Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: Mathematical Association of America), Chapter 5.

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 7.

Hairer, E., Nørsett, S.P., and Wanner, G. 1993, *Solving Ordinary Differential Equations I. Nonstiff Problems*, 2nd ed. (New York: Springer)

Hairer, E., Nørsett, S.P., and Wanner, G. 1996, *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*, 2nd ed. (New York: Springer)

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).

17.1 Runge-Kutta Method

The formula for the Euler method is

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{17.1.1}$$

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 17.1.1). 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 (17.1.1).

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 §17.5 below).

Consider, however, the use of a step like (17.1.1) to take a "trial" step to the midpoint of the interval. Then use the values of both x and y at that midpoint to compute the "real" step across the whole interval. Figure 17.1.2 illustrates the idea. In equations,

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{17.1.2}$$

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 nth order if its error term is $O(h^{n+1})$.] In fact, (17.1.2) 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 idea. By far the most often used is the classical *fourth-order Runge-Kutta formula*,

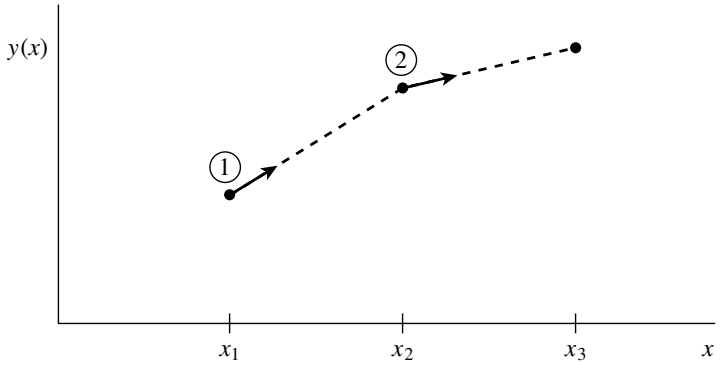

Figure 17.1.1. Euler's method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

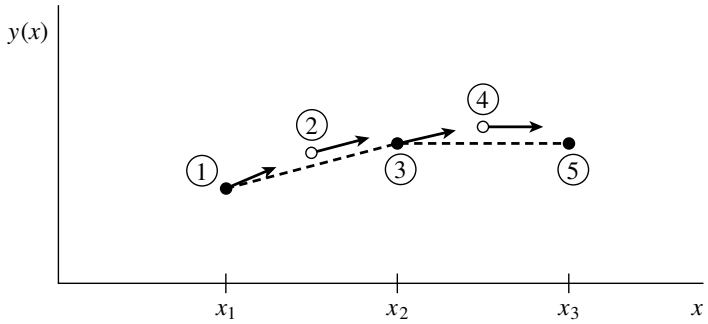

Figure 17.1.2. Midpoint method. Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open dots represent function values that are discarded once their derivatives have been calculated and used.

which has a certain sleekness of organization about it:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1) \\ k_3 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}k_1 + \tfrac{1}{3}k_2 + \tfrac{1}{3}k_3 + \tfrac{1}{6}k_4 + O(h^5) \end{aligned} \tag{17.1.3}$$

The fourth-order Runge-Kutta method requires four evaluations of the right-hand side per step h (see Figure 17.1.3). This will be superior to the midpoint method (17.1.2) *if* at least twice as large a step is possible with (17.1.3) 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 as much a statement about the kind of problems that people solve as a statement about strict mathematics.

For many scientific users, fourth-order Runge-Kutta is not just the first word

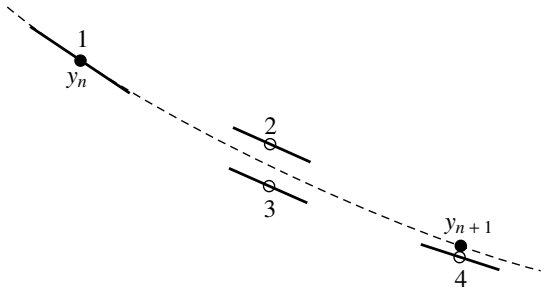

Figure 17.1.3. Fourth-order Runge-Kutta method. In each step the derivative is evaluated four times: once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the final function value (shown as a filled dot) is calculated. (See text for details.)

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: Newer Runge-Kutta methods are *much* more efficient, and Bulirsch-Stoer or predictor-corrector methods can be even 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 §17.2 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 `rk4` for carrying out one classical Runge-Kutta step on a set of `n` differential equations. This routine is completely separate from the various stepper routines introduced in the previous section and given in the rest of the chapter. It is meant for only the most trivial applications. You input the values of the independent variables, and you get out new values that 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`.

rk4.h

```
void rk4(VecDoub_I &y, VecDoub_I &dydx, const Doub x, const Doub h,
    VecDoub_O &yout, void derivs(const Doub, VecDoub_I &, VecDoub_O &))
Given values for the variables y[0..n-1] and their derivatives dydx[0..n-1] 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[0..n-1]. The user supplies the routine derivs(x,y,dydx),
which returns derivatives dydx at x.
{
    Int n=y.size();
    VecDoub dym(n),dyt(n),yt(n);
    Doub hh=h*0.5;
    Doub h6=h/6.0;
    Doub xh=x+hh;
```

```
    for (Int i=0;i<n;i++) yt[i]=y[i]+hh*dydx[i];          First step.
    derivs(xh,yt,dyt);                                    Second step.
    for (Int i=0;i<n;i++) yt[i]=y[i]+hh*dyt[i];
    derivs(xh,yt,dym);                                    Third step.
    for (Int i=0;i<n;i++) {
        yt[i]=y[i]+h*dym[i];
        dym[i] += dyt[i];
    }
    derivs(x+h,yt,dyt);                                   Fourth step.
    for (Int i=0;i<n;i++)                                 Accumulate increments with
        yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);          proper weights.
}
```

The Runge-Kutta method treats every step in a sequence of steps in an 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.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at http://numerical.recipes/aands, §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, "The Art of Writing a Runge-Kutta Code, Part I," in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, "The Art of Writing a Runge-Kutta Code. II," *Applied Mathematics and Computation*, vol. 5, pp. 93–121.[3]

17.2 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, 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,