16.1 Runge-Kutta Method

The formula for the Euler method is

\[ y_{n+1} = y_n + hf(x_n, y_n) \]  

(16.1.1)

which advances a solution from \( x_n \) to \( x_{n+1} \equiv x_n + h \). The formula is unsymmetrical: It advances the solution through an interval \( h \), but uses derivative information only at the beginning of that interval (see Figure 16.1.1). That means (and you can verify by expansion in power series) that the step's error is only one power of \( h \) smaller than the correction, i.e \( O(h^2) \) added to (16.1.1).

There are several reasons that Euler’s method is not recommended for practical use, among them, (i) the method is not very accurate when compared to other, fancier, methods run at the equivalent stepsize, and (ii) neither is it very stable (see §16.6 below).

Consider, however, the use of a step like (16.1.1) to take a “trial” step to the midpoint of the interval. Then use the value of both \( x \) and \( y \) at that midpoint to compute the “real” step across the whole interval. Figure 16.1.2 illustrates the idea. In equations,

\[
\begin{align*}
  k_1 &= hf(x_n, y_n) \\
  k_2 &= hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1\right) \\
  y_{n+1} &= y_n + k_2 + O(h^3)
\end{align*}
\]  

(16.1.2)

As indicated in the error term, this symmetrization cancels out the first-order error term, making the method second order. [A method is conventionally called \( n \)th order if its error term is \( O(h^{n+1}) \).] In fact, (16.1.2) is called the second-order Runge-Kutta or midpoint method.

We needn’t stop there. There are many ways to evaluate the right-hand side \( f(x, y) \) that all agree to first order, but that have different coefficients of higher-order error terms. Adding up the right combination of these, we can eliminate the error terms order by order. That is the basic idea of the Runge-Kutta method. Abramowitz and Stegun [1], and Gear [2], give various specific formulas that derive from this basic
16.1 Runge-Kutta Method

16.1.1 Runge-Kutta Method

The Runge-Kutta method is a powerful tool for solving ordinary differential equations. The method is based on the idea of approximating the solution of the differential equation by a polynomial of degree four or higher.

The Runge-Kutta method is defined by the following iterative formula:

\[
y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\]

where:

- \(y_n\) is the value of the solution at the start of the interval.
- \(y_{n+1}\) is the value of the solution at the end of the interval.
- \(h\) is the step size.
- \(k_1, k_2, k_3, k_4\) are intermediate values calculated using the differential equation.

The Runge-Kutta method is a widely used method for solving differential equations due to its accuracy and efficiency. It is particularly useful for problems where the solution is not known explicitly.

The Runge-Kutta method is a powerful tool for solving ordinary differential equations. The method is based on the idea of approximating the solution of the differential equation by a polynomial of degree four or higher.

The Runge-Kutta method is defined by the following iterative formula:

\[
y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\]

where:

- \(y_n\) is the value of the solution at the start of the interval.
- \(y_{n+1}\) is the value of the solution at the end of the interval.
- \(h\) is the step size.
- \(k_1, k_2, k_3, k_4\) are intermediate values calculated using the differential equation.

The Runge-Kutta method is a widely used method for solving differential equations due to its accuracy and efficiency. It is particularly useful for problems where the solution is not known explicitly.

The Runge-Kutta method is a powerful tool for solving ordinary differential equations. The method is based on the idea of approximating the solution of the differential equation by a polynomial of degree four or higher.

The Runge-Kutta method is defined by the following iterative formula:

\[
y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\]

where:

- \(y_n\) is the value of the solution at the start of the interval.
- \(y_{n+1}\) is the value of the solution at the end of the interval.
- \(h\) is the step size.
- \(k_1, k_2, k_3, k_4\) are intermediate values calculated using the differential equation.

The Runge-Kutta method is a widely used method for solving differential equations due to its accuracy and efficiency. It is particularly useful for problems where the solution is not known explicitly.
contemporary practice of science rather than as a statement about strict mathematics. That is, it reflects the nature of the problems that contemporary scientists like to solve.

For many scientific users, fourth-order Runge-Kutta is not just the first word on ODE integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm. Keep in mind, however, that the old workhorse’s last trip may well be to take you to the poorhouse: Bulirsch-Stoer or predictor-corrector methods can be very much more efficient for problems where very high accuracy is a requirement. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields. However, even the old workhorse is more nimble with new horseshoes. In §16.2 we will give a modern implementation of a Runge-Kutta method that is quite competitive as long as very high accuracy is not required. An excellent discussion of the pitfalls in constructing a good Runge-Kutta code is given in [3].

Here is the routine for carrying out one classical Runge-Kutta step on a set of $n$ differential equations. You input the values of the independent variables, and you get out new values which are stepped by a stepsize $h$ (which can be positive or negative). You will notice that the routine requires you to supply not only function derivs for calculating the right-hand side, but also values of the derivatives at the starting point. Why not let the routine call derivs for this first value? The answer will become clear only in the next section, but in brief is this: This call may not be your only one with these starting conditions. You may have taken a previous step with too large a stepsize, and this is your replacement. In that case, you do not want to call derivs unnecessarily at the start. Note that the routine that follows has, therefore, only three calls to derivs.

SUBROUTINE rk4(y,dydx,n,x,h,yout,derivs)
INTEGER n,NMAX
REAL h,x,dydx(n),y(n),yout(n)
EXTERNAL derivs
PARAMETER (NMAX=50)

Set to the maximum number of functions.

Given values for the variables $y(1:n)$ and their derivatives $dydx(1:n)$ known at $x$, use the fourth-order Runge-Kutta method to advance the solution over an interval $h$ and return the incremented variables as $yout(1:n)$, which need not be a distinct array from $y$. The user supplies the subroutine derivs(x,y,dydx), which returns derivatives dydx at x.

INTEGER i
REAL h6,hh,xh,dym(NMAX),dyt(NMAX),yt(NMAX)

hh=h*0.5
h6=h/6.
xh=x+hh

For many scientific users, fourth-order Runge-Kutta is not just the first word on ODE integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm. Keep in mind, however, that the old workhorse’s last trip may well be to take you to the poorhouse: Bulirsch-Stoer or predictor-corrector methods can be very much more efficient for problems where very high accuracy is a requirement. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields. However, even the old workhorse is more nimble with new horseshoes. In §16.2 we will give a modern implementation of a Runge-Kutta method that is quite competitive as long as very high accuracy is not required. An excellent discussion of the pitfalls in constructing a good Runge-Kutta code is given in [3].

Here is the routine for carrying out one classical Runge-Kutta step on a set of $n$ differential equations. You input the values of the independent variables, and you get out new values which are stepped by a stepsize $h$ (which can be positive or negative). You will notice that the routine requires you to supply not only function derivs for calculating the right-hand side, but also values of the derivatives at the starting point. Why not let the routine call derivs for this first value? The answer will become clear only in the next section, but in brief is this: This call may not be your only one with these starting conditions. You may have taken a previous step with too large a stepsize, and this is your replacement. In that case, you do not want to call derivs unnecessarily at the start. Note that the routine that follows has, therefore, only three calls to derivs.

SUBROUTINE rk4(y,dydx,n,x,h,yout,derivs)
INTEGER n,NMAX
REAL h,x,dydx(n),y(n),yout(n)
EXTERNAL derivs
PARAMETER (NMAX=50)

Set to the maximum number of functions.

Given values for the variables $y(1:n)$ and their derivatives $dydx(1:n)$ known at $x$, use the fourth-order Runge-Kutta method to advance the solution over an interval $h$ and return the incremented variables as $yout(1:n)$, which need not be a distinct array from $y$. The user supplies the subroutine derivs(x,y,dydx), which returns derivatives dydx at x.

INTEGER i
REAL h6,hh,xh,dym(NMAX),dyt(NMAX),yt(NMAX)

hh=h*0.5
h6=h/6.
xh=x+hh

SUBROUTINE rk4(y,dydx,n,x,h,yout,derivs)
INTEGER n,NMAX
REAL h,x,dydx(n),y(n),yout(n)
EXTERNAL derivs
PARAMETER (NMAX=50)

Set to the maximum number of functions.

Given values for the variables $y(1:n)$ and their derivatives $dydx(1:n)$ known at $x$, use the fourth-order Runge-Kutta method to advance the solution over an interval $h$ and return the incremented variables as $yout(1:n)$, which need not be a distinct array from $y$. The user supplies the subroutine derivs(x,y,dydx), which returns derivatives dydx at x.

INTEGER i
REAL h6,hh,xh,dym(NMAX),dyt(NMAX),yt(NMAX)

hh=h*0.5
h6=h/6.
xh=x+hh

SUBROUTINE rk4(y,dydx,n,x,h,yout,derivs)
INTEGER n,NMAX
REAL h,x,dydx(n),y(n),yout(n)
EXTERNAL derivs
PARAMETER (NMAX=50)

Set to the maximum number of functions.

Given values for the variables $y(1:n)$ and their derivatives $dydx(1:n)$ known at $x$, use the fourth-order Runge-Kutta method to advance the solution over an interval $h$ and return the incremented variables as $yout(1:n)$, which need not be a distinct array from $y$. The user supplies the subroutine derivs(x,y,dydx), which returns derivatives dydx at x.

INTEGER i
REAL h6,hh,xh,dym(NMAX),dyt(NMAX),yt(NMAX)

hh=h*0.5
h6=h/6.
xh=x+hh
The Runge-Kutta method treats every step in a sequence of steps in identical manner. Prior behavior of a solution is not used in its propagation. This is mathematically proper, since any point along the trajectory of an ordinary differential equation can serve as an initial point. The fact that all steps are treated identically also makes it easy to incorporate Runge-Kutta into relatively simple “driver” schemes.

We consider adaptive stepsizes control, discussed in the next section, an essential for serious computing. Occasionally, however, you just want to tabulate a function at equally spaced intervals, and without particularly high accuracy. In the most common case, you want to produce a graph of the function. Then all you need may be a simple driver program that goes from an initial $x_s$ to a final $x_f$ in a specified number of steps. To check accuracy, double the number of steps, repeat the integration, and compare results. This approach surely does not minimize computer time, and it can fail for problems whose nature requires a variable stepsize, but it may well minimize user effort. On small problems, this may be the paramount consideration.

Here is such a driver, self-explanatory, which tabulates the integrated functions in a common block path.

```fortran
SUBROUTINE rkdumb(vstart,nvar,x1,x2,nstep,derivs)
INTEGER nstep,nvar,NMAX,NSTPMX
PARAMETER (NMAX=50,NSTPMX=200)
REAL x1,x2,vstart(nvar),xx(NSTPMX),y(NMAX,NSTPMX)
EXTERNAL derivs
COMMON /path/ xx,y
C USES rk4

Starting from initial values vstart(1:nvar) known at x1 use fourth-order Runge-Kutta to advance nstep equal increments to x2. The user-supplied subroutine derivs(x,v,dvdx) evaluates derivatives. Results are stored in the common block /path/. Be sure to dimension the common block appropriately.

INTEGER i,k
REAL h,x,dv(NMAX),v(NMAX)
do i=1,nvar
   v(i)=vstart(i)
y(i,1)=v(i)
endo
dx(i)=x1
x=x1
h=(x2-x1)/nstep
do k=1,nstep
call derivs(x,v,dv)
do i=1,nvar
   yt(i)=y(i)+hh*dydx(i)
endo
call derivs(xh,yt,dyt)
do i=1,nvar
   yt(i)=y(i)+hh*dyt(i)
endo
call derivs(xh,yt,dym)
do i=1,nvar
   yt(i)=y(i)+h*dym(i)
dym(i)=dyt(i)+dym(i)
endo
call derivs(x+h,yt,dyt)
do i=1,nvar
   yout(i)=y(i)+h*.6*(dydx(i)+dyt(i)+2.*dym(i))
endo
return
END
```
call rk4(v,dv,nvar,x,h,v,derivs)
if(x+h.eq.x)pause 'stepsize not significant in rkdumb'
x=x+h
xx(k+1)=x
Store intermediate steps.
do 12 i=1,nvar
   \[ y(i,k+1)=v(i) \]
endo 12
13 return
END

CITED REFERENCES AND FURTHER READING:

16.2 Adaptive Stepsize Control for Runge-Kutta

A good ODE integrator should exert some adaptive control over its own progress, making frequent changes in its stepsize. Usually the purpose of this adaptive stepsize control is to achieve some predetermined accuracy in the solution with minimum computational effort. Many small steps should tiptoe through treacherous terrain, while a few great strides should speed through smooth uninteresting countryside. The resulting gains in efficiency are not mere tens of percents or factors of two; they can sometimes be factors of ten, a hundred, or more. Sometimes accuracy may be demanded not directly in the solution itself, but in some related conserved quantity that can be monitored.

Implementation of adaptive stepsize control requires that the stepping algorithm return information about its performance, most important, an estimate of its truncation error. In this section we will learn how such information can be obtained. Obviously, the calculation of this information will add to the computational overhead, but the investment will generally be repaid handsomely.

With fourth-order Runge-Kutta, the most straightforward technique by far is step doubling (see, e.g., [1]). We take each step twice, once as a full step, then, independently, as two half steps (see Figure 16.2.1). How much overhead is this, say in terms of the number of evaluations of the right-hand sides? Each of the three separate Runge-Kutta steps in the procedure requires 4 evaluations, but the single and double sequences share a starting point, so the total is 11. This is to be compared not to 4, but to 8 (the two half-steps), since — stepsizes control aside — we are achieving the accuracy of the smaller (half) stepsize. The overhead cost is therefore a factor 1.375. What does it buy us?