CHAPTER 6 Differential Equations

Aims and objectives

Many practical problems in science and engineering are appropriately modelled by *differential equations*. The objective of this chapter is to introduce the fundamental methods for their numerical solution. The methods vary with the nature of the equation to be solved. Simple initial-value problems lead to methods based on using local approximations to try to 'follow' the solution curve. These form the basis of methods used for more difficult situations. For boundary-value problems, *shooting methods* use the principle of bracketing a target, and then refining the initial conditions until the target is achieved. In the case of linear boundary-value problems, numerical differentiation formulas can be used to generate a system of linear equations for values of the solution function. This last idea provides a natural lead into Chapter 7.

6.1 Introduction and Euler's method

This chapter is concerned with the basic ideas behind the numerical solution of differential equations. We shall concentrate largely on the solution of first-order *initial-value problems*. These have the basic form

$$y' = f(x,y); \qquad y(x_0) = y_0 \tag{6.1}$$

Later in the chapter we will consider briefly both higher-order initial-value problems (or systems of them) and the solution of *boundary-value problems* where the conditions are specified at 2 distinct points. A typical second-order 2-point boundary-value problem has the form

$$y'' = f(x,y,y'); \qquad y(a) = y_a, y(b) = y_b \tag{6.2}$$

Most of the methods we shall consider for initial-value problems are based on a Taylor series approximation to

$$y(x_1) = y(x_0 + h_0) \tag{6.3}$$

for some steplength h_0. The process can then be repeated for subsequent steps. An approximate value $y_1 \approx y(x_1)$ is used with a steplength h_1 to obtain an approximation $y_2 \approx y(x_2)$, and so on throughout the domain of interest. Typically, there is more than one possible derivation and explanation of the methods. Taylor series provide one. Many of the methods can also be viewed as applications of simple numerical integration rules to the integration of the function $f(x, y(x))$. The simplest method – from either of these viewpoints – is *Euler's method* for which there is also a simple graphical explanation. It is with Euler's method that we begin our study.

First, we consider Euler's method as a graphical technique. Figure 6.1 shows a slope field for the differential equation

$$y' = 3x^2y \tag{6.4}$$

which, with the initial condition $y(0) = 1$, we shall use as a basic example for much of this chapter. This particular initial-value problem

$$y' = 3x^2y; \qquad y(0) = 1 \tag{6.5}$$

is easily solved by standard techniques. However that allows us to compare our computed solution with the true one to analyze the errors.

The *slope field* (or direction field) consists of short line segments whose slopes are the slopes of solutions to (6.4) passing through those points. These slopes are computed by evaluating the right-hand side of the differential equation at a grid of points (x, y). The line segments are therefore tangent lines to solutions of the differential equation at the various points.

Using the slope field it is also easy to see the basic shape of the solution through any particular point. Try tracing a curve starting at (0, 1) in Figure 6.1 following the

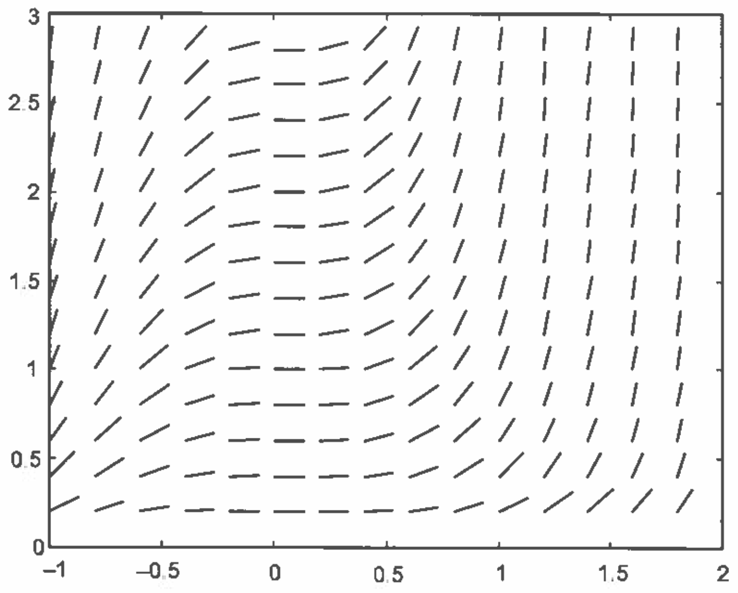


Figure 6.1 Slope field for (6.4)

slopes indicated and you will quickly obtain a good sketch of the solution to the initial-value problem (6.5).

By following a tangent line for a short distance, we obtain an approximation to the solution at a nearby point. The tangent line at that point can then be used for the next step in the solution process. This is the essence of Euler's method.

We next derive Euler's method algebraically from a Taylor approximation. For this purpose, we shall fix the steplength h. Let $x_0 = 0$ and $y_0 = 1$ as in (6.5). Also denote $x_0 + kh$ by x_k for $k = 0, 1, \ldots$ The graphical process just described is then equivalent to using a first-order Taylor approximation to $y(x_1)$

$$y(x_1) = y(x_0 + h) \approx y_0 + hy'(x_0) = y_0 + hf(x_0, y_0)$$

We denote this approximation by y_1 so that

$$y_1 = y_0 + hf(x_0, y_0) \tag{6.6}$$

is the basic step of Euler's method.

Continuing in this manner, we obtain the general Euler step:

$$y_{k+1} = y_k + hf(x_k, y_k) \tag{6.7}$$

Note 1: Apart from y_0, we use y_k to represent our approximation to the solution value $y(x_k)$. This is in contrast to our notation for interpolation and integration where we often used $f_k = f(x_k)$.

Note 2: (6.7) is not quite the generalization of (6.6) that we would like. In the first step (6.6), we have exact values to use on the right-hand side. In subsequent steps, the exact values are not available and so subsequent steps are based on approximate values. This can result in a substantial build-up of the error in our approximate solution. This build-up of error is illustrated in Figure 6.2.

The successive approximate solution points are generated by following the tangents to the different solution curves through these points. As is obvious from Figure 6.2, the approximate solution is lagging ever further behind the true solution. At each step the approximate solution follows the tangent line for $h = 0.25$. At the next step, it follows the tangent to the solution to the original differential equation that passes through this *erroneous* point. Although all the solution curves have similar behavior, those below the desired solution are all less steep at corresponding x-values and the convex nature of the curves implies that the tangent will lie below the curve resulting in further growth in the error.

We shall return to the question of the error analysis of Euler's, and other, methods shortly.

It was stated earlier that many of our methods can also be described in terms of the approximate integration of the right-hand side of the differential equation (6.1). To see this, we again consider the first step of the solution. By the fundamental theorem of calculus, we have

$$y(x_1) - y(x_0) = \int_{x_0}^{x_1} f(x, y(x))dx \tag{6.8}$$

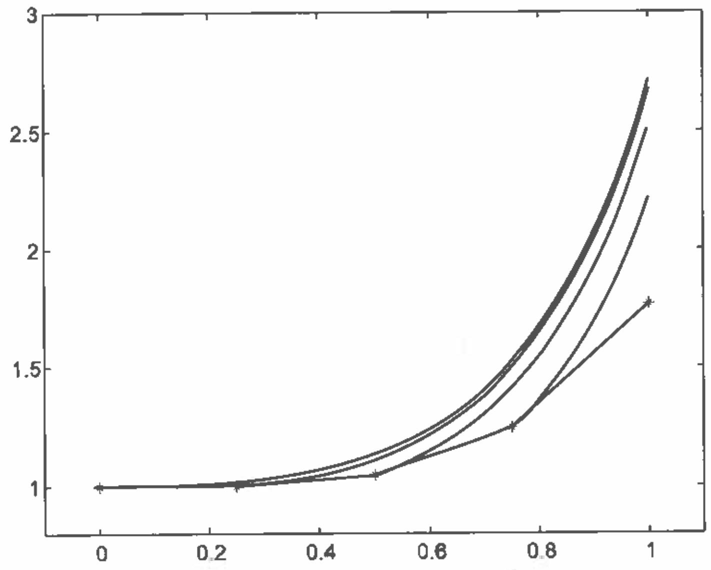


Figure 6.2 Build-up of error in Euler's method

Approximating this integral with the left-hand end-point rule, we obtain

$$y(x_1) - y(x_0) \approx (x_1 - x_0)f(x_0, y(x_0)) = hf(x_0, y(x_0))$$

which is equivalent to the approximation (6.6) derived above for Euler's method.

This suggests that alternative techniques could be based on better numerical integration rules, such as the midpoint rule or Simpson's rule. This is not completely straightforward since such formulas would require knowledge of the solution at points where it is not yet known. We return to this topic later, also.

Euler's method is easily implemented in MATLAB. The inputs required are the function f, the interval $[a, b]$ over which the solution is sought, the initial value $y_0 = y(a)$, and the number of steps to be used. The following code can be used.

Program

MATLAB implementation of Euler's method

```
function sol=euler1(fcn,a,b,y0,N)
%Generates Euler solution to y'=f(x,y) using N steps
%Initial condition is y(a)=y0
h=(b-a)/N;
x=a+(0:N)*h;
y(1)=y0;
for k=1:N
  y(k+1)=y(k)+h*feval(fcn,x(k),y(k));
end
sol=[x',y'];
```

The output here consists of a table of values x_k, y_k.

Note: The m-file here is called euler1.m rather than just 'euler' because there is a built-in MATLAB m-file with that name which is used for a slightly different purpose.

Example 1 Apply Euler's method to the solution of the initial-value problem (6.5) $y' = 3x^2y$; $y(0) = 1$. Begin with $N = 4$ steps and repeatedly double this number of steps up to 128

Applying the program above with $N = 4$, we use the command

```
» s4=euler1('testde',0,1,1,4)
```

to get the results

x	y
0	1.0000
0.25	1.0000
0.5	1.0469
0.75	1.2432
1.0	1.7676

We illustrate Euler's method by reproducing these results without the aid of the computer.

With $x_0 = 0$, $y_0 = 1$, $N = 4$, and $h = 1/4$, we obtain

$$y_1 = y_0 + \frac{1}{4}3x_0^2y_0 = 1$$

Then with $x_1 = 1/4, y_1 = 1$:

$$y_2 = y_1 + \frac{1}{4}3x_1^2y_1 = 1 + \frac{3}{4}\left(\frac{1}{4}\right)^2 = 1.046875$$

and, subsequently,

$$y_3 = y_2 + \frac{1}{4}3x_2^2y_2 = 1.046875 + \frac{3}{4}\left(\frac{1}{2}\right)^2 1.046875 = 1.243164$$

$$y_4 = y_3 + \frac{1}{4}3x_3^2y_3 = 1.243164 + \frac{3}{4}\left(\frac{3}{4}\right)^2 1.243164 = 1.767624$$

This is also the solution plotted in Figure 6.2. Similar tables, with more entries, are generated for the other values of N. The values for $y(1)$ obtained, and their errors, are tabulated below. (**Note**: The true solution of (6.5) is $y = \exp(x^3)$ so that $y(1) = e$.)

N	4	8	16	32	64	128
y_N	1.7676	2.1181	2.3726	2.5312	2.6207	2.6684
$\|Error\|$	0.9507	0.6002	0.3456	0.1870	0.0975	0.0498

We see that the errors are steadily, if slowly being reduced. The first few solutions are plotted, along with the true solution, in Figure 6.3.

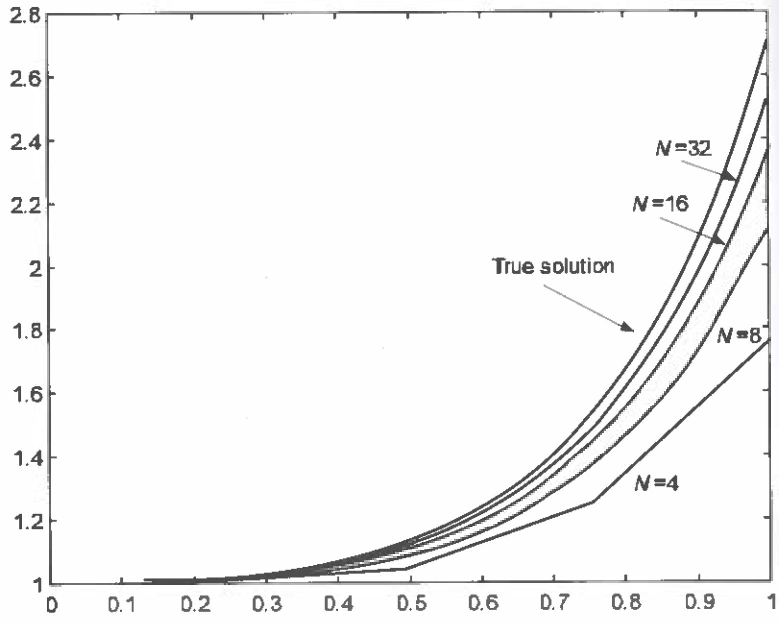


Figure 6.3 Euler's method for (6.5)

The ratios of the successive final errors in Example 1 are approximately 0.63, 0.58, 0.54, 0.52 and 0.51 which seem to be settling down close to 1/2 which is, of course, the same factor by which the steplength h is reduced. This suggests that the overall error in Euler's method is of order h; that is, the error is $O(h)$.

We turn now to the analysis of this truncation error. As we have already observed, there are 2 components to this error. There is a contribution due to the straight line approximation used at each step. There is also a significant contribution resulting from the earlier errors. Their effect is that the linear approximation is not tangent to the required solution but to another solution of the differential equation passing through the current point. These 2 contributions are called the *local* and *global* truncation errors.

The situation is illustrated in Figure 6.4 for step 2 of Euler's method.

In this case, the local contribution to the error at step 2 is relatively small, with a much larger contribution coming from the effect of the (local and global) error in step 1.

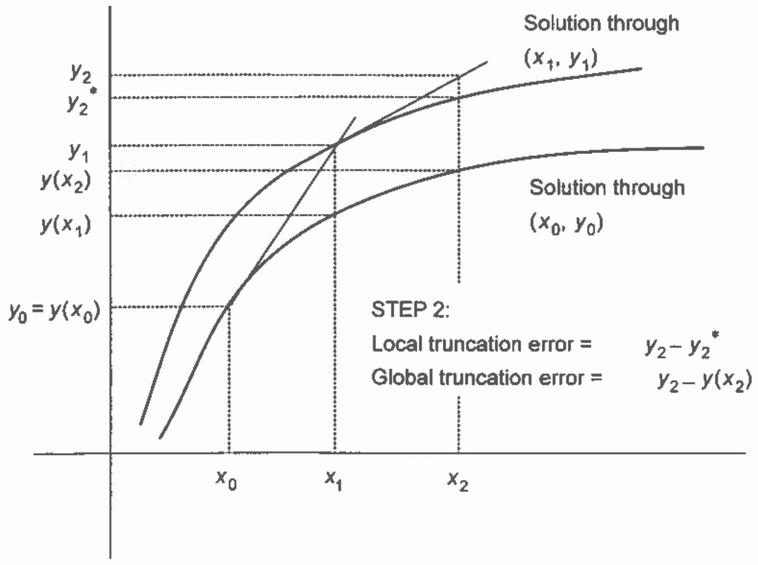

Solution through (x_1, y_1)
Solution through (x_0, y_0)
y_2
y_2^*
y_1
$y(x_2)$
$y(x_1)$
$y_0 = y(x_0)$
STEP 2:
Local truncation error = $y_2 - y_2^*$
Global truncation error = $y_2 - y(x_2)$
x_0
x_1
x_2

Figure 6.4 Global and local truncation error

For step 1, we have

$$y_1 = y_0 + hf(x_0, y_0)$$

while Taylor's theorem gives us

$$y(x_1) = y_0 + hy_0' + \frac{h^2}{2}y''(\xi) = y_0 + hf(x_0, y_0) + \frac{h^2}{2}y''(\xi)$$

It follows that the local truncation error in the approximation is

$$|y_1 - y(x_1)| = \frac{h^2}{2}|y''(\xi)| \leq Mh^2$$

where M is a bound on the second derivative of the solution. A similar local truncation error will occur at each step of the solution process.

We see that the local truncation error for Euler's method is $O(h^2)$ yet we observed that the global truncation error appears to be just $O(h)$.

By the time we have computed our estimate y_N of $y(b)$, we have committed N of these truncation errors from which we may conclude that the global truncation error has the form

$$E = NMh^2 = M(b-a)h$$

since $Nh = b - a$. Thus the global truncation error is indeed of order h.

The 'derivation' above is not rigorous, but it does lead to the correct result in a reasonably intuitive manner. The stated error is the leading contribution to the rigorously defined global truncation error which is of order h. (The rigorous derivation is more complicated than is appropriate here.)

In this error analysis, we have completely ignored the effect of roundoff errors. These too will tend to accumulate and their effect can be analyzed in a manner similar to that for the global truncation error. The result is that the global roundoff error *bound* is *inversely* proportional to the steplength h. As with numerical differentiation in Chapter 5, this places a restriction on the accuracy that can be achieved. However, for systems such as MATLAB, working with IEEE double-precision arithmetic, the effect of roundoff error is typically *much* smaller than the global truncation error, unless we are seeking very high accuracy and therefore using very small steplengths.

Clearly, Euler's method is not itself going to be sufficient to generate accurate solutions to differential equations. It does, however, provide the basis for much of what follows in the next two sections.

Exercises: Section 6.1

Exercises 1–6 and 8 concern the differential equation

$$y' = x/y$$

1 Find the general solution of the differential equation.

2 For the initial condition $y(0) = 3$, use Euler's method with steps $h = 1, 1/2$ and $1/4$ to approximate $y(1)$.

3 Use Euler's method to solve the initial-value problem of Exercise 2 over $[0,4]$ with $N = 10, 20, 50, 100, 200$ steps.

4 Tabulate the errors in the approximate values of $y(4)$ in Exercise 3. Verify that their errors appear to be $O(h)$.

5 Using negative steplengths solve the initial-value problem in Exercise 2 over $[-4,0]$ to obtain a graph of its solution over $[-4,4]$.

6 Repeat Exercise 5 for different initial conditions: $y(0) = 1, 2, 3, 4, 5$ and plot the solutions on the same axes.

7 Use Euler's method with steplengths $h = 10^{-k}$ for $k = 1, 2, 3$ to solve the initial-value problem $y' = x + y^2$ with $y(0) = 0$ on $[0,1]$. Tabulate the results for $x = 0, 0.1, 0.2, \ldots, 1$ and graph the solutions.

8 (More challenging) Use Euler's method to solve the initial-value problems with initial conditions $y(k) = 0$ over the interval $[k, 10]$ for $k = 1, 2, 3, 4, 5$. Plot these solutions on the same axes. (**Note** The slope of the solution is infinite at $x = k$. The differential equation can be rewritten in the form $dx/dy = y/x$ for these cases. This can be solved in the same way as before – except that we do not know the y-interval over which we must compute the solution.)

6.2 Runge–Kutta methods

The basic idea of the *Runge–Kutta methods* is to use additional points between x_k and x_{k+1} which allow the resulting approximation to agree with more terms of the

Taylor series. Another interpretation of this is that we are able to use higher-order numerical integration methods to approximate

$$y(x_{k+1}) - y(x_k) = \int_{x_k}^{x_{k+1}} f(x,y)dx$$

The general derivation of Runge–Kutta methods is somewhat complicated. We shall illustrate the process with examples of second-order Runge–Kutta methods which are derived from a second-order Taylor series expansion. Higher-order, especially fourth-order, Runge–Kutta methods are commonly used in practice. The basic ideas behind these are similar to those presented. We shall discuss the most commonly used fourth-order Runge–Kutta method in some detail without deriving it in detail.

Consider the second-order Taylor expansion about the point (x_0, y_0) using a steplength h:

$$y(x_1) \approx y_0 + hy_0' + \frac{h^2}{2}y_0''$$

from which we obtain the approximate value

$$y_1 = y_0 + hf(x_0, y_0) + \frac{h^2}{2}y_0'' \tag{6.9}$$

This formula has error of order $O(h^3)$. However, we need an approximation for y_0''. Since this term is multiplied by h^2, it follows that an approximation to y_0'' which is itself accurate to order $O(h)$ will preserve the overall error in (6.9) at $O(h^3)$.

It will be helpful to introduce a little notation here. Denote the slope $y_0' = f(x_0, y_0)$ by k_1. Also let $\alpha \in [0, 1]$ and consider an 'Euler step' of length αh. We get

$$y(x_0 + \alpha h) \approx y_0 + \alpha hk_1$$

and denote by k_2 the slope at the point $(x_0 + \alpha h, y_0 + \alpha hk_1)$ so that $k_2 = f(x_0 + \alpha h, y_0 + \alpha hk_1)$. The simplest divided difference estimate of y_0'' is then given by

$$\begin{aligned} y_0'' &\approx \frac{y'(x_0 + \alpha h) - y'(x_0)}{\alpha h} \\ &\approx \frac{hf(x_0 + \alpha h, y_0 + \alpha hk_1) - f(x_0, y_0)}{\alpha h} \\ &= \frac{k_2 - k_1}{\alpha h} \end{aligned} \tag{6.10}$$

Substituting this approximation into (6.9), we obtain

$$\begin{aligned} y_1 &= y_0 + hk_1 + \frac{h^2}{2}\frac{k_2 - k_1}{\alpha h} \\ &= y_0 + h\left[k_1\left(1 - \frac{1}{2\alpha}\right) + \frac{k_2}{2\alpha}\right] \end{aligned} \tag{6.11}$$

which is the general form of the Runge–Kutta second-order formulas. The corresponding formula for the n-th step in the solution process is

$$y_{n+1} = y_n + h\left[k_1\left(1 - \frac{1}{2\alpha}\right) + \frac{k_2}{2\alpha}\right] \tag{6.12}$$

where now

$$\begin{aligned} k_1 &= f(x_n, y_n) \\ k_2 &= f(x_n + \alpha h, y_n + \alpha h k_1) \end{aligned} \tag{6.13}$$

Because the approximation (6.10) has error of order $O(h)$, it follows that the local truncation error in (6.11) is $O(h^3)$ as desired. We shall return to the error analysis of these methods shortly.

There are three important special cases of (6.12) which correspond to choosing $\alpha = 1/2$, 1 or 2/3.

$\alpha = 1/2$: **corrected Euler, or midpoint, method**

$$y_{n+1} = y_n + hk_2 \tag{6.14}$$

where $k_2 = f(x_n + h/2, y_n + hk_1/2)$. This method is essentially the application of the midpoint rule for integration of $f(x, y)$ over the current step with the (approximate) value at the midpoint being obtained by a preliminary (half-) Euler step.

$\alpha = 1$: **modified Euler method**

$$y_{n+1} = y_n + \frac{h}{2}(k_1 + k_2) \tag{6.15}$$

with $k_2 = f(x_n + h, y_n + hk_1)$. This method corresponds to using the trapezoid rule to estimate the integral where a preliminary (full) Euler step is taken to obtain the (approximate) value at x_{n+1}.

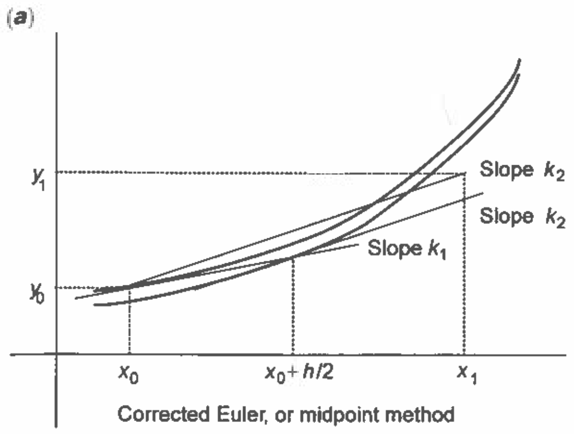
$\alpha = 2/3$: **Heun's method**

$$y_{n+1} = y_n + \frac{h}{4}(k_1 + 3k_2) \tag{6.16}$$

with $k_2 = f(x_n + 2h/3, y_n + 2hk_1/3)$. Heun's method does not correspond to any of the standard numerical integration formulas.

The corrected Euler, or midpoint, and modified Euler methods are illustrated in Figure 6.5.

In each case of Figure 6.5, the second curve represents the solution to the differential equation passing through the appropriate point. For the corrected Euler this 'midpoint' slope k_2 is then applied for the full step from x_0 to x_1. For the modified Euler method, the slope used is the average of the slope k_1 at x_0 and the (estimated) slope at x_1, k_2. It is clear that both these methods result in significantly improved estimates of the average slope of the true solution over $[x_0, x_1]$ than Euler's method, which just uses k_1.



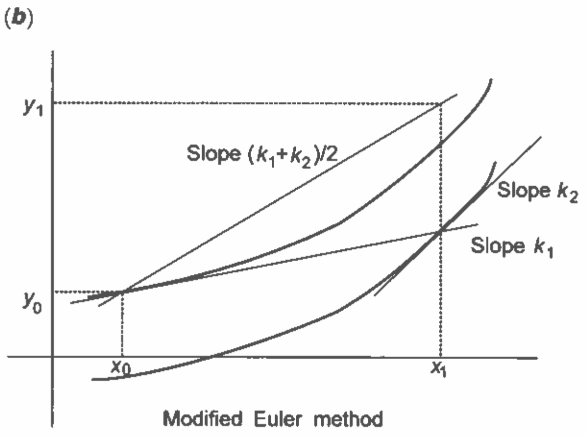


Figure 6.5 Second-order Runge–Kutta methods

We have already observed that the local truncation error for all these 'second-order' Runge–Kutta methods is $O(h^3)$. So, why are they called 'second-order' methods? In just the same way as for Euler's method, by the time we have computed our approximate value of $y(b) = y(a + Nh)$, we have committed N of these local truncation errors. The effect is that the global truncation error is of order $O(Nh^3)$ which is $O(h^2)$ since $Nh = b - a$ is constant. (As for Euler's method, this is not a rigorous argument, but it explains the situation adequately for our present purpose.)

It follows that reducing the steplength by a factor of 1/2 should now result in approximately a 75% improvement in the final answer. That is the error should be reduced by a factor of (about) 1/4. We shall verify this with examples.

Example 2 Use the second-order Runge–Kutta methods (6.14), (6.15) and (6.16) with $N = 2$ to solve the initial-value problem $y' = 3x^2y; y(0) = 1$ on $[0, 1]$

For the Corrected Euler (midpoint) (6.14) method we obtain the results below. The true solution is tabulated for comparison:

k	*Corrected Euler*	*True solution value*
0	$k_1 = f(0, 1) = 0$ $k_2 = f(1/4, 1) = 3/16$ $y_1 = 1 + (1/2)(3/16) = 1.09375$	$\exp(1/8) = 1.133148$
1	$k_1 = f(1/2, 1.09375) = 0.8203125$ $k_2 = f(3/4, 1.298828) = 2.191772$ $y_1 = 1.09375 + (1/2)(2.191772) = 2.189636$	$\exp(1) = 2.718282$

The other two methods yield the results tabulated here in somewhat less detail:

k	*Modified Euler*	*Heun's method*
0	$k_1 = f(0,1) = 0$ $k_2 = f(1/2,1) = 3/4$ $y_1 = 1 + (1/4)(0 + 3/4) = 1.1875$	$k_1 = 0$ $k_2 = 1/3$ $y_1 = 1.125$
1	$k_1 = f(1/2, 1.1875) = 0.890625$ $k_2 = f(1, 1.632813) = 4.898439$ $y_1 = 2.634766$	$k_1 = 0.84375$ $k_2 = 2.929688$ $y_1 = 2.329102$

All of these methods are easily programmed. (See the Exercises.) They can be applied to this same example with more steps.

Example 3 Apply the three Runge–Kutta methods (6.14), (6.15) and (6.16) to the same initial-value problem with $N = 10$ and $N = 100$. Compare their errors and verify that the global truncation error is $O(h^2)$

For $N = 10$, the following MATLAB commands produce the results tabulated below.

```
» sc=correuler('testde',0,1,1,10);
» sm=modeuler('testde',0,1,1,10);
» sh=heun('testde',0,1,1,10);
```

where the m-files implementing the various methods have fairly obvious names. (These are not built-in MATLAB functions.)

x	*Corrected*	*Modified*	*Heun*
0	1.0000	1.0000	1.0000
0.1	1.0008	1.0015	1.0010
0.2	1.0075	1.0090	1.0080
0.3	1.0265	1.0289	1.0273
0.4	1.0648	1.0681	1.0659
0.5	1.1310	1.1357	1.1326
0.6	1.2375	1.2442	1.2397
0.7	1.4028	1.4128	1.4061
0.8	1.6569	1.6722	1.6619
0.9	2.0505	2.0749	2.0585
1.0	2.6732	2.7138	2.6865

For this particular example, the modified Euler method appears to be performing somewhat better than the other two. The errors at $x = 1$ are

Corrected	*Modified*	*Heun*
−0.0451	−0.0045	−0.0318

For comparison, the error in Euler's method for $N = 10$ is −0.5069, so that we see a substantial improvement resulting from using the second-order methods.

For $N = 100$, we of course do not tabulate all the results. The errors at $x = 1$ are now

Corrected	*Modified*	*Heun*
−0.00054	−0.00004	−0.00037

while that for Euler's method is −0.0634.

With the increase in N by a factor of 10 we should expect the error in the first-order Euler's method to be reduced by approximately this same factor. The errors in the second-order methods should be reduced by a factor of about 10^2. It is apparent that *approximately* this level of improvement has indeed been achieved.

For the particular equation in Example 3, it appears that the modified Euler method (which uses an approximate trapezoid rule to integrate the slope) is performing best. This should not be taken as an indicator of the relative merits of these methods in general. In this particular case, it is probably due to the fact that the function is sharply convex (concave up), so that using slope estimates from further to the right is likely to be beneficial.

One of the most widely used Runge–Kutta methods is the classical fourth-order formula, RK4

$$y_{n+1} = y_n + \frac{h}{6}[k_1 + 2(k_2 + k_3) + k_4] \tag{6.17}$$

where the various slope estimates are

$$\begin{aligned} k_1 &= f(x_n, y_n) \\ k_2 &= f(x_n + h/2, y_n + hk_1/2) \\ k_3 &= f(x_n + h/2, y_n + hk_2/2) \\ k_4 &= f(x_n + h, y_n + hk_3) \end{aligned}$$

The derivation of this result, and of the fact that its global truncation error is $O(h^4)$, are omitted here. The basic idea is again that of obtaining approximations which agree with more terms of the Taylor expansion.

In keeping with the earlier methods we can also view (6.17) as an approximate application of Simpson's rule to the integration of the slope over $[x_n, x_{n+1}]$. Here k_1 is the slope at x_n, k_2 and k_3 are both estimated slopes at the midpoint $x_n + h/2$, and k_4 is then an estimate of this slope at $x_n + h = x_{n+1}$. With this interpretation, (6.17) is equivalent to Simpson's rule where the value at the midpoint is replaced by the average of the 2 estimates. This interpretation of the classical RK4 formula also lends some credence to the claim that it *is* a fourth-order method. We already know that Simpson's rule has an error of order $O(h^4)$, and we have already seen that the second-order Runge–Kutta methods (6.14) and (6.15) have errors of the same order as their corresponding integration rules.

Program

MATLAB code for the classical Runge–Kutta fourth-order method RK4 (6.17)

```
function sol=RK4(fcn,a,b,y0,N)
%Generates RK4 solution to y'=f(x,y) using N steps
%Initial condition is y(a)=y0
h=(b-a)/N;
x=a+(0:N)*h;
y(1)=y0;
for k=1:N
  k1=feval(fcn,x(k),y(k));
  k2=feval(fcn,x(k)+h/2,y(k)+h*k1/2);
  k3=feval(fcn,x(k)+h/2,y(k)+h*k2/2);
  k4=feval(fcn,x(k)+h,y(k)+h*k3);
  y(k+1)=y(k)+h*(k1+2*(k2+k3)+k4)/6;
end
sol=[x',y'];
```

Example 4 Apply RK4 to our usual example with both $N = 10$ and $N = 100$

Using $N = 10$ the command

```
» rk=rk4('testde',0,1,1,10)
```

produces the results tabulated.

x	*RK4 solution*	*True solution*
0	1.0000	1.0000
0.1	1.0010	1.0010
0.2	1.0080	1.0080
0.3	1.0274	1.0274
0.4	1.0661	1.0661
0.5	1.1331	1.1331