16.1 Runge-Kutta Method

The formula for the Euler method is

\[ y_{n+1} = y_n + h f(x_n, y_n) \]  \hspace{1cm} (16.1.1)

which advances a solution from \( x_n \) to \( x_{n+1} = x_n + h \). 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,

\[ k_1 = h f(x_n, y_n) \]
\[ k_2 = h f \left( x_n + \frac{1}{2} h, y_n + \frac{1}{2} k_1 \right) \]  \hspace{1cm} (16.1.2)
\[ y_{n+1} = y_n + k_2 + O(h^3) \]

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

Figure 16.1.1. Euler’s method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

Figure 16.1.2. Midpoint method. Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open dots represent function values that are discarded once their derivatives have been calculated and used.

The idea. By far the most often used is the classical fourth-order Runge-Kutta formula, which has a certain sleekness of organization about it:

\[
\begin{align*}
  k_1 &= hf(x_n, y_n) \\
  k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\
  k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\
  k_4 &= hf(x_n + h, y_n + k_3) \\
  y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\end{align*}
\]

(16.1.3)

The fourth-order Runge-Kutta method requires four evaluations of the right-hand side per step \( h \) (see Figure 16.1.3). This will be superior to the midpoint method (16.1.2) if at least twice as large a step is possible with (16.1.3) for the same accuracy. Is that so? The answer is: often, perhaps even usually, but surely not always! This takes us back to a central theme, namely that high order does not always mean high accuracy. The statement “fourth-order Runge-Kutta is generally superior to second-order” is a true one, but you should recognize it as a statement about the
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.

```c
#include "nrutil.h"

void rk4(float y[], float dydx[], int n, float x, float h, float yout[],
          void (*derivs)(float, float [], float []))
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 routine derivs(x,y,dydx), which returns derivatives dydx at x.
{
    int i;
    float xb,hh,h6,*dym,*dyt,*yt;

dym=vector(1,n);
dyt=vector(1,n);
```
16.1 Runge-Kutta Method

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 stepsize 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 the global arrays $*x$ and $**y$; be sure to allocate memory for them with the
routines vector() and matrix(), respectively.

```c
#include "nrutil.h"

float **y,*xx;

void rkdumb(float vstart[], int nvar, float x1, float x2, int nstep,
            void (*derivs)(float, float [], float []));
```

Starting from initial values $vstart[1..nvar]$ known at $x_1$ use fourth-order Runge-Kutta
to advance $nstep$ equal increments to $x_2$. The user-supplied routine $derivs(x,v,dvdx)$
evaluates derivatives. Results are stored in the global variables $y[1..nvar][1..nstep+1]$ and
$xx[1..nstep+1]$.

```c
{ void rk4(float y[], float dydx[], int n, float x, float h, float yout[],
            void (*derivs)(float, float [], float []));
    int i,k;
    float x,h;
    float *v,*yout,*dv;

    v=vector(1,nvar);
    yout=vector(1,nvar);
```
dv=vector(1,nvar);
for (i=1;i<=nvar;i++) { Load starting values.
    v[i]=vstart[i];
    y[i][1]=v[i];
}
xx[1]=x1;
x=x1;
h=(x2-x1)/nstep;
for (k=1;k<=nstep;k++) { Take nstep steps.
    (*derivs)(x,v,dv);
    rk4(v,dv,nvar,x,h,yout,derivs);
    if ((float)(x+h)==x) nrerror("Step size too small in routine rkdumb");
    x+=h;
    xx[k+1]=x; Store intermediate steps.
    for (i=1;i<=nvar;i++) {
        v[i]=yout[i];
        y[i][k+1]=v[i];
    }
}
free_vector(dv,1,nvar);
free_vector(vout,1,nvar);
free_vector(v,1,nvar);

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 signal 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,