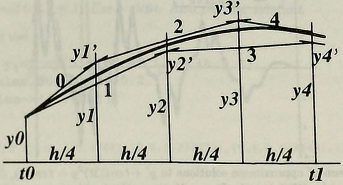
$$y_2 = y_0 + 2k\, y_1' \Rightarrow y_2'.$$

We interleave these steps until we reach the endpoint, where we compute a final ending step y_{4f} and then average the last two estimates to give the final value for y at $t_0 + h$.

$$y_{4f} = y_3 + k\, y_4', \qquad y(t_0 + h) = \frac{y_4 + y_{4f}}{2}.$$

A complete fourth-order Bulirsch-Stoer step ($k = h/4$) is given by

$$\begin{aligned}
&(0) \quad & y_1 &= y_0 + k\, y_0' \Rightarrow y_1',\\
&(1) \quad & y_2 &= y_0 + 2k\, y_1' \Rightarrow y_2',\\
&(2) \quad & y_3 &= y_1 + 2k\, y_2' \Rightarrow y_3',\\
&(3) \quad & y_4 &= y_2 + 2k\, y_3' \Rightarrow y_4',\\
&(4) \quad & y_{4f} &= y_3 + k\, y_4',\\
& & y(t+h) &= \frac{y_4 + y_{4f}}{2}.
\end{aligned}$$



Bulirsch-Stoer basic integration step. Each line bears the number of the abscissa from which its slope was derived.

It is clear that each Bulirsch-Stoer step is much more complicated than any Runge-Kutta or predictor-corrector step, and it requires more evaluations of the differential equation. Its primary advantage is that it permits large integration steps, for example, two steps per period of the cosine function. If, however, we are interested in a moderate number of intermediate values, then the advantage is not so clear (see [StoBul]).

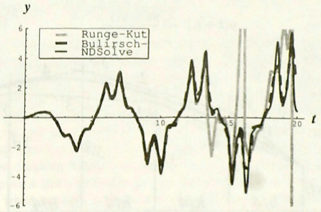
Example 6.1. *Solve the initial value problem*

$$\begin{cases} y' + t\cos(3t)^2 y = t\cos(t), \\ y(0) = 0 \end{cases}$$

over the interval $0 \le t \le 20$ by the Runge-Kutta method ($h = 0.3$), Bulirsch-Stoer method ($h = 0.3$) and NDSolve and simultaneously plot the three approximations.

Solution. We use

```
sol = ODE[{y' + t Cos[3 t]^2 y == t Cos[t], y[0]==0},
y,{t, 0, 20},Method->NDSolve,MaxSteps->2000,
PlotSolution->{{t,0,20},PlotPoints->100}];
Show[{ListPlot[RungeKutta4[t Cos[t] - t Cos[3 t]^2 y,
{0,0},0.3,66][t,y],PlotJoined->True,
PlotStyle->{GrayLevel[.7]},DisplayFunction->Identity],
ListPlot[BulirschStoer[t Cos[t] - t Cos[3 t]^2 y,
{0,0},0.3,66,4][t,y],PlotJoined->True,
PlotStyle->{GrayLevel[.4],AbsoluteDashing[{6,6}]}],
DisplayFunction->Identity],
Plot[Evaluate[y/.sol],{t,0,20},DisplayFunction->Identity]},
DisplayFunction->$DisplayFunction];
```



Comparison of approximate solutions to $y' + t\,cos(3t)^2 y = t\,cos(t)$, $y(0) = 0$, found by NDSolve, the Runge-Kutta method with step size 0.3 and the Bulirsch-Stoer method of order 4 with step size 0.3

This graph reflects the strength of the Bulirsch-Stoer method when the step size is large. We see that the fourth-order Runge-Kutta method eventually diverges from the true solution while the Bulirsch-Stoer method continues to track the solution quite closely.

7 Solving Differential Equations Numerically with ODE

In this section we explain how to use **ODE** to solve first order differential equations numerically (see [GMP] for a complete discussion of **ODE**). Any of the following options can be used:

```
Method -> Euler                    Method -> Heun
Method -> RungeKutta4              Method -> RungeKutta45
Method -> Milne                    Method -> NDSolve
Method -> ImplicitRungeKutta       Method -> AdamsBashforth
Method -> SecondOrderEuler         Method -> BulirschStoer
```

With the exception of **NDSolve** (which is explained in Section 9), all of the above commands follow the same pattern:

```
ODE[{diffeq,initcond},y[t],{t,a,b},
   Method->method,StepSize->size]
```

The default step size is 0.1. Any numerical approximation found with **ODE** can be plotted using the option **PlotSolution**. When plotting, it is frequently useful to suppress the numerical output by using the option
NumericalOutput->None.

Example 7.1. *Use* **ODE** *with the option* **Method->Euler** *to find a numerical approximation to the solution of the initial value problem*

$$\begin{cases} y' + y = -y^3, \\ y(0) = 1 \end{cases}$$

over the interval $0 \le t \le 1$. *Use* 5 *steps. Also plot the solution.*

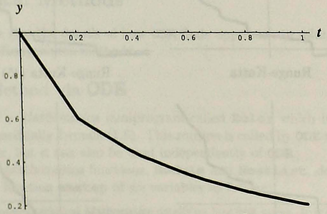
Solution. We use

```
ODE[{y' + y == -y^3,y[0] == 1},y,{t,0,1},
Method->Euler,StepSize->0.2,
PlotSolution->{{t,0,1}}]
```

to obtain

```
{{0,1.},{0.2,0.6},{0.4,0.4368},{0.6,0.332772},{0.8,0.258848},{1.,0.203609}}
```

and the graph



Approximate solution to $y' + y = -y^3, \quad y(0) = 0$
found by the Euler method

In Example 7.1 any of the methods listed on page 537 can be substituted for **Euler**. Furthermore, the option **Method->AllNumerical** can be used to plot all of the approximations, one after another.

Example 7.2. *Use* **ODE** *with the option* **Method->AllNumerical** *to find and plot numerical approximations to the solution of the initial value problem*

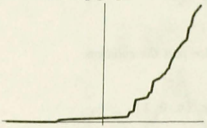
$$\begin{cases} y' - y = e^{5\sin(5y)}, \\ y(0) = 1 \end{cases}$$

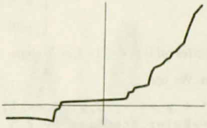
over the interval $-1 \le t \le 1$. *Use step size* 0.02.

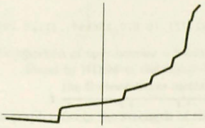
Solution. We use

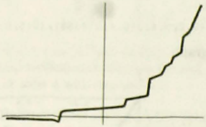
```
ODE[{y' - y  == E^(5Sin[5y]),y[0] == 1},y,{t,-1,1},
Method->AllNumerical,
StepSize->0.02,NumericalOutput->None,
PlotSolution->{{t,-1,1}}]
```

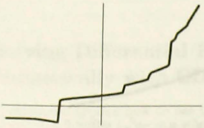
to obtain the following graphs:

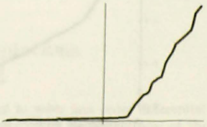

Euler

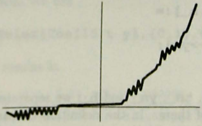

Heun

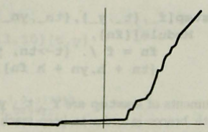

Runge-Kutta

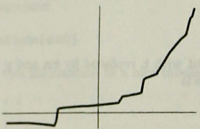

Runge-Kutta (45)

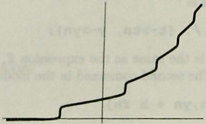

Implicit Runge-Kutta


Second Order Euler


Milne


Adams-Bashforth


Bulirsch-Stoer


NDSolve

From these plots it is clear that some numerical methods are better than others. Inside of *Mathematica* the best method to use is **NDSolve**, as will be explained in Section 9. The Euler and Heun methods are too primitive to be of much practical use. The best all-around method is Runge-Kutta. The Milne and Adams -Bashforth methods are examples of multistep predictor-corrector methods. The Milne method is included only for historical reasons; in practice it gives bad results. The Adams-Bashforth method provides the same accuracy as the Runge-Kutta method at twice the speed.

The reader is invited to experiment with the methods listed on page 537.

8 ODE's Implementation of Numerical Methods[7]

In this section we give the details of how **ODE** finds approximate solutions using the Euler, Heun and RungeKutta methods.

The Euler Method via ODE

We first describe a *Mathematica* miniprogram called **Euler** which implements the Euler method, specifically formula (1.6). This routine is called by **ODE** with the option **Method->Euler**, but it can also be used independently of **ODE**.

We need two *Mathematica* functions, **Module** and **NestList**, described below. We first define a function **emstep** of six variables as follows:

[7] This section contains technical *Mathematica* details on how **ODE** works. It is not needed to use **ODE**. However, the reader is encouraged to explore it because it contains useful information about programming techniques in *Mathematica*.

```
emstep[f_,{t_,y_},{tn_,yn_},h_]:=
    Module[{fn},
        fn = f /. {t->tn, y->yn};
        {tn + h,yn + h fn}]
```

The arguments of **emstep** are **f_**, **t_**, **y_**, **tn_**, **yn_** and **h_**; we separate some of them with braces in order to keep track of them. In the definition of **emstep** we use **Module** with one local variable **fn** and two *Mathematica* commands. The first command is

```
fn = f /. {t->tn, y->yn};
```

Here **fn** is the same as the expression **f**, but with **t** replaced by **tn** and **y** replaced by **yn**. The second command in the module is

```
{tn + h,yn + h fn}
```

It does nothing more than compute the ordered pair **{tn + h,yn + h fn}**.

Next we define *Mathematica* function **Euler** that makes use of **emstep**:

```
Euler[f_,{t0_,y0_},h_,steps_][t_,y_]:=
    NestList[emstep[f,{t,y},#,h]&,{t0,y0},steps]
```

Note that **Euler** is a function of the seven variables

```
f, t0, y0, h, steps, t, y
```

The command **NestList** is useful for constructing a table of values of the result of repeated application of a function to an expression. Specifically

```
NestList[g, expr, n]
```

gives a list of the results of applying **g** to **expr** 0 through n times. For the command **Euler** we use

steps	*for*	**n**
{t0,y0}	*for*	**expr**
emstep[f,{t,y}, #,h]&	*for*	**g**

Here **emstep[f,t,y,#,h]&** is *Mathematica*'s abbreviation for the function that assigns the value **emstep[f,{t,y},{t0,y0},h]** to the ordered pair **{t0,y0}**.

The following example demonstrates the use of the command **Euler**.

Example 8.1. *Use the command* **Euler** *to find a numerical approximation to the solution of the initial value problem*

$$\begin{cases} y' = \cos(15t\,y), \\ y(0) = 1 \end{cases}$$

over the interval $0 \le t \le 1$. *Use step size* $h = 0.1$.

Solution. We use

```
ex0=Euler[Cos[15 t y],{0,1},0.1,10][t,y]
```

which results in

```
{{0,1}, {0.1, 1.1}, {0.2,1.09209}, {0.3,0.992993},
 {0.4,0.968843}, {0.5,1.05799}, {0.6,1.04991},
 {0.7,0.949935}, {0.8,0.864659},{0.9,0.806582},{1.0, 0.79593}}
```

The command

```
TableForm[ex0]
```

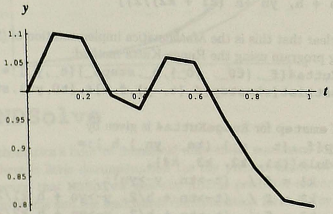
prints this information in a nice tabular form:

```
0.0   1.0
0.1   1.1
0.2   1.09209
0.3   0.992993
0.4   0.968843
0.5   1.05799
0.6   1.04991
0.7   0.949935
0.8   0.864659
0.9   0.806582
1.0   0.79593
```

The solution **ex0** can be plotted via the command **ListPlot**. This command is frequently used for plotting data as points. The option **PlotJoined->True**. draws line segments to connect the points. Here we use

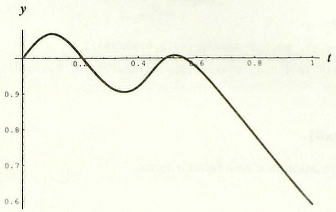
```
ListPlot[ex0,PlotJoined->True];
```

to get



A better approximation is obtained if we take $h = 0.01$ and do 100 steps. Let us skip over printing out the numerical data and do the plot directly:

```
ListPlot[Euler[Cos[15 t y],{0,1},0.01,100][t,y],
PlotJoined->True];
```



The Heun and Runge-Kutta Methods via ODE

There is a more complicated miniprogram for solving a differential equation numerically using the Heun method:

```
    Heun[f_,{t0_, y0_},h_,steps_][t_,y_]:=
        NestList[iemstep[f,{t, y},#,h]&,{t0,y0},steps]
```

Here `iemstep` iterates for `Heun` in the same way that `emstep` iterates for `Euler`. The definition is as follows.

```
    iemstep[f_,{t_, y_},{tn_,yn_},h_]:=
      Module[{k1, k2},
      k1 = f /. {t->tn, y->yn};
      k2 = f /. {t->tn + h, y->yn + h k1};
      {tn + h, yn +h (k1 + k2)/2}]
```

It should be clear that this is the *Mathematica* implementation of (2.13). Here is the corresponding program using the Runge-Kutta method:

```
    RungeKutta4[f_,{t0_, y0_},h_,steps_][t_,y_]:=
        NestList[rkmstep[f,{t, y},#,h]&,{t0,y0},steps]
```

The analog of `emstep` for `RungeKutta4` is given by

```
    rkmstep[f_,{t_, y_},{tn_, yn_},h_]:=
        Module[{k1, k2, k3, k4},
            k1 = f /. {t->tn, y->yn};
            k2 = f /. {t->tn + h/2, y->yn + h k1/2};
            k3 = f /. {t->tn + h/2, y->yn + h k2/2};
            k4 = f /. {t->tn + h, y->yn + h k3};
            {tn + h, yn +h (k1 + 2 k2 + 2 k3 + k4)/6}]
```

This is the *Mathematica* version of (3.18).

Example 8.2. *Solve the initial value problem*

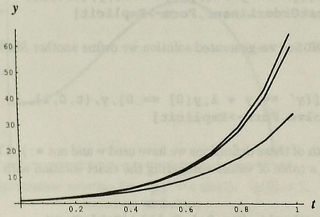
$$\begin{cases} y' = 1 - t + 4y, \\ y(0) = 1 \end{cases}$$

using the commands **Euler**, **Heun** *and* **RungeKutta4** *and simultaneously plot the three approximations.*

Solution. We use

```
Show[
{ListPlot[Euler[1 - t + 4y,{0,1},0.1,10][t,y],
PlotJoined->True,DisplayFunction->Identity],
ListPlot[Heun[1 - t + 4y,{0,1},0.1,10][t,y],
PlotJoined->True,DisplayFunction->Identity],
ListPlot[RungeKutta4[1 - t + 4y,{0,1},0.1,10][t,y],
PlotJoined->True,DisplayFunction->Identity]},
DisplayFunction->$DisplayFunction];
```

to get



9 Using **NDSolve**

NDSolve is *Mathematica*'s built-in numerical differential equations solver. Although very powerful, there is little documentation (at the present time) on how it works. It is known, however, that **NDSolve** uses the Adams predictor-corrector method to handle nonstiff problems and the backwards differentiation formula (or Gear method) for stiff problems. Stiffness occurs in a problem where there are two or more very different scales of the independent variable on which the dependent variables are changing. Stiff problems arise in several fields of science, most notably in the theory of chemical kinetics, where, say, one part of a reaction occurs in a few milliseconds while the remainder takes hours to complete. Many popular algorithms exhibit an extreme numerical instability which is not connected with any instability of the initial value problem.

NDSolve returns its solutions as **InterpolatingFunction** objects, which consist of internal *Mathematica* representations as piecewise cubic polynomials. Although it is possible to use *Mathematica's* commands **InputForm** and **FullForm** to view an **InterpolatingFunction** object, it is not usually useful to do so.

ODE can call **NDSolve** using the option **Method->NDSolve**.

Example 9.1. *Use* **ODE** *with the option* **Method->NDSolve** *to find a numerical approximation to the solution of the initial value problem*

$$\begin{cases} y' = y + 1, \\ y(0) = 0, \end{cases} \tag{9.34}$$

for $0 \le t \le 1$. *Use step size* $h = 0.1$ *and compare the approximation with the exact solution.*

Solution. To get the exact solution we define a *Mathematica* function **f** by

```
f[t_] = ODE[{y' == y + 1,y[0] == 0},y,{t,0,1},
Method->FirstOrderLinear,Form->Explicit]
```

and to get the **NDSolve**-generated solution we define another *Mathematica* function **g** by

```
g[t_] = ODE[{y' == y + 1,y[0] == 0},y,{t,0,1},
Method->NDSolve,Form->Explicit]
```

(note that in both of these definitions we have used = and not =:). The following command generates a table of values comparing the exact solution with the approximate solution:

```
Table[{f[xx],g[xx]},{xx,0,1,0.1}]//TableForm
```

to get

```
0          0.
0.105171   0.105171
0.221403   0.221404
0.349859   0.34986
0.491825   0.491826
0.648721   0.648723
0.822119   0.822121
1.01375    1.01375
1.22554    1.22554
1.4596     1.45961
1.71828    1.71829
```

Thus the solution generated by **NDSolve** is indeed very close to the exact solution ■

The output of **NDSolve** is best understood by plotting it. We can consider an **InterpolatingFunction** object to be a pseudo function; as such it can be plotted in much the same way that *Mathematica* plots an ordinary function. Furthermore, algebraic operations as well as differentiation can be performed on **InterpolatingFunction** objects, and the results of these operations can be plotted.

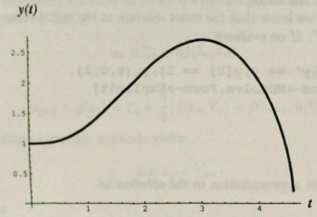
Example 9.2. *Use* **ODE** *with the options* **Method->NDSolve** *and* **PlotSolution** *to plot the approximate solution to the initial value problem*

$$\begin{cases} y\,y' = t\sin(t), \\ y(0) = 1. \end{cases}$$

Solution. We use

```
ODE[{y y' == t Sin[t], y[0] == 1},y,{t,0,4.6},
Method->NDSolve,PlotSolution->{{t,0,4.6}}]
```

to get the plot



Approximate solution to $y\,y' = t\,\sin(t)$, $y(0) = 1$, found by **NDSolve**

There are several options that control how **NDSolve** approximates the solution to an initial value problem. **MaxSteps** limits the number of steps for each calculation. With **MaxSteps->n**, the maximum number of steps taken is **n**. The default limitation is 500 steps. This limit can be effectively removed with **MaxSteps->Infinity**.

For example, to solve the initial value problem $y' = t\sin(t^2)$, $y(0) = 1$ on the interval $0 \leq t \leq 15$, we must increase the number of steps **NDSolve** will use by entering

```
ODE[{y' == y Sin[t^2],y[0] == 1},y,{t,0,15},
Method->NDSolve,MaxSteps->1000]
```

In general, *Mathematica* uses three terms to control the value of approximate numerical results. **WorkingPrecision** is simply the number of digits used in the arithmetic. The default **WorkingPrecision** is defined by **$MachinePrecision**, typically 16 on modern computers. **PrecisionGoal** defines the total number of correct significant digits, also related to the relative error in the calculation. **AccuracyGoal** defines the number of correct significant digits to the right of the decimal point, also related to the absolute error in the calculation. **NDSolve** uses the following defaults:

```
WorkingPrecision -> 16
PrecisionGoal    -> Automatic
AccuracyGoal     -> Automatic
```

Here, **Automatic** means 10 digits less than **WorkingPrecision** or 6 decimal digits on most computers. With **PrecisionGoal->Automatic** and **AccuracyGoal->Automatic**, convergence is determined by the first one to be satisfied. To insist on a particular convergence criterion, say **PrecisionGoal**, simply set **AccuracyGoal->Infinity**. To perform calculations at a higher level, say n, we normally begin with **WorkingPrecision->2n**, **PrecisionGoal->n+2** and **AccuracyGoal->n+2** and then experiment until we are satisfied with the result.

For example, we know that the exact solution to the initial value problem $y' = y$, $y(0) = 1$ is $y = e^t$. If we evaluate

```
f[t_] = ODE[{y' == y,y[0] == 1},y,{t,0,1},
      Method->NDSolve,Form->Explicit]
```

we can use

```
N[f[1],10]
```

to obtain a 10 digit approximation to the solution as

```
2.718289889
```

However, the correct 10 digit approximation is

```
2.718281828
```

To obtain a more precise solution, we can enter

```
g[t_] = ODE[{y' == y,y[0] == 1},y,{t,0,1},Method->NDSolve,
WorkingPrecision->25,AccuracyGoal->12,PrecisionGoal->12,
Form->Explicit];
```

Then when we enter **g[1]** we get the value of e with 25 digits of precision:

```
2.718281828477414818469359 7
```

10 Implicit Methods and Stiff Equations

In this section we discuss a class of methods which attempts to solve certain instability problems.

Implicit Methods

Recall that the formula

$$Y_{n+1} = Y_n + \frac{h}{2}(f(t_n, Y_n) + f(t_{n+1}, Y_{n+1})),$$

used to introduce the Heun method, had Y_{n+1} on both sides of the equation. This equation implicitly defines Y_{n+1} and in general, this presents a difficult problem. If we define a new function g by

$$g(u) = Y_n + \frac{h}{2}(f(t_n, Y_n) + f(t_{n+1}, u)),$$

then we can attempt to find the next estimate Y_{n+1} by finding a fixed point of g. That is, we seek a point $\hat{u}$ such that $g(\hat{u}) = \hat{u}$. Although several methods can be used (from simple bracketing methods to Newton's method) a simple iteration is usually sufficient. Thus if we define

$$u_0 = Y_n + hf(t_n, Y_n)$$

and let

$$u_{j+1} = g(u_j) = Y_n + \frac{h}{2}(f(t_n, Y_n) + f(t_{n+1}, u_j)),$$

then the stability of implicit methods yields

$$\lim_{j\to\infty} u_j = Y_{n+1}$$

provided that

$$\frac{h}{2}\left|\frac{\partial f(t,y)}{\partial y}\right| < 1. \tag{10.35}$$

Condition 10.35 is equivalent to

$$|g'(u)| < 1,$$

a sufficient condition for the sequence of iterates u_i to converge to a unique fixed point of g.

The principal advantage of all implicit methods is that they can be made absolutely stable, which will be defined and discussed in Section 11. The principal disadvantage is the added computational demands resulting from a required iteration at each step. The number of iterations used to approximate a root of f is determined by the error tolerance and the maximum number of iterations normally required to make the method deterministic.

Other algorithms also have implicit forms. For example, a third-order implicit Runge-Kutta method has been implemented in **ODE**. It is defined by

$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \left(\frac{h}{2}\right), y_n + \left(\frac{h}{4}\right)(k_1 + k_2)\right), \\
k_3 &= f(t_n + h, y_n + h\,k_2), \\
y_{n+1} &= y_n + \frac{h}{6}(k_1 + 4k_2 + k_3),
\end{aligned}$$

for $n = 0, \ldots, N-1$, where $t_n = t_0 + n\,h$. The quantity k_2 is computed by iterating the second equation until

$$\|k_2 - \widehat{k}_2\|_2 = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{k_2(i) - \widehat{k}_2(i)}{\max\{|k_1(i)|, |\widehat{k}_2(i)|, 1\}}\right)^2} < \epsilon,$$

where n is equal to the number of equations and ϵ is the desired **Tolerance** with a default value of 10^{-5}. To restrict the number of iterations, a maximum number of iterations is defined by **ODEMaxSteps** with a default value of 500. Since this method is absolutely stable, it is particularly well suited to *stiff* systems of equations, for example, like those normally found in biochemical kinetics.

Stiff Equations

The term **stiff** applies to differential equations in which there are two or more very different scales of the independent variable on which the dependent variables are changing. For example, consider the second order equation

$$y'' = 100\,y,$$

with general solution

$$y = C_1 e^{-10t} + C_2 e^{10t}$$

for arbitrary constants C_1 and C_2. If the initial conditions are

$$y(0) = 1 \qquad \text{and} \qquad y'(0) = -10,$$

then the true solution is $y = e^{-10t}$. All numerical integration methods would start off decaying as e^{-10t}, but every explicit method will explode as t becomes large. This occurs because our numerical approximation is precisely that, an approximation to the true solution. Therefore what we actually observe is

$$y_{\text{approx}} \approx e^{-10t} + \epsilon e^{10t}.$$

Although taking a very small step size will make ϵ small, eventually the second term becomes dominant for explicit methods. Also, increasing the precision at which the calculations are performed will only delay the effect of the second term.

Although stiffness is usually associated with higher order or systems of differential equations, the round-off error characteristics of a particular numerical method applied to a stiff system can be predicted by examining the round-off error produced when the method is applied to the test initial value problem

$$\begin{cases} y' = \lambda y, \\ y(0) = Y_0 \end{cases}$$

where λ is a negative real number. The following example illustrates the effective use of implicit methods for a stiff equation by applying a suite of numerical integration techniques to the test equation.

Example 10.1. *Describe graphically the behavior of all numerical approximations to the solution of the initial value problem*

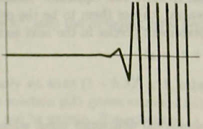
$$\begin{cases} y' = -100y, \\ y(0) = 1/3 \end{cases}$$

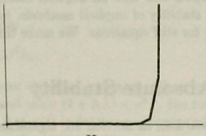
over the interval $0 \le t \le 1$ *using a step size of* 0.05.

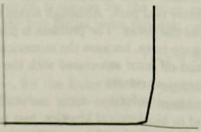
Solution. We use

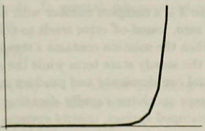
```
ODE[{y'==-100y,y[0]==1/3},y,{t,0,1},Method->AllNumerical,
StepSize->0.05,PlotSolution->{{t,0,1}}]
```

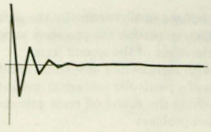
to obtain the following graphs:

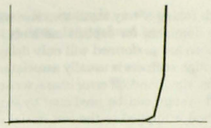

Euler

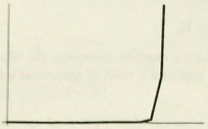

Heun

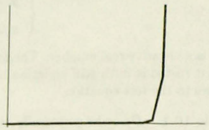

Runge-Kutta

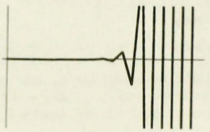

Runge-Kutta (45)

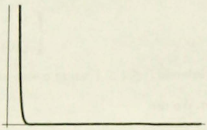

Implicit Runge-Kutta


Second Order Euler


Milne


Adams-Bashforth


Bulirsch-Stoer


NDSolve

`NDSolve` also uses an implicit technique to solve this equation. Hence, we see that the stability of implicit methods, in general, causes them to be the procedures of choice for stiff equations. We make this statement precise in the next section.

11 Absolute Stability

As we have seen in Section 10, significant numerical difficulties occur when we approximate the solution of a differential equation which contains terms of the form $e^{\lambda t}$, where λ is a complex number with negative real part. Although terms like these decay to zero, round-off error tends to conceal the decay. The problem is particularly serious when the solution contains a steady state term, because the numerical method will seek the steady state term while the round-off error associated with the decaying exponential can dominate and produce meaningless results.

Problems involving rapidly decaying transient solutions occur naturally in the study of damped springs, control systems and in biochemical kinetics, just to name a few. The accepted way to test a method's ability to deal with an equation is to apply it to the simple test initial value problem

$$\begin{cases} y' = \lambda y, \\ y(0) = Y_0, \end{cases} \tag{11.36}$$

where λ is a negative real number. To understand how we might perform a stability analysis for a particular method, let us consider the Euler method applied to the test initial value problem. If we let $h = (b-a)/N$ and $t_k = a + k\,h$, for $k = 0, 1, \ldots, N$, then (11.36) implies that

$$Y_{k+1} = (1 + h\,\lambda)Y_k,$$

for $k = 0, 1, \ldots, N-1$. By induction we have

$$Y_{k+1} = (1 + h\,\lambda)^{k+1} Y_0.$$

Since the exact solution of (11.36) is assumed to be

$$y(t) = y_0 e^{\lambda t},$$

the absolute error is

$$|y(t_k) - Y_k| = |e^{\lambda hk} - (1 + h\,\lambda)^k|\,|y_0|$$

and the accuracy is determined by how well the term $1 + h\,\lambda$ approximates

$$e^{h\lambda} = 1 + h\lambda + \frac{(h\,\lambda)^2}{2!} + \cdots.$$

If we introduce a rounding error in the initial condition for the Euler method,

$$Y_0 = y_0 + \delta_0$$

then at the k^{th} step the rounding error becomes

$$\delta_k = (1 + h\,\lambda)^k \delta_0.$$

Obviously we want $(1 + h\,\lambda)^k < 1$, otherwise the error increases. Observe that if $\lambda > 0$, the solution $y(t)$ grows exponentially and since $(1 + h\,\lambda) < e^{h\,\lambda}$, the rounding error may not be serious. For equations, $\lambda < 0$ so $y(t)$ is decaying exponentially; any growth in rounding error eventually dominates the approximation.

The situation is similar for the other methods. In general, whenever we apply a method to the test initial value problem, we get

$$Y_{k+1} = S(h\,\lambda)Y_k.$$

For example, for the Euler method we have $S(h\,\lambda) = 1 + h\lambda$, for the Heun method we have $S(h\,\lambda) = 1 + h\lambda + (h\lambda)^2/2!$ and for the Runge-Kutta method we have

$$S(h\,\lambda) = 1 + h\lambda + \frac{(h\,\lambda)^2}{2!} + \frac{(h\,\lambda)^3}{3!} + \frac{(h\,\lambda)^4}{4!}.$$

Note that in each case, the order of the method defines the function S.

For multistep methods, like the Milne and Adams-Bashforth methods, the function S becomes

$$S(z, h\lambda) = (1 - h\lambda b_m)z^m - (a_{m-1} - h\lambda b_{m-1})z^{m-1} - \cdots - (a_0 - h\lambda b_0),$$

where the result of applying the method to the test equation is of the form

$$Y_{k+1} = a_{m-1}Y_k + \cdots + a_0 Y_{k+1-m} + h\lambda(b_m Y_{k+1} + b_{m-1}Y_k + \cdots + b_0 Y_{j+1-m}),$$

where $k = m-1, \ldots, N-1$. To describe the amount of step size reduction needed for a particular method to be used on a problem, we need the following definition.

Definition. *The* **region R of absolute stability** *for a single step method is defined by*

$$R = \{h\lambda \in \mathbb{C} \mid |S(h\lambda)| < 1\},$$

and for a multistep method by

$$R = \{\, h\lambda \in \mathbb{C} \mid |\beta_k| < 1 \textit{ for all roots } \beta_k \textit{ of } S(z, h\lambda) = 0\}.$$

For example, the region of absolute stability for the Euler method is the circle in the complex plane of radius 1 and centered at -1:

$$R = \{h\lambda \,||1 + h\lambda| < 1\}.$$

Generally, numerical methods can be applied to stiff equations only if $h\lambda$ is in the region of absolute stability, which for a given problem places a restriction on the size of h and usually forces many small steps to be taken as the approximation is computed. Since the region of absolute stability of a method is generally the critical factor in producing accurate solutions to stiff equations, special numerical methods have been developed with as large a region of absolute stability as possible. This leads to the following definition.

Definition. *A numerical method is said to be* **A-stable** *if its region of absolute stability contains the half-plane* $\{h\lambda \in \mathbb{C} \mid \mathrm{Re}(h\lambda) < 0\}$.

Applying a similar analysis to the implicit Euler method, we see that the rounding error at the k^{th} step becomes

$$\delta_k = \frac{\delta_0}{(1 - h\lambda)^k}$$

and the rounding error goes to zero even as $h \to \infty$. Therefore, this method is A-stable. The implicit Runge-Kutta method included in **ODE** is a third order accurate A-stable method and it is well suited to solving stiff equations. In general, implicit methods are A-stable and some sophisticated implicit multistep methods have been developed (see [Gear]).

12 Adaptive Step Size and Error Control

All modern numerical differential equation solvers exert some adaptive control over their own progress, making frequent adjustments to the step size. To see why the step size needs to be adjusted, let us consider a simple example using the Euler method.

Example 12.1. *Use the Euler method to find an approximate solution to*

$$\begin{cases} y' = -100y, \\ y(0) = 1/3 \end{cases} \tag{12.37}$$

over the interval $0 \le t \le 1$ *using step size* $h = 0.2$. *Compare the approximate solution with the exact solution.*

Solution. First let us note that the exact solution of (12.37) is given by

$$y_{\text{exact}}(t) = (1/3)e^{-100t};$$

consequently $y_{\text{exact}}(t)$ is very nearly 0 for $t \ge 0.1$. However, the Euler method gives very different results. The Euler method formula for (12.37) is easy to find:

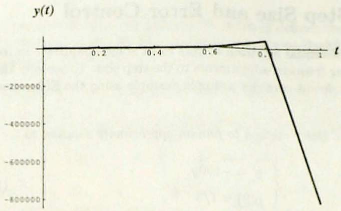
$$y_{n+1} = y_n + (0.2)(-100y_n) = -19y_n. \tag{12.38}$$

Using (12.38), we construct the following table:

n	t_n	Y_n	$y_{\text{exact}}(t_n)$
0	0.0	.333	.333
1	0.2	−6.33	$6.87\,10^{-10}$
2	0.4	120	0
3	0.6	−2290	0
4	0.8	43400	0
5	1.0	−825000	0

So the approximate solution obtained from the Euler method is very bad. Instead of tending to 0, the Y_n's oscillate between positive and negative values whose absolute value is rapidly increasing. Here is the plot obtained with the command

```
ODE[{y' == -100y,y[0] == 1/3},y,{t,0,1},
Method->Euler,StepSize->0.2,
PlotSolution->{{t,0,1},PlotRange->All}]
```



Euler approximation to $y' = -100y$, $y(0) = 1/3$
plotted over the interval $0 < t < 1$ without adaptive step control

Of course, better results are obtained with smaller step size or with a more sophisticated technique such as Runge-Kutta. However, in this section we explore a different technique, that of **adaptive step size**. Adaptive step size control in **ODE** is activated using the option **VariableStepSize->True**. In addition to this option, several other options can affect the overall outcome. **StepSize** sets the initial step size, **MaxStepSize** sets the maximum value for the step size and **Tolerance** defines the maximum allowable relative error at each integration step.

Most of the methods in **ODE** use a technique called **step doubling**, when using the option **VariableStepSize->True**. Each step is taken twice, once as a full step, then as two half steps. Then the two estimates are compared to the tolerance defined by **Tolerance**. If the relative difference is greater than the tolerance, then the step size is halved and the process is begun again. If the relative difference is significantly smaller than the tolerance, then the step size is doubled for the next iteration.

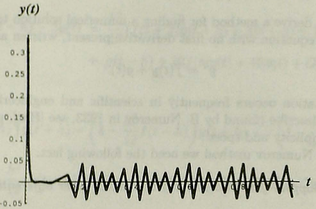
Quite often **VariableStepSize->True** will produce an accurate solution, even for a stiff equation such as $y' = -100y$, but the price is a requirement for many small steps. These small steps are automatically created by *Mathematica*, so in general it will take longer for **ODE** to solve a problem with **VariableStepSize->True** than it would with
VariableStepSize->False.

Example 12.2. *Use the Euler method to find an approximate solution to* (12.37) *over the interval* $0 \le t \le 1$ *using an adaptive step size control.*

Solution. We use

```
ODE[{y' == -100y,y[0] == 1/3},y,{t,0,1},Method->Euler,
VariableStepSize->True,StepSize->0.2,Tolerance->0.05,
PlotSolution->{{t,0,1},PlotRange->All}]
```

to obtain the plot



Euler approximation to $y' = -100y, \quad y(0) = 1/3$
plotted over the interval $0 < t < 1$ with adaptive step control
using a tolerance of 0.05

Thus the option **VariableStepSize->True** causes **ODE** to find a much more accurate solution. Even more accurate results would be obtained by setting **Tolerance** nearer to its default value of 0.00001 ■

Adaptive step size control is more difficult to implement in multistep methods such as the **Milne** method and the **AdamsBashforth** method. These methods derive some of their simplicity by assuming a uniform step size over the interval of interpolation. Whenever this assumption changes, new points within the interval already covered must be computed. Two methods are typically used. Either interpolation is used to generate intermediate points or the method is restarted at the appropriate new starting point using **RungeKutta4** or some other single step method. **ODE** adopts latter technique for all multistep methods.

For extrapolation techniques such as the **BulirschStoer** method, estimates of the solution can be generated by changing the order of the method, where the orders are defined by $\{2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96\}$. If the estimates computed for orders $n-1$ and n lead to an acceptable tolerance, then the final value is returned. If the maximum order is reached, then the step size is halved and the process is started over.

Finally, implicit methods such as the **ImplicitRungeKutta** method iterate at each step until the tolerance is achieved. Because of this, these methods require another control parameter called **ODEMaxSteps**, set to a default value of 500. Obviously, **Tolerance** and **ODEMaxSteps** affect each other and some problems may require a larger value for **ODEMaxSteps**.

13 The Numerov Method

In this section we derive a method for finding a numerical solution to a second order linear differential equation with no first derivative present, written as

$$y'' = f(t)y + g(t).$$

This type of equation occurs frequently in scientific and engineering applications. The method we describe (found by B. Numerov in 1933, see [Har], p.142) is popular because of its simplicity and speed.

To derive the Numerov method we need the following fact.

Lemma 13.1. Suppose y is a solution of the second order differential equation

$$y'' = f(t)y + g(t). \tag{13.39}$$

Let $z(t) = \left(1 - \frac{h^2}{12}f(t)\right)y(t)$, where h is some number; then for $h \longrightarrow 0$

$$z(t+h) + z(t-h) - \left(2 + \frac{h^2 f(t)}{1 - \frac{h^2}{12}f(t)}\right)z(t)$$
$$= \frac{h^2}{12}\left(g(t+h) + g(t-h) + 10g(t)\right) + O(h^3). \tag{13.40}$$

Proof. The Taylor expansion of $y(t \pm h)$ is

$$y(t \pm h) = y(t) \pm h\,y'(t) + \frac{h^2}{2}y''(t) \pm \frac{h^3}{6}y'''(t) + \frac{h^4}{24}y^{(4)}(t) + O(h^5).$$

We add $y(t+h)$ and $y(t-h)$ to obtain

$$y(t+h) + y(t-h) = 2y(t) + h^2y''(t) + \frac{h^4}{12}y^{(4)}(t) + O(h^5). \tag{13.41}$$

Similarly

$$y''(t+h) + y''(t-h) = 2y''(t) + h^2y^{(4)}(t) + O(h^3),$$

so that

$$h^2y^{(4)}(t) = y''(t+h) + y''(t-h) - 2y''(t) + O(h^3). \tag{13.42}$$

From (13.41) and (13.42) we obtain

$$y(t+h) + y(t-h) - 2y(t) = \frac{h^2}{12}\left(y''(t+h) + y''(t-h) + 10y''(t)\right) \tag{13.43}$$
$$+ O(h^3). \tag{13.44}$$

Now assume that (13.39) holds. From (13.44) we obtain

$$\begin{aligned} y(t+h)+y(t-h)-2y(t) &= \frac{h^2}{12}(f(t+h)y(t+h)+g(t+h)+f(t-h)y(t-h) \\ &\quad + g(t-h)+10f(t)y(t)+10g(t)+O(h^3). \end{aligned}$$

or

$$\begin{aligned} &\left(1-\frac{h^2}{12}f(t+h)\right)y(t+h)+\left(1-\frac{h^2}{12}f(t-h)\right)y(t-h)-2\left(1-\frac{h^2}{12}f(t)\right)y(t) \\ &= \frac{h^2}{12}(g(t+h)+g(t-h)+10g(t)+12f(t)y(t))+O(h^3). \qquad (13.45) \end{aligned}$$

Then we can rewrite (13.45) as

$$\begin{aligned} &z(t+h)+z(t-h)-2z(t)+O(h^3) \\ &= \frac{h^2}{12}\left(g(t+h)+g(t-h)+10g(t)+12\left(1-\frac{h^2}{12}f(t)\right)^{-1}z(t)\right), \end{aligned}$$

from which we get (13.40) ■

The **Numerov method** finds a numerical approximation to the solution of the second order linear initial value problem

$$\begin{cases} y''=f(t)y+g(t), \\ y(a)=Y_0,\ y'(a)=Y_1 \end{cases} \qquad (13.46)$$

on an interval $a \le t \le b$. To derive the iteration scheme for the Numerov method, we need a discrete version of Lemma 13.1. To this end, let h be a small positive number. We subdivide the interval $a < t < b$ as $a = t_0 < t_1 < \cdots < t_N = b$, where $t_{j+1} - t_j = h$ for $j = 1, \ldots, n$. Let

$$w_n = \frac{h^2 f(t_n)}{1-\dfrac{h^2}{12}f(t_n)}. \qquad (13.47)$$

Then z_0 and z_1 are given by

$$z_0 = \left(1-\frac{h^2}{12}f(t_0)\right)Y_0 \quad \text{and} \quad z_1 = \left(1-\frac{h^2}{12}f(t_1)\right)(Y_0+hY_1). \qquad (13.48)$$

From (13.39) we get

$$z_{n+2} = (2+w_{n+1})z_{n+1} - z_n + \frac{h^2}{12}(g(t_{n+2})+10g(t_{n+1})+g(t_n)). \qquad (13.49)$$

We can use (13.48) and (13.49) to determine z_n for $n = 2, 3, \ldots$ Finally, we determine Y_n for $n = 2, 3, \ldots$ by

$$Y_n = \left(1 - \frac{h^2}{12} f(t_n)\right)^{-1} z_n. \tag{13.50}$$

The *Mathematica* implementation of (13.47)—(13.50) is as follows:

```
Numerov[f_,g_,{t0_,Y0_,Y1_},h_,steps_][t_,y_]:=
   Module[{tt,ff,gg,z0,z1,tmp,z,zsol},
      ff[tt_,n_]:= f /. t -> tt + n h;
      gg[tt_,n_]:= g /. t -> tt + n h;
      z0 = Y0(1 - (h^2/12)ff[t0,0]);
      z1 = (1 - (h^2/12)ff[t0,1])(Y0 + h Y1);
      tmp = Table[{1,1/((1 - (h^2/12)ff[t0,k]))},
        {k,0,steps}];
      zsol = NestList[nmstep[ff,gg,{t,z},#,h]&,
        {t0,z0,z1},steps];
      Simplify[Map[Drop[#,{3}]&,zsol] tmp]]

nmstep[f_,g_,{t_,z_},{tn_,zn_,znp1_},h_]:=
   Module[{wnp1},
      wnp1 = h^2 f[tn,1]/(1 - (h^2/12)f[tn,1]);
      {tn + h,znp1,(2+wnp1)znp1 - zn +
        (h^2/12)(g[tn,2] + 10g[tn,1] + g[tn,0])}]
```

Let us see how the Numerov method works in practice.

Example 13.1. *Use the Numerov method to find a numerical approximation to the second order initial value problem*

$$\begin{cases} y'' = -t^3 y + 1, \\ y(0) = 1,\ y'(0) = 0. \end{cases}$$

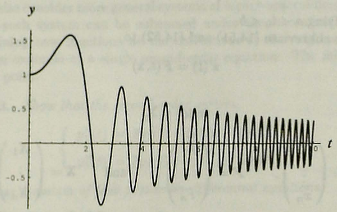
Solution. We use

```
ODE[{y'' == -t^3 y + 1,y[0] == 1,y'[0] == 0},y,{t,0,10},
Method->Numerov,StepSize->0.05,NumericalOutput->None,
PlotSolution->{{t,0,10}}]
```

or

```
ListPlot[Numerov[-t^3,1,{0,1,0},0.05,200][t,y],
PlotJoined->True]
```

to obtain the plot



Numerov approximation to
$$y'' = -t^3 y + 1, \quad y(0) = 1, \quad y'(0) = 0$$

Although the above plot can also be obtained using **NDSolve**, for example using

```
ODE[{y'' == -t^3 y + 1,y[0] == 1,y'[0] == 0},y,{t,0,10},
Method->NDSolve,MaxSteps->2000,NumericalOutput->None,
PlotSolution->{{t,0,10},PlotPoints->400}]
```

Numerov is faster.

14 Systems of Ordinary Differential Equations

So far the differential equations we have discussed have involved only one unknown function. In many applications, however, there is more than one unknown function and more than one equation. A **system of first-order differential equations** is written in the form

$$\left\{\begin{aligned} \frac{dx_1}{dt} &= F_1(t, x_1, \ldots, x_n), \\ \vdots\; & \quad \vdots \\ \frac{dx_n}{dt} &= F_n(t, x_1, \ldots, x_n). \end{aligned}\right. \tag{14.51}$$

The independent variable is t, and the unknown functions are $x_1(t), \ldots, x_n(t)$. The functions $F_1, \ldots, F_n$ are assumed to be given. A **solution of** (14.51) consists of a collection of differentiable functions $\{x_1(t), \ldots, x_n(t)\}$ that satisfy the system (14.51) for all values of t in some interval $a < t < b$.

We may also speak of an **initial value problem** corresponding to a system of differential equations. Such an initial value problem consists of a system of the form (14.51) together with n initial conditions

$$x_1(t_0) = X_1, x_2(t_0) = X_2, \ldots, x_n(t_0) = X_n \tag{14.52}$$

for some t_0 satisfying $a < t_0 < b$.

It is useful to abbreviate (14.51) and (14.52) to

$$\mathbf{x}'(t) = \boldsymbol{F}(t, \mathbf{x}) \tag{14.53}$$

and

$$\mathbf{x}(t_0) = \mathbf{X}, \tag{14.54}$$

where

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad \boldsymbol{F} = \begin{pmatrix} F_1 \\ \vdots \\ F_n \end{pmatrix}, \quad \text{and} \quad \mathbf{X} = \begin{pmatrix} X_1 \\ \vdots \\ X_n \end{pmatrix}.$$

We can think of $\mathbf{X}$ as a point or vector in the space $\mathbb{R}^n$ of n-tuples of real numbers, and $t \longmapsto \mathbf{x}(t)$ as a vector-valued function or (parametrized) curve in $\mathbb{R}^n$. The vector notation in (14.53) and (14.54) contains no new mathematics; however, it is a very convenient abbreviation.

An important observation is that a higher-order single differential equation can be written in terms of first-order systems. The following example illustrates the procedure.

Example 14.1. *Show that the second-order equation*

$$y'' + p(t)y'(t) + q(t)y(t) = r(t) \tag{14.55}$$

can be written as a system of two first-order differential equations.

Solution. By writing $x_1(t) = y(t)$ and $x_2(t) = y'(t)$, we convert (14.55) to the system

$$\begin{cases} \dfrac{dx_1}{dt} = x_2(t), \\ \dfrac{dx_2}{dt} = -p(t)x_2(t) - q(t)x_1(t) + r(t). \end{cases} \tag{14.56}$$

Clearly, (14.56) is a special case of (14.51) (with $n = 2$) by taking

$$F_1(t, x_1, x_2) = x_2 \quad \text{and} \quad F_2(t, x_1, x_2) = -p(t)x_2 - q(t)x_1 + r(t) \qquad \blacksquare$$

More generally, an n^{th}-order differential equation gives rise to n first-order differential equations.

Example 14.2. *Write $y''' + y'' + y' + y = 0$ as a system of three first-order differential equations.*

Solution. We put $x_1 = y$, $x_2 = y'$, and $x_3 = y''$. This yields the system

$$\begin{cases} x_1' = x_2, \\ x_2' = x_3, \\ x_3' = -x_1 - x_2 - x_3 \end{cases} \qquad \blacksquare$$

One could also consider more general systems of *higher-order* differential equations. However, any such system can be subsumed under the formalism of a first-order system by defining new functions as the derivatives of the original functions, just as in the above example of a single second-order equation. The following example illustrates this point.

Example 14.3. *Show that the second-order system*

$$\begin{cases} y_1''(t) = 3y_1(t) + y_1'(t) + 4y_2(t), \\ y_2''(t) = 3y_1(t) + 4y_1'(t) + 5y_2(t) \end{cases} \tag{14.57}$$

can be written as a system of four first-order differential equations.

Solution. We simply define

$$x_1 = y_1, \qquad x_2 = y_1', \qquad x_3 = y_2, \qquad x_4 = y_2',$$

and compute the derivatives to obtain

$$\begin{cases} x_1' = x_2, \\ x_2' = 3x_1 + x_2 + 4x_3, \\ x_3' = x_4, \\ x_4' = 3x_1 + 4x_2 + 5x_3. \end{cases} \tag{14.58}$$

Then (14.58) is the first-order system equivalent to (14.57) ■

15 Numerical Solutions of Systems of Ordinary Differential Equations

Earlier, we discussed numerical methods for solving initial value problems associated with first-order differential equations. Most such methods can be extended to systems of first-order differential equations more or less automatically. The trick is to use vectors instead of scalars. For example, suppose we are given a first-order initial value problem

$$\begin{cases} \mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}), \\ \mathbf{y}(t_0) = \mathbf{Y}_0, \end{cases} \tag{15.59}$$

where now $t \longmapsto \mathbf{y}(t)$ and $(t, \mathbf{y}) \longmapsto \mathbf{f}(t, \mathbf{y})$ are vector-valued functions and $\mathbf{Y}_0$ is a vector. It is true that we can write out these objects in terms of their components:

$$\begin{cases} \mathbf{y}(t) = (y_1(t), \ldots, y_n(t)), \\ \mathbf{f}(t, \mathbf{y}) = \Big(f_1(t, (y_1(t), \ldots, y_n(t))), \ldots, f_n(t, (y_1(t), \ldots, y_n(t)))\Big), \\ \mathbf{Y}_0 = (y_1(t_0), \ldots, y_n(t_0)). \end{cases} \tag{15.60}$$

But usually (15.60) is more of a hindrance than a help for understanding the theory; the compact notation using $\mathbf{y}(t)$, $\mathbf{f}(t, \mathbf{y})$ and $\mathbf{Y}_0$ is better [8]. Of course, when specific systems are solved components must eventually be inserted, although it is best to keep the compact boldface notation as long as possible in the solution process. Since boldface letters are used to denote vectors and vector-valued functions, a great deal of the theory given in the earlier sections can be generalized to systems simply by replacing appropriate nonboldface letters by boldface letters. Notice that t and t_0 are never boldface.

For example, the Euler method for systems can be explained as follows. Just as with its single-equation counterpart, the objective is to construct an approximation to the solution of the initial value problem (15.59) for $a \le t \le b$. We divide the interval $a \le t \le b$ into equal subintervals:

$$a = t_0 < t_1 < \cdots < t_N = b,$$

where $h = t_{k+1} - t_k$ is the **step size**. The generalization to systems of the **Euler method formula** (1.6) is just

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\,\mathbf{f}(t_k, \mathbf{Y}_k), \tag{15.61}$$

where each $\mathbf{Y}_k$ is a vector.

Example 15.1. *Find a numerical approximation to the solution of the initial value problem*

$$\begin{cases} x' = y\sin(t), & x(0) = 1, \\ y' = -x\cos(t), & y(0) = 0 \end{cases} \tag{15.62}$$

over the interval $0 \le t \le 10$ *with step size* $h = 1.0$. *Use the Euler method.*

Solution. Write $\mathbf{Y}_k = (Y_{k0}, Y_{k1})$; the Euler method formula (15.61) for the initial value problem (15.62) with step size $h = 1.0$ becomes

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + 1.0(Y_{k1}\sin(t_k), -Y_{k0}\sin(t_k))$$

for $0 \le n \le N - 1$. The interval $0 \le t \le 10$ is divided into 10 equal pieces, so $N = 10$. We are given $\mathbf{Y}_0 = (1, 0)$; then

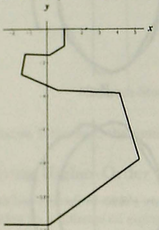
$$\mathbf{Y}_1 = \mathbf{Y}_0 + 1.0f(0, (1, 0)) = (1, 0) + (0, -1) = (1, -1),$$

[8] It was not until the beginning of the twentieth century that vectors began to replace lists of components in the mathematical literature. The driving force was vector analysis as formulated by Gibbs (see [Crowe]). In *Mathematica* a vector is a list.

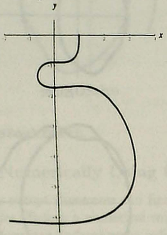
and so forth. We obtain the following table:

$k = t_k$	$\mathbf{Y}_k$
0	(1.0, 0)
1	(1.0 − 1.0)
2	(0.1586, −1.5403)
3	(−1.2420, −1.4743)
4	(−1.45012, −2.7040)
5	(0.5962, −3.6518)
6	(4.0981, −3.8210)
7	(5.1657, −7.7558)
8	(0.07025, −11.6502)
9	(−11.4560, −11.6400)
10	(−16.2531, −22.0780)

The first plot below is the phase plot of the data given by this table. If we had used a step size of 0.01, we would have obtained the second plot.



Phase plot of (15.62)
with step size 1.0



Phase plot of (15.62)
with step size 0.01

In the next example, we compare the phase portraits produced by the various numerical methods.

Example 15.2 *Use* **ODE** *with the option* **Method->AllNumerical** *to solve numerically the initial value problem*

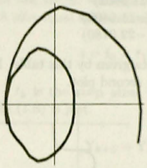
$$\begin{cases} x' = y, & x(0) = 1 \\ y' + x = -\sin(x^2 + t^2), & y(0) = 0 \end{cases}$$

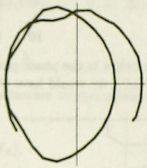
over the interval $-\pi \le t \le 2\pi$ *using step size* 0.02. *Draw the phase portraits.*

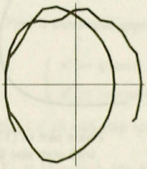
Solution. We use

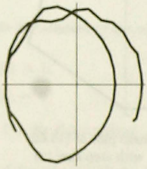
```
ODE[{x' == y,y' + x == -Sin[x^2 + t^2],x[0] == 1,y[0] == 0},
{x,y},{t,-Pi,2Pi},
Method->AllNumerical,ODETrace->False,
NumericalOutput->None,StepSize->0.2,
PlotPhase->{{t,-Pi,2Pi}}]
```

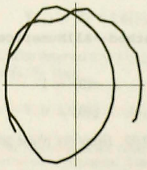
to obtain the following phase portraits:

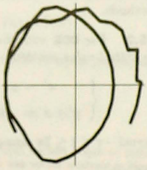

Euler

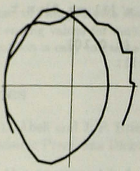

Heun

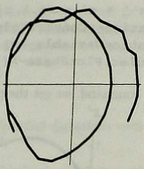

Runge-Kutta

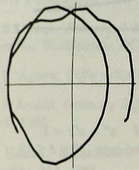

Runge-Kutta (45)

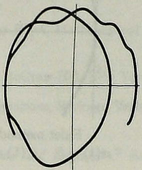

Implicit Runge-Kutta


Second-order Euler


Milne


Adams-Bashforth


Bulirsch-Stoer


NDSolve

`NDSolve` is the best; the runner-up is **`AdamsBashforth`**.

Solving Higher-Order Equations Numerically Using ODE

`NDSolve` and all of **`ODE`**'s numerical solvers except **`Numerov`** can find a numerical solution to a differential equation of any order. Before a numerical solver begins its work, the command **`Transformation->ConvertToSystem`** is called to convert the differential equation to a system. The conversion can be done either automatically or more explicitly using **`ConvertToSystem`**. We illustrate the two procedures:

Example 15.3. *Use the Euler method with step size* 0.1 *to solve the second-order initial value problem*

$$\begin{cases} x'' + x'x = 1, \\ x(1) = 0,\ x'(1) = 0. \end{cases}$$

Find the phase plot of the solution over the interval $0 \leq t \leq 10$.

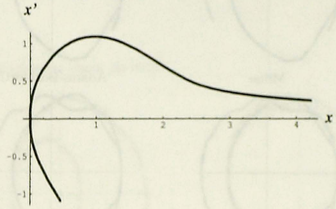
Solution. The simplest command to use is

```
ODE[{x'' + x' x == 1,x[1] == 0,x'[1] == 0},x,{t,0,10},
Method->Euler,PlotPhase->{{t,0,10}}];
```

A more complicated command that accomplishes the same thing is

```
ODE[ODE[{x'' + x' x == 1,x[1] == 0,x'[1] == 0},x,t,
Transformation->ConvertToSystem,
TransformationVariable->w],{w1,w2},{t,0,10},
Method->Euler,PlotPhase->{{t,0,10}}];
```

Using either command, we get the plot



Euler method solution of $x'' + x'x = 1$, $x(1) = 0$, $x'(1) = 0$ plotted over $0 \leq t \leq 10$

Recommendations

In closing, it seems appropriate to give some recommendations on methods to use for solving a general problem. In general, **NDSolve** is to be used for all numerical solutions. It is fast and accurate. It includes a dynamic adjustment of stepsize determined by single step error estimates and it uses efficient algorithms for both non-stiff and stiff systems. The other algorithms have been included for pedagogical reasons, since they describe the general concepts used in **NDSolve**. All of the methods we have discussed converge in the sense that the truncation error goes to zero with the step size. However, the Milne method is unstable for some differential equations, making it less desirable to use. It is included for historical reasons and because its derivation is easier to understand. One way to rate the remaining methods is to apply them to a variety of problems and compare the computer time used and the results. Given a particular method, there is probably a special problem and a step size for which this method is better than all others. Therefore, a superior method can be better than others for a *class* of problems, but not necessarily best for any particular problem.

If the equation is stiff, then an implicit method should be used. Here the desired accuracy will determine the method, but in general, implicit multistep methods are commonly used. Since no such method currently exists in **ODE**, the best choice will be **ImplicitRungeKutta**. For nonstiff equations, the decision is normally made based on the complexity of the functions being evaluated. If the functions are relatively simple, then **BulirschStoer** will be the most efficient, while the **AdamsBashforth** predictor-corrector method is favored when the evaluation of the functions is complicated. The use of Runge-Kutta methods are normally restricted to finding starting and possibly ending values for multistep and extrapolation methods, or to problems where the function is easy to evaluate and the accuracy needed is small, about 10^{-4}.

References

[AbBr] M. L. Abell and J. P. Braselton, *Differential Equations with Mathematica*, Academic Press, San Diego, CA, 1992.

[AbSt] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, New York, 1965.

[Acton] F. S. Acton, *Numerical Methods that Work*, Mathematical Association of America, Washington, DC, 1990.

[Agnew] R. P. Agnew, *Differential Equations*, McGraw-Hill, New York, 1960.

[Arnold] V. I. Arnold, *Ordinary Differential Equations*, Springer-Verlag, Berlin-New York, 1992.

[Bahd] T. Bahder, *Mathematica for Scientists and Engineers*,[9] Addison-Wesley, Reading, MA, 1995.

[Braun] M. Braun, *Differential Equations and Their Applications*, Fourth Edition, Springer-Verlag, Berlin-New York, 1993.

[Bugl] P. Bugl, *Differential Equations, Matrices and Models*, Prentice-Hall, Englewood Cliffs, NJ, 1995.

[StoBul] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, Springer-Verlag, New York, 1980.

[Butcher] J. C. Butcher,*The Numerical Analysis of Ordinary Differential Equations*, John Wiley,New York,1987.

[CelGra] M. Celia and W. Gray, *Numerical Methods for Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[CrHa] T. M. Creese and R. M. Haralick, *Differential Equations for Engineers*, McGraw-Hill, New York, 1978.

[Crowe] M. J. Crowe, *A History of Vector Analysis*, Dover Publications, New York, 1994.

[Davis] H. T. Davis, *Introduction to Nonlinear Differential and Integral Equations*, U.S. Atomic Energy Commission, Washington, DC, 1960. Reprint by Dover Publications, New York, 1962.

[9] *Mathematica* programs for this book are available from `mathsource.wri.com`

[Gear] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1971.

[GMP] A. Gray, M. Pinsky and M. Mezzino, *Introduction to Ordinary Differential Equations with Mathematica*, TELOS/Springer-Verlag, New York, NY, 1997.

[Ham] R. W. Hamming, *Numerical Methods for Scientists and Engineers*, Second Edition, McGraw-Hill, New York, 1973. Reprint by Dover Publications, New York, 1986.

[Har] D. R. Hartree, *Numerical Analysis*, Second Edition, Clarendon Press, Oxford, 1958.

[Hur] W. Hurewicz, *Lectures on Ordinary Differential Equations*, M. I. T. Press, Cambridge, MA, 1958.

[Ince] E. L. Ince, *Integration of Ordinary Differential Equations*, Seventh Edition, Oliver and Boyd, New York, 1967.

[Iserles] Arieh Iserles, *A First Course in the Numerical Analysis of Differential equations* Cambridge University Press, Cambridge, 1996.

[Kamke] E. Kamke, *Differentialgleichungen, Lösungenmethoden und Lösungen*, Tenth Edition, B. G. Teubner, Stuttgart, 1983.

[KeWi] J. B. Keiper and D. Withoff, *Numerical Computation in Mathematica*, Course Notes from the 1992 *Mathematica* Conference, Wolfram Research, Champaign, IL, 1992.

[LaSe] L. Lapidus and J. H. Seinfeld, *Numerical Solution of Ordinary Differential Equations*, Academic Press, New York, 1971.

[LaLe] J. LaSalle and S. Lefschetz, *Stability by Liapunov's Direct Method*, Academic Press, New York, 1961.

[Milne] W. E. Milne, *Numerical Solutions of Differential Equations*, John Wiley and Sons, New York, 1970. Reprint by Dover Publications, New York, 1970.

[PFTV] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, Cambridge, 1988.

[Rass] J. M. Rassias, *Counter Examples in Differential Equations and Related Topics*, World Scientific, Singapore 1991.

[Sim] G. F. Simmons, *Differential Equations, with Applications and Historical Notes*, Second Edition, McGraw-Hill, New York, 1991.

[SkKe] R. K. Skeel and J. B. Keiper, *Elementary Numerical Computing with Mathematica*, McGraw-Hill, New York, 1993.

[StpBul] J. Stoer and R. Bulirsch *Introduction to Numerical Analysis* Springer-Verlag, Berlin-New York, 1980.

[Verh] F. Verhulst, *Nonlinear Differential Equations and Dynamical Systems*, McGraw-Hill, New York, 1972.

[Vess] E. Vessiot, "Sur l'intégration des equations differentielles lineares", *Annales Scientifiques de l'Ecole Normale Supérieure*, 3e série *9* (1892), 197–280.

[Wm2] S. Wolfram, The *Mathematica* Book, Third Edition, Wolfram Media, Champaign, Il, 1997.

[WyBa] C. R. Wylie and L. C. Barrett, *Advanced Engineering Mathematics*, Sixth Edition, McGraw-Hill, New York, 1995.

[Zwill] D. Zwillinger, *Handbook of Differential Equations*, Second Edition, Academic Press, San Diego, CA, 1992.