

CITED REFERENCES AND FURTHER READING:

- Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall).
- Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 5.
- Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 7.
- Lambert, J. 1973, *Computational Methods in Ordinary Differential Equations* (New York: Wiley).
- Lapidus, L., and Seinfeld, J. 1971, *Numerical Solution of Ordinary Differential Equations* (New York: Academic Press).

16.1 Runge-Kutta Method

The formula for the Euler method is

$$y_{n+1} = y_n + hf(x_n, y_n) \quad (16.1.1)$$

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,

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \quad (16.1.2)$$

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

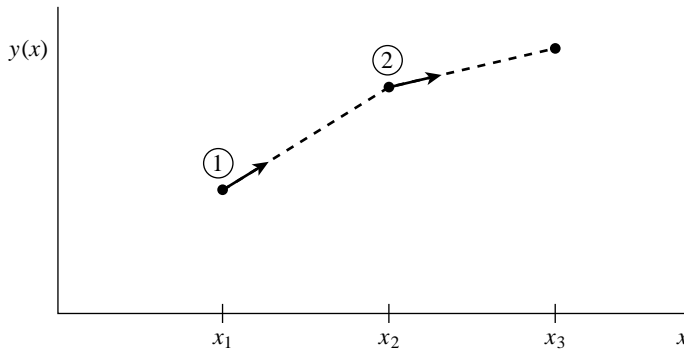


Figure 16.1.1. Euler's method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

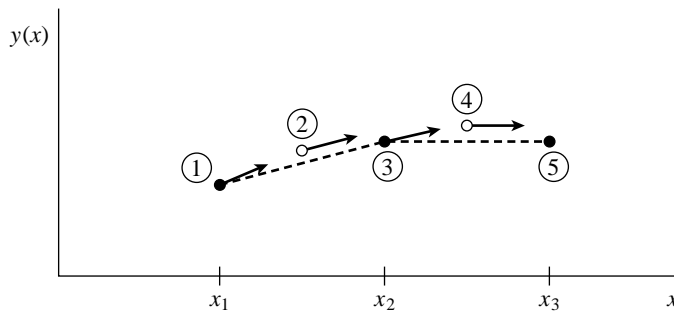


Figure 16.1.2. Midpoint method. Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open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

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

$$\begin{aligned}
 k_1 &= hf(x_n, y_n) \\
 k_2 &= hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \\
 k_3 &= hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \\
 k_4 &= hf(x_n + h, y_n + k_3) \\
 y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
 \end{aligned} \tag{16.1.3}$$

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


```

yt=vector(1,n);
hh=h*0.5;
h6=h/6.0;
xh=x+hh;
for (i=1;i<=n;i++) yt[i]=y[i]+hh*dydx[i];      First step.
(*derivs)(xh,yt,dyt);                          Second step.
for (i=1;i<=n;i++) yt[i]=y[i]+hh*dym[i];
(*derivs)(xh,yt,dym);                          Third step.
for (i=1;i<=n;i++) {
    yt[i]=y[i]+h*dym[i];
    dym[i] += dyt[i];
}
(*derivs)(x+h,yt,dyt);                          Fourth step.
for (i=1;i<=n;i++) Accumulate increments with proper
    yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]); weights.
free_vector(yt,1,n);
free_vector(dyt,1,n);
free_vector(dym,1,n);
}

```

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.

```

#include "nrutil.h"

float **y,**xx;                                For communication back to main.

void rk4(float vstart[], int nvar, float x1, float x2, int nstep,
        void (*derivs)(float, float [], float []))
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 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].
{
    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,*vout,*dv;

    v=vector(1,nvar);
    vout=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,vout,derivs);
    if ((float)(x+h) == x) nrerror("Step size too small in routine rk4");
    x += h;
    xx[k+1]=x;                  Store intermediate steps.
    for (i=1;i<=nvar;i++) {
        v[i]=vout[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:

- Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.5. [1]
- Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 2. [2]
- Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121. [3]
- Rice, J.R. 1983, *Numerical Methods, Software, and Analysis* (New York: McGraw-Hill), §9.2.

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,