Chapter 16. Integration of Ordinary Differential Equations

16.0 Introduction

Problems involving ordinary differential equations (ODEs) can always be reduced to the study of sets of first-order differential equations. For example the second-order equation

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x) \tag{16.0.1}$$

can be rewritten as two first-order equations

$$\begin{aligned} \frac{dy}{dx} &= z(x) \\ \frac{dz}{dx} &= r(x) - q(x)z(x) \end{aligned} \tag{16.0.2}$$

where z is a new variable. This exemplifies the procedure for an arbitrary ODE. The usual choice for the new variables is to let them be just derivatives of each other (and of the original variable). Occasionally, it is useful to incorporate into their definition some other factors in the equation, or some powers of the independent variable, for the purpose of mitigating singular behavior that could result in overflows or increased roundoff error. Let common sense be your guide: If you find that the original variables are smooth in a solution, while your auxiliary variables are doing crazy things, then figure out why and choose different auxiliary variables.

The generic problem in ordinary differential equations is thus reduced to the study of a set of N coupled *first-order* differential equations for the functions y_i, $i = 1, 2, \ldots, N$, having the general form

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \ldots, y_N), \qquad i = 1, \ldots, N \tag{16.0.3}$$

where the functions f_i on the right-hand side are known.

A problem involving ODEs is not completely specified by its equations. Even more crucial in determining how to attack the problem numerically is the nature of the problem's boundary conditions. Boundary conditions are algebraic conditions on the values of the functions y_i in (16.0.3). In general they can be satisfied at

discrete specified points, but do not hold between those points, i.e., are not preserved automatically by the differential equations. Boundary conditions can be as simple as requiring that certain variables have certain numerical values, or as complicated as a set of nonlinear algebraic equations among the variables.

Usually, it is the nature of the boundary conditions that determines which numerical methods will be feasible. Boundary conditions divide into two broad categories.

- In *initial value problems* all the y_i are given at some starting value x_s, and it is desired to find the y_i's at some final point x_f, or at some discrete list of points (for example, at tabulated intervals).
- In *two-point boundary value problems*, on the other hand, boundary conditions are specified at more than one x. Typically, some of the conditions will be specified at x_s and the remainder at x_f.

This chapter will consider exclusively the initial value problem, deferring two-point boundary value problems, which are generally more difficult, to Chapter 17.

The underlying idea of any routine for solving the initial value problem is always this: Rewrite the dy's and dx's in (16.0.3) as finite steps Δy and Δx, and multiply the equations by Δx. This gives algebraic formulas for the change in the functions when the independent variable x is "stepped" by one "stepsize" Δx. In the limit of making the stepsize very small, a good approximation to the underlying differential equation is achieved. Literal implementation of this procedure results in *Euler's method* (16.1.1, below), which is, however, *not* recommended for any practical use. Euler's method is conceptually important, however; one way or another, practical methods all come down to this same idea: Add small increments to your functions corresponding to derivatives (right-hand sides of the equations) multiplied by stepsizes.

In this chapter we consider three major types of practical numerical methods for solving initial value problems for ODEs:

- Runge-Kutta methods
- Richardson extrapolation and its particular implementation as the Bulirsch-Stoer method
- predictor-corrector methods.

A brief description of each of these types follows.

1. *Runge-Kutta* methods propagate a solution over an interval by combining the information from several Euler-style steps (each involving one evaluation of the right-hand f's), and then using the information obtained to match a Taylor series expansion up to some higher order.

2. *Richardson extrapolation* uses the powerful idea of extrapolating a computed result to the value that *would* have been obtained if the stepsize had been very much smaller than it actually was. In particular, extrapolation to zero stepsize is the desired goal. The first practical ODE integrator that implemented this idea was developed by Bulirsch and Stoer, and so extrapolation methods are often called Bulirsch-Stoer methods.

3. *Predictor-corrector* methods store the solution along the way, and use those results to extrapolate the solution one step advanced; they then correct the extrapolation using derivative information at the new point. These are best for very smooth functions.

Runge-Kutta is what you use when (i) you don't know any better, or (ii) you have an intransigent problem where Bulirsch-Stoer is failing, or (iii) you have a trivial

problem where computational efficiency is of no concern. Runge-Kutta succeeds virtually always; but it is not usually fastest, except when evaluating f_i is cheap and moderate accuracy ($\lesssim 10^{-5}$) is required. Predictor-corrector methods, since they use past information, are somewhat more difficult to start up, but, for many smooth problems, they are computationally more efficient than Runge-Kutta. In recent years Bulirsch-Stoer has been replacing predictor-corrector in many applications, but it is too soon to say that predictor-corrector is dominated in all cases. However, it appears that only rather sophisticated predictor-corrector routines are competitive. Accordingly, we have chosen *not* to give an implementation of predictor-corrector in this book. We discuss predictor-corrector further in §16.7, so that you can use a canned routine should you encounter a suitable problem. In our experience, the relatively simple Runge-Kutta and Bulirsch-Stoer routines we give are adequate for most problems.

Each of the three types of methods can be organized to monitor internal consistency. This allows numerical errors which are inevitably introduced into the solution to be controlled by automatic, (*adaptive*) changing of the fundamental stepsize. We always recommend that adaptive stepsize control be implemented, and we will do so below.

In general, all three types of methods can be applied to any initial value problem. Each comes with its own set of debits and credits that must be understood before it is used.

We have organized the routines in this chapter into three nested levels. The lowest or "nitty-gritty" level is the piece we call the *algorithm* routine. This implements the basic formulas of the method, starts with dependent variables y_i at x, and calculates new values of the dependent variables at the value $x + h$. The algorithm routine also yields up some information about the quality of the solution after the step. The routine is dumb, however, and it is unable to make any adaptive decision about whether the solution is of acceptable quality or not.

That quality-control decision we encode in a *stepper* routine. The stepper routine calls the algorithm routine. It may reject the result, set a smaller stepsize, and call the algorithm routine again, until compatibility with a predetermined accuracy criterion has been achieved. The stepper's fundamental task is to take the largest stepsize consistent with specified performance. Only when this is accomplished does the true power of an algorithm come to light.

Above the stepper is the *driver* routine, which starts and stops the integration, stores intermediate results, and generally acts as an interface with the user. There is nothing at all canonical about our driver routines. You should consider them to be examples, and you can customize them for your particular application.

Of the routines that follow, `rk4`, `rkck`, `mmid`, `stoerm`, and `simpr` are algorithm routines; `rkqs`, `bsstep`, `stiff`, and `stifbs` are steppers; `rkdumb` and `odeint` are drivers.

Section 16.6 of this chapter treats the subject of *stiff equations*, relevant both to ordinary differential equations and also to partial differential equations (Chapter 19).

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

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

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,

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

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 nth 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

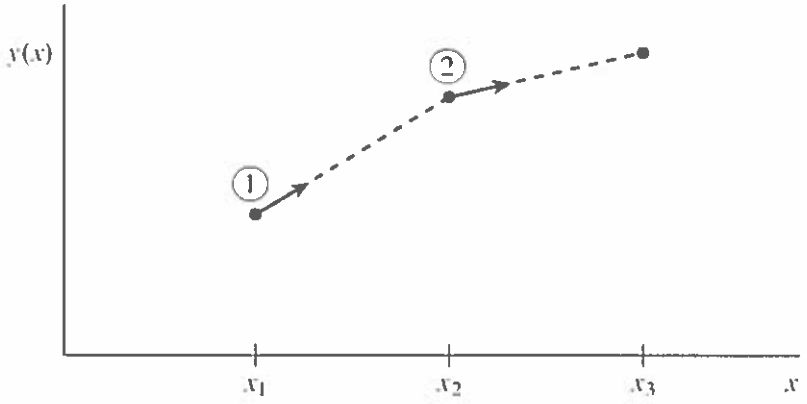

Figure 16.1.1. Euler's method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

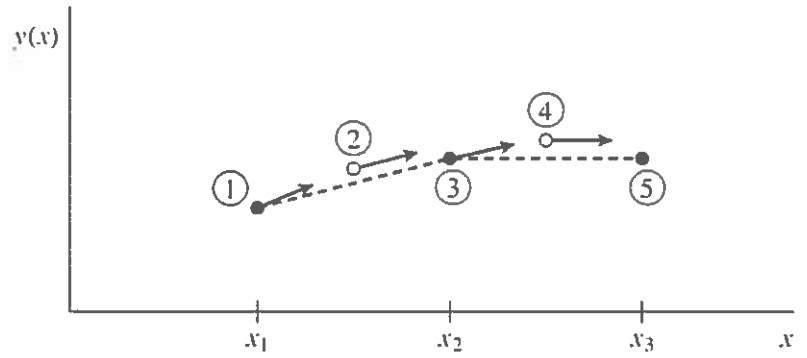

Figure 16.1.2. Midpoint method. Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open 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:

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

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

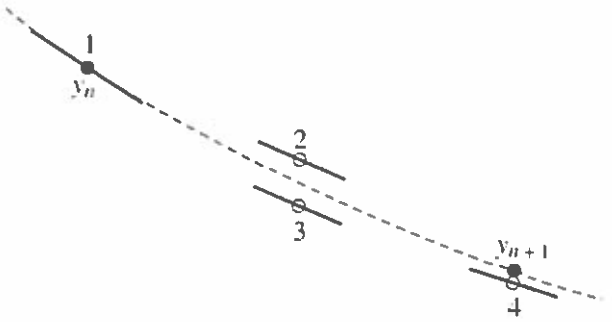

Figure 16.1.3. Fourth-order Runge-Kutta method. In each step the derivative is evaluated four times: once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the final function value (shown as a filled dot) is calculated. (See text for details.)

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`.

```
#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 xh,hh,h6,*dym,*dyt,*yt;

    dym=vector(1,n);
    dyt=vector(1,n);
```

```
    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*dyt[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 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 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 rkdumb");
        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,

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 — stepsize 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?

Let us denote the exact solution for an advance from x to $x + 2h$ by $y(x + 2h)$ and the two approximate solutions by y_1 (one step $2h$) and y_2 (2 steps each of size h). Since the basic method is fourth order, the true solution and the two numerical approximations are related by

$$\begin{aligned} y(x + 2h) &= y_1 + (2h)^5\phi + O(h^6) + \dots \\ y(x + 2h) &= y_2 + 2(h^5)\phi + O(h^6) + \dots \end{aligned} \tag{16.2.1}$$

where, to order h^5, the value ϕ remains constant over the step. [Taylor series expansion tells us the ϕ is a number whose order of magnitude is $y^{(5)}(x)/5!$.] The first expression in (16.2.1) involves $(2h)^5$ since the stepsize is $2h$, while the second expression involves $2(h^5)$ since the error on each step is $h^5\phi$. The difference between the two numerical estimates is a convenient indicator of truncation error

$$\Delta \equiv y_2 - y_1 \tag{16.2.2}$$

It is this difference that we shall endeavor to keep to a desired degree of accuracy, neither too large nor too small. We do this by adjusting h.

It might also occur to you that, ignoring terms of order h^6 and higher, we can solve the two equations in (16.2.1) to improve our numerical estimate of the true solution $y(x + 2h)$, namely,

$$y(x + 2h) = y_2 + \frac{\Delta}{15} + O(h^6) \tag{16.2.3}$$

This estimate is accurate to *fifth order*, one order higher than the original Runge-Kutta steps. However, we can't have our cake and eat it: (16.2.3) may be fifth-order accurate, but we have no way of monitoring *its* truncation error. Higher order is not always higher accuracy! Use of (16.2.3) rarely does harm, but we have no way of directly knowing whether it is doing any good. Therefore we should use Δ as the error estimate and take as "gravy" any additional accuracy gain derived from (16.2.3). In the technical literature, use of a procedure like (16.2.3) is called "local extrapolation."

An alternative stepsize adjustment algorithm is based on the *embedded Runge-Kutta formulas*, originally invented by Fehlberg. An interesting fact about Runge-Kutta formulas is that for orders M higher than four, more than M function evaluations (though never more than $M + 2$) are required. This accounts for the

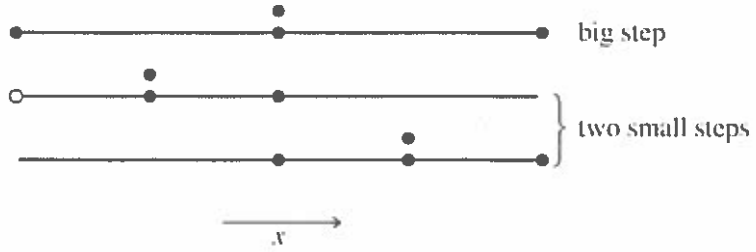

Figure 16.2.1. Step-doubling as a means for adaptive stepsize control in fourth-order Runge-Kutta. Points where the derivative is evaluated are shown as filled circles. The open circle represents the same derivatives as the filled circle immediately above it, so the total number of evaluations is 11 per two steps. Comparing the accuracy of the big step with the two small steps gives a criterion for adjusting the stepsize on the next step, or for rejecting the current step as inaccurate.

popularity of the classical fourth-order method: It seems to give the most bang for the buck. However, Fehlberg discovered a fifth-order method with six function evaluations where another combination of the six functions gives a fourth-order method. The difference between the two estimates of $y(x+h)$ can then be used as an estimate of the truncation error to adjust the stepsize. Since Fehlberg's original formula, several other embedded Runge-Kutta formulas have been found.

Many practitioners were at one time wary of the robustness of Runge-Kutta-Fehlberg methods. The feeling was that using the same evaluation points to advance the function and to estimate the error was riskier than step-doubling, where the error estimate is based on independent function evaluations. However, experience has shown that this concern is not a problem in practice. Accordingly, embedded Runge-Kutta formulas, which are roughly a factor of two more efficient, have superseded algorithms based on step-doubling.

The general form of a fifth-order Runge-Kutta formula is

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + a_2 h, y_n + b_{21} k_1) \\ &\cdots \\ k_6 &= hf(x_n + a_6 h, y_n + b_{61} k_1 + \cdots + b_{65} k_5) \\ y_{n+1} &= y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 + O(h^6) \end{aligned} \tag{16.2.4}$$

The embedded fourth-order formula is

$$y^*_{n+1} = y_n + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4 + c^*_5 k_5 + c^*_6 k_6 + O(h^5) \tag{16.2.5}$$

and so the error estimate is

$$\Delta \equiv y_{n+1} - y^*_{n+1} = \sum_{i=1}^{6} (c_i - c^*_i) k_i \tag{16.2.6}$$

The particular values of the various constants that we favor are those found by Cash and Karp [2], and given in the accompanying table. These give a more efficient method than Fehlberg's original values, with somewhat better error properties.

Cash-Karp Parameters for Embedded Runga-Kutta Method

i	a_i	b_{ij}					c_i	c_i^*
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$
$j =$		1	2	3	4	5		

Now that we know, at least approximately, what our error is, we need to consider how to keep it within desired bounds. What is the relation between Δ and h? According to (16.2.4) – (16.2.5), Δ scales as h^5. If we take a step h_1 and produce an error Δ_1, therefore, the step h_0 that *would have given* some other value Δ_0 is readily estimated as

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2} \tag{16.2.7}$$

Henceforth we will let Δ_0 denote the *desired* accuracy. Then equation (16.2.7) is used in two ways: If Δ_1 is larger than Δ_0 in magnitude, the equation tells how much to decrease the stepsize *when we retry the present (failed) step.* If Δ_1 is smaller than Δ_0, on the other hand, then the equation tells how much we can safely increase the stepsize *for the next step.* Local extrapolation consists in accepting the fifth order value y_{n+1}, even though the error estimate actually applies to the fourth order value y_{n+1}^*.

Our notation hides the fact that Δ_0 is actually a vector of desired accuracies, one for each equation in the set of ODEs. In general, our accuracy requirement will be that all equations are within their respective allowed errors. In other words, we will rescale the stepsize according to the needs of the "worst-offender" equation.

How is Δ_0, the desired accuracy, related to some looser prescription like "get a solution good to one part in 10^6"? That can be a subtle question, and it depends on exactly what your application is! You may be dealing with a set of equations whose dependent variables differ enormously in magnitude. In that case, you probably want to use fractional errors, $\Delta_0 = \epsilon y$, where ϵ is the number like 10^{-6} or whatever. On the other hand, you may have oscillatory functions that pass through zero but are bounded by some maximum values. In that case you probably want to set Δ_0 equal to ϵ times those maximum values.

A convenient way to fold these considerations into a generally useful stepper routine is this: One of the arguments of the routine will of course be the vector of dependent variables at the beginning of a proposed step. Call that `y[1..n]`. Let us require the user to specify for each step another, corresponding, vector argument `yscal[1..n]`, and also an overall tolerance level `eps`. Then the desired accuracy

for the ith equation will be taken to be

$$\Delta_0 = \texttt{eps} \times \texttt{yscal[i]} \tag{16.2.8}$$

If you desire constant fractional errors, plug a pointer to y into the pointer to `yscal` calling slot (no need to copy the values into a different array). If you desire constant absolute errors relative to some maximum values, set the elements of `yscal` equal to those maximum values. A useful "trick" for getting constant fractional errors *except* "very" near zero crossings is to set `yscal[i]` equal to $|\texttt{y[i]}| + |h \times \texttt{dydx[i]}|$. (The routine `odeint`, below, does this.)

Here is a more technical point. We have to consider one additional possibility for `yscal`. The error criteria mentioned thus far are "local," in that they bound the error of each step individually. In some applications you may be unusually sensitive about a "global" accumulation of errors, from beginning to end of the integration and in the worst possible case where the errors all are presumed to add with the same sign. Then, the smaller the stepsize h, the smaller the value Δ_0 that you will need to impose. Why? Because there will be *more steps* between your starting and ending values of x. In such cases you will want to set `yscal` proportional to h, typically to something like

$$\Delta_0 = \epsilon h \times \texttt{dydx[i]} \tag{16.2.9}$$

This enforces fractional accuracy ϵ not on the values of y but (much more stringently) on the *increments* to those values at each step. But now look back at (16.2.7). If Δ_0 has an implicit scaling with h, then the exponent 0.20 is no longer correct: When the stepsize is reduced from a too-large value, the new predicted value h_1 will fail to meet the desired accuracy when `yscal` is also altered to this new h_1 value. Instead of $0.20 = 1/5$, we must scale by the exponent $0.25 = 1/4$ for things to work out.

The exponents 0.20 and 0.25 are not really very different. This motivates us to adopt the following pragmatic approach, one that frees us from having to know in advance whether or not you, the user, plan to scale your `yscal`'s with stepsize. Whenever we decrease a stepsize, let us use the larger value of the exponent (whether we need it or not!), and whenever we increase a stepsize, let us use the smaller exponent. Furthermore, because our estimates of error are not exact, but only accurate to the leading order in h, we are advised to put in a safety factor S which is a few percent smaller than unity. Equation (16.2.7) is thus replaced by

$$h_0 = \begin{cases} Sh_1 \left| \dfrac{\Delta_0}{\Delta_1} \right|^{0.20} & \Delta_0 \geq \Delta_1 \\ Sh_1 \left| \dfrac{\Delta_0}{\Delta_1} \right|^{0.25} & \Delta_0 < \Delta_1 \end{cases} \tag{16.2.10}$$

We have found this prescription to be a reliable one in practice.

Here, then, is a stepper program that takes one "quality-controlled" Runge-Kutta step.

```
#include <math.h>
#include "nrutil.h"
#define SAFETY 0.9
#define PGROW -0.2
#define PSHRNK -0.25
#define ERRCON 1.89e-4
```
The value ERRCON equals (5/SAFETY) raised to the power (1/PGROW), see use below.

```
void rkqs(float y[], float dydx[], int n, float *x, float htry, float eps,
    float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []))
```
Fifth-order Runge-Kutta step with monitoring of local truncation error to ensure accuracy and adjust stepsize. Input are the dependent variable vector y[1..n] and its derivative dydx[1..n] at the starting value of the independent variable x. Also input are the stepsize to be attempted htry, the required accuracy eps, and the vector yscal[1..n] against which the error is scaled. On output, y and x are replaced by their new values, hdid is the stepsize that was actually accomplished, and hnext is the estimated next stepsize. derivs is the user-supplied routine that computes the right-hand side derivatives.

```
{
    void rkck(float y[], float dydx[], int n, float x, float h,
        float yout[], float yerr[], void (*derivs)(float, float [], float []));
    int i;
    float errmax,h,htemp,xnew,*yerr,*ytemp;

    yerr=vector(1,n);
    ytemp=vector(1,n);
    h=htry;                                  Set stepsize to the initial trial value.
    for (;;) {
        rkck(y,dydx,n,*x,h,ytemp,yerr,derivs);       Take a step.
        errmax=0.0;                          Evaluate accuracy.
        for (i=1;i<=n;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
        errmax /= eps;                       Scale relative to required tolerance.
        if (errmax <= 1.0) break;            Step succeeded. Compute size of next step.
        htemp=SAFETY*h*pow(errmax,PSHRNK);
        Truncation error too large, reduce stepsize.
        h=(h >= 0.0 ? FMAX(htemp,0.1*h) : FMIN(htemp,0.1*h));
        No more than a factor of 10.
        xnew=(*x)+h;
        if (xnew == *x) nrerror("stepsize underflow in rkqs");
    }
    if (errmax > ERRCON) *hnext=SAFETY*h*pow(errmax,PGROW);
    else *hnext=5.0*h;                       No more than a factor of 5 increase.
    *x += (*hdid=h);
    for (i=1;i<=n;i++) y[i]=ytemp[i];
    free_vector(ytemp,1,n);
    free_vector(yerr,1,n);
}
```

The routine rkqs calls the routine rkck to take a Cash-Karp Runge-Kutta step:

```
#include "nrutil.h"

void rkck(float y[], float dydx[], int n, float x, float h, float yout[],
    float yerr[], void (*derivs)(float, float [], float []))
```
Given values for n variables y[1..n] and their derivatives dydx[1..n] known at x, use the fifth-order Cash-Karp Runge-Kutta method to advance the solution over an interval h and return the incremented variables as yout[1..n]. Also return an estimate of the local truncation error in yout using the embedded fourth-order method. The user supplies the routine derivs(x,y,dydx), which returns derivatives dydx at x.

```
{
    int i;
```

```
    static float a2=0.2,a3=0.3,a4=0.6,a5=1.0,a6=0.875,b21=0.2,
        b31=3.0/40.0,b32=9.0/40.0,b41=0.3,b42 = -0.9,b43=1.2,
        b51 = -11.0/54.0, b52=2.5,b53 = -70.0/27.0,b54=35.0/27.0,
        b61=1631.0/55296.0,b62=175.0/512.0,b63=575.0/13824.0,
        b64=44275.0/110592.0,b65=253.0/4096.0,c1=37.0/378.0,
        c3=250.0/621.0,c4=125.0/594.0,c6=512.0/1771.0,
        dc5 = -277.00/14336.0;
    float dc1=c1-2825.0/27648.0,dc3=c3-18575.0/48384.0,
        dc4=c4-13525.0/55296.0,dc6=c6-0.25;
    float *ak2,*ak3,*ak4,*ak5,*ak6,*ytemp;

    ak2=vector(1,n);
    ak3=vector(1,n);
    ak4=vector(1,n);
    ak5=vector(1,n);
    ak6=vector(1,n);
    ytemp=vector(1,n);
    for (i=1;i<=n;i++)                  First step.
        ytemp[i]=y[i]+b21*h*dydx[i];
    (*derivs)(x+a2*h,ytemp,ak2);        Second step.
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b31*dydx[i]+b32*ak2[i]);
    (*derivs)(x+a3*h,ytemp,ak3);        Third step.
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b41*dydx[i]+b42*ak2[i]+b43*ak3[i]);
    (*derivs)(x+a4*h,ytemp,ak4);        Fourth step.
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b51*dydx[i]+b52*ak2[i]+b53*ak3[i]+b54*ak4[i]);
    (*derivs)(x+a5*h,ytemp,ak5);        Fifth step.
    for (i=1;i<=n;i++)
        ytemp[i]=y[i]+h*(b61*dydx[i]+b62*ak2[i]+b63*ak3[i]+b64*ak4[i]+b65*ak5[i]);
    (*derivs)(x+a6*h,ytemp,ak6);        Sixth step.
    for (i=1;i<=n;i++)                  Accumulate increments with proper weights.
        yout[i]=y[i]+h*(c1*dydx[i]+c3*ak3[i]+c4*ak4[i]+c6*ak6[i]);
    for (i=1;i<=n;i++)
        yerr[i]=h*(dc1*dydx[i]+dc3*ak3[i]+dc4*ak4[i]+dc5*ak5[i]+dc6*ak6[i]);
        Estimate error as difference between fourth and fifth order methods.
    free_vector(ytemp,1,n);
    free_vector(ak6,1,n);
    free_vector(ak5,1,n);
    free_vector(ak4,1,n);
    free_vector(ak3,1,n);
    free_vector(ak2,1,n);
}
```

Noting that the above routines are all in single precision, don't be too greedy in specifying eps. The punishment for excessive greediness is interesting and worthy of Gilbert and Sullivan's *Mikado*: The routine can always achieve an apparent *zero* error by making the stepsize so small that quantities of order hy' add to quantities of order y as if they were zero. Then the routine chugs happily along taking infinitely many infinitesimal steps and never changing the dependent variables one iota. (You guard against this catastrophic loss of your computer budget by signaling on abnormally small stepsizes or on the dependent variable vector remaining unchanged from step to step. On a personal workstation you guard against it by not taking too long a lunch hour while your program is running.)

Here is a full-fledged "driver" for Runge-Kutta with adaptive stepsize control. We warmly recommend this routine, or one like it, for a variety of problems, notably

including garden-variety ODEs or sets of ODEs, and definite integrals (augmenting the methods of Chapter 4). For storage of intermediate results (if you desire to inspect them) we assume that the top-level pointer references *xp and **yp have been validly initialized (e.g., by the utilities vector() and matrix()). Because steps occur at unequal intervals results are only stored at intervals greater than dxsav. The top-level variable kmax indicates the maximum number of steps that can be stored. If kmax=0 there is no intermediate storage, and the pointers *xp and **yp need not point to valid memory. Storage of steps stops if kmax is exceeded, except that the ending values are always stored. Again, these controls are merely indicative of what you might need. The routine odeint should be customized to the problem at hand.

```
#include <math.h>
#include "nrutil.h"
#define MAXSTP 10000
#define TINY 1.0e-30

extern int kmax,kount;
extern float *xp,**yp,dxsav;
```

User storage for intermediate results. Preset kmax and dxsav in the calling program. If kmax $\neq$ 0 results are stored at approximate intervals dxsav in the arrays xp[1..kount], yp[1..nvar][1..kount], where kount is output by odeint. Defining declarations for these variables, with memory allocations xp[1..kmax] and yp[1..nvar][1..kmax] for the arrays, should be in the calling program.

```
void odeint(float ystart[], int nvar, float x1, float x2, float eps, float h1,
    float hmin, int *nok, int *nbad,
    void (*derivs)(float, float [], float []),
    void (*rkqs)(float [], float [], int, float *, float, float, float [],
    float *, float *, void (*)(float, float [], float [])))
```

Runge-Kutta driver with adaptive stepsize control. Integrate starting values ystart[1..nvar] from x1 to x2 with accuracy eps, storing intermediate results in global variables. h1 should be set as a guessed first stepsize, hmin as the minimum allowed stepsize (can be zero). On output nok and nbad are the number of good and bad (but retried and fixed) steps taken, and ystart is replaced by values at the end of the integration interval. derivs is the user-supplied routine for calculating the right-hand side derivative, while rkqs is the name of the stepper routine to be used.

```
{
    int nstp,i;
    float xsav,x,hnext,hdid,h;
    float *yscal,*y,*dydx;

    yscal=vector(1,nvar);
    y=vector(1,nvar);
    dydx=vector(1,nvar);
    x=x1;
    h=SIGN(h1,x2-x1);
    *nok = (*nbad) = kount = 0;
    for (i=1;i<=nvar;i++) y[i]=ystart[i];
    if (kmax > 0) xsav=x-dxsav*2.0;                    Assures storage of first step.
    for (nstp=1;nstp<=MAXSTP;nstp++) {                 Take at most MAXSTP steps.
        (*derivs)(x,y,dydx);
        for (i=1;i<=nvar;i++)
            Scaling used to monitor accuracy. This general-purpose choice can be modified
            if need be.
            yscal[i]=fabs(y[i])+fabs(dydx[i]*h)+TINY;
        if (kmax > 0 && kount < kmax-1 && fabs(x-xsav) > fabs(dxsav)) {
            xp[++kount]=x;                             Store intermediate results.
            for (i=1;i<=nvar;i++) yp[i][kount]=y[i];
            xsav=x;
        }
        if ((x+h-x2)*(x+h-x1) > 0.0) h=x2-x;           If stepsize can overshoot, decrease.
```

```
        (*rkqs)(y,dydx,nvar,&x,h,eps,yscal,&hdid,&hnext,derivs);
        if (hdid == h) ++(*nok); else ++(*nbad);
        if ((x-x2)*(x2-x1) >= 0.0) {                Are we done?
            for (i=1;i<=nvar;i++) ystart[i]=y[i];
            if (kmax) {
                xp[++kount]=x;                      Save final step.
                for (i=1;i<=nvar;i++) yp[i][kount]=y[i];
            }
            free_vector(dydx,1,nvar);
            free_vector(y,1,nvar);
            free_vector(yscal,1,nvar);
            return;                                 Normal exit.
        }
        if (fabs(hnext) <= hmin) nrerror("Step size too small in odeint");
        h=hnext;
    }
    nrerror("Too many steps in routine odeint");
}
```

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Cash, J.R., and Karp, A.H. 1990, *ACM Transactions on Mathematical Software*, vol. 16, pp. 201–222. [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.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall).

16.3 Modified Midpoint Method

This section discusses the *modified midpoint method*, which advances a vector of dependent variables $y(x)$ from a point x to a point $x + H$ by a sequence of n substeps each of size h,

$$h = H/n \tag{16.3.1}$$

In principle, one could use the modified midpoint method in its own right as an ODE integrator. In practice, the method finds its most important application as a part of the more powerful Bulirsch-Stoer technique, treated in §16.4. You can therefore consider this section as a preamble to §16.4.

The number of right-hand side evaluations required by the modified midpoint method is $n + 1$. The formulas for the method are

$$\begin{aligned} z_0 &\equiv y(x) \\ z_1 &= z_0 + hf(x, z_0) \\ z_{m+1} &= z_{m-1} + 2hf(x + mh, z_m) \qquad \text{for} \quad m = 1, 2, \ldots, n-1 \\ y(x + H) \approx y_n &\equiv \frac{1}{2}[z_n + z_{n-1} + hf(x + H, z_n)] \end{aligned} \tag{16.3.2}$$

Here the z's are intermediate approximations which march along in steps of h, while y_n is the final approximation to $y(x + H)$. The method is basically a "centered difference" or "midpoint" method (compare equation 16.1.2), except at the first and last points. Those give the qualifier "modified."

The modified midpoint method is a second-order method, like (16.1.2), but with the advantage of requiring (asymptotically for large n) only one derivative evaluation per step h instead of the two required by second-order Runge-Kutta. Perhaps there are applications where the simplicity of (16.3.2), easily coded in-line in some other program, recommends it. In general, however, use of the modified midpoint method by itself will be dominated by the embedded Runge-Kutta method with adaptive stepsize control, as implemented in the preceding section.

The usefulness of the modified midpoint method to the Bulirsch-Stoer technique (§16.4) derives from a "deep" result about equations (16.3.2), due to Gragg. It turns out that the error of (16.3.2), expressed as a power series in h, the stepsize, contains only *even* powers of h,

$$y_n - y(x + H) = \sum_{i=1}^{\infty} \alpha_i h^{2i} \tag{16.3.3}$$

where H is held constant, but h changes by varying n in (16.3.1). The importance of this even power series is that, if we play our usual tricks of combining steps to knock out higher-order error terms, we can gain *two* orders at a time!

For example, suppose n is even, and let $y_{n/2}$ denote the result of applying (16.3.1) and (16.3.2) with half as many steps, $n \to n/2$. Then the estimate

$$y(x + H) \approx \frac{4y_n - y_{n/2}}{3} \tag{16.3.4}$$

is *fourth-order* accurate, the same as fourth-order Runge-Kutta, but requires only about 1.5 derivative evaluations per step h instead of Runge-Kutta's 4 evaluations. Don't be too anxious to implement (16.3.4), since we will soon do even better.

Now would be a good time to look back at the routine `qsimp` in §4.2, and especially to compare equation (4.2.4) with equation (16.3.4) above. You will see that the transition in Chapter 4 to the idea of Richardson extrapolation, as embodied in Romberg integration of §4.3, is exactly analogous to the transition in going from this section to the next one.

Here is the routine that implements the modified midpoint method, which will be used below.

```
#include "nrutil.h"

void mmid(float y[], float dydx[], int nvar, float xs, float htot, int nstep,
    float yout[], void (*derivs)(float, float[], float[]))
```

Modified midpoint step. At `xs`, input the dependent variable vector `y[1..nvar]` and its derivative vector `dydx[1..nvar]`. Also input is `htot`, the total step to be made, and `nstep`, the number of substeps to be used. The output is returned as `yout[1..nvar]`, which need not be a distinct array from `y`; if it is distinct, however, then `y` and `dydx` are returned undamaged.

```
{
    int n,i;
    float x,swap,h2,h,*ym,*yn;
```

```
    ym=vector(1,nvar);
    yn=vector(1,nvar);
    h=htot/nstep;                          Stepsize this trip.
    for (i=1;i<=nvar;i++) {
        ym[i]=y[i];
        yn[i]=y[i]+h*dydx[i];              First step.
    }
    x=xs+h;
    (*derivs)(x,yn,yout);                  Will use yout for temporary storage of deriva-
    h2=2.0*h;                                  tives.
    for (n=2;n<=nstep;n++) {               General step.
        for (i=1;i<=nvar;i++) {
            swap=ym[i]+h2*yout[i];
            ym[i]=yn[i];
            yn[i]=swap;
        }
        x += h;
        (*derivs)(x,yn,yout);
    }
    for (i=1;i<=nvar;i++)                  Last step.
        yout[i]=0.5*(ym[i]+yn[i]+h*yout[i]);
    free_vector(yn,1,nvar);
    free_vector(ym,1,nvar);
}
```

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.4.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §7.2.12.

16.4 Richardson Extrapolation and the Bulirsch-Stoer Method

The techniques described in this section are not for differential equations containing nonsmooth functions. For example, you might have a differential equation whose right-hand side involves a function that is evaluated by table look-up and interpolation. If so, go back to Runge-Kutta with adaptive stepsize choice: That method does an excellent job of feeling its way through rocky or discontinuous terrain. It is also an excellent choice for quick-and-dirty, low-accuracy solution of a set of equations. A second warning is that the techniques in this section are not particularly good for differential equations that have singular points *inside* the interval of integration. A regular solution must tiptoe very carefully across such points. Runge-Kutta with adaptive stepsize can sometimes effect this; more generally, there are special techniques available for such problems, beyond our scope here.

Apart from those two caveats, we believe that the Bulirsch-Stoer method, discussed in this section, is the best known way to obtain high-accuracy solutions to ordinary differential equations with minimal computational effort. (A possible exception, infrequently encountered in practice, is discussed in §16.7.)

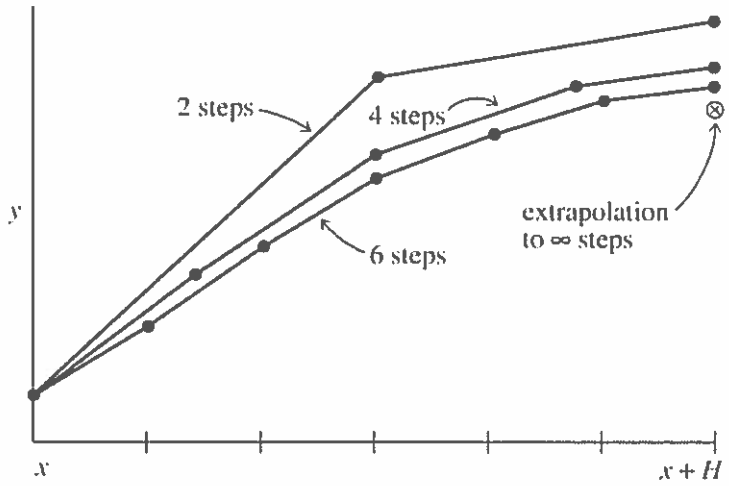

Figure 16.4.1. Richardson extrapolation as used in the Bulirsch-Stoer method. A large interval H is spanned by different sequences of finer and finer substeps. Their results are extrapolated to an answer that is supposed to correspond to infinitely fine substeps. In the Bulirsch-Stoer method, the integrations are done by the modified midpoint method, and the extrapolation technique is rational function or polynomial extrapolation.

Three key ideas are involved. The first is *Richardson's deferred approach to the limit*, which we already met in §4.3 on Romberg integration. The idea is to consider the final answer of a numerical calculation as itself being an analytic function (if a complicated one) of an adjustable parameter like the stepsize h. That analytic function can be probed by performing the calculation with various values of h, *none* of them being necessarily small enough to yield the accuracy that we desire. When we know enough about the function, we *fit* it to some analytic form, and then *evaluate* it at that mythical and golden point $h = 0$ (see Figure 16.4.1). Richardson extrapolation is a method for turning straw into gold! (Lead into gold for alchemist readers.)

The second idea has to do with what kind of fitting function is used. Bulirsch and Stoer first recognized the strength of *rational function extrapolation* in Richardson-type applications. That strength is to break the shackles of the power series and its limited radius of convergence, out only to the distance of the first pole in the complex plane. Rational function fits can remain good approximations to analytic functions even after the various terms in powers of h all have comparable magnitudes. In other words, h can be so large as to make the whole notion of the "order" of the method meaningless — and the method can still work superbly. Nevertheless, more recent experience suggests that for smooth problems straightforward polynomial extrapolation is slightly more efficient than rational function extrapolation. We will accordingly adopt polynomial extrapolation as the default, but the routine `bsstep` below allows easy substitution of one kind of extrapolation for the other. You might wish at this point to review §3.1–§3.2, where polynomial and rational function extrapolation were already discussed.

The third idea was discussed in the section before this one, namely to use a method whose error function is strictly even, allowing the rational function or polynomial approximation to be in terms of the variable h^2 instead of just h.

Put these ideas together and you have the *Bulirsch-Stoer method* [1]. A single Bulirsch-Stoer step takes us from x to $x + H$, where H is supposed to be quite a large

— not at all infinitesimal — distance. That single step is a grand leap consisting of many (e.g., dozens to hundreds) substeps of modified midpoint method, which are then extrapolated to zero stepsize.

The sequence of separate attempts to cross the interval H is made with increasing values of n, the number of substeps. Bulirsch and Stoer originally proposed the sequence

$$n = 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, \ldots, [n_j = 2n_{j-2}], \ldots \tag{16.4.1}$$

More recent work by Deuflhard [2,3] suggests that the sequence

$$n = 2, 4, 6, 8, 10, 12, 14, \ldots, [n_j = 2j], \ldots \tag{16.4.2}$$

is usually more efficient. For each step, we do not know in advance how far up this sequence we will go. After each successive n is tried, a polynomial extrapolation is attempted. That extrapolation gives both extrapolated values and error estimates. If the errors are not satisfactory, we go higher in n. If they are satisfactory, we go on to the next step and begin anew with $n = 2$.

Of course there must be some upper limit, beyond which we conclude that there is some obstacle in our path in the interval H, so that we must reduce H rather than just subdivide it more finely. In the implementations below, the maximum number of n's to be tried is called KMAXX. For reasons described below we usually take this equal to 8; the 8th value of the sequence (16.4.2) is 16, so this is the maximum number of subdivisions of H that we allow.

We enforce error control, as in the Runge-Kutta method, by monitoring internal consistency, and adapting stepsize to match a prescribed bound on the local truncation error. Each new result from the sequence of modified midpoint integrations allows a tableau like that in §3.1 to be extended by one additional set of diagonals. The size of the new correction added at each stage is taken as the (conservative) error estimate. How should we use this error estimate to adjust the stepsize? The best strategy now known is due to Deuflhard [2,3]. For completeness we describe it here:

Suppose the absolute value of the error estimate returned from the kth column (and hence the $k+1$st row) of the extrapolation tableau is $\epsilon_{k+1,k}$. Error control is enforced by requiring

$$\epsilon_{k+1,k} < \epsilon \tag{16.4.3}$$

as the criterion for accepting the current step, where ϵ is the required tolerance. For the even sequence (16.4.2) the order of the method is $2k+1$:

$$\epsilon_{k+1,k} \sim H^{2k+1} \tag{16.4.4}$$

Thus a simple estimate of a new stepsize H_k to obtain convergence in a fixed column k would be

$$H_k = H \left(\frac{\epsilon}{\epsilon_{k+1,k}} \right)^{1/(2k+1)} \tag{16.4.5}$$

Which column k should we aim to achieve convergence in? Let's compare the work required for different k. Suppose A_k is the work to obtain row k of the extrapolation tableau, so A_{k+1} is the work to obtain column k. We will assume the work is dominated by the cost of evaluating the functions defining the right-hand sides of the differential equations. For n_k subdivisions in H, the number of function evaluations can be found from the recurrence

$$\begin{aligned} A_1 &= n_1 + 1 \\ A_{k+1} &= A_k + n_{k+1} \end{aligned} \tag{16.4.6}$$

The work per unit step to get column k is A_{k+1}/H_k, which we nondimensionalize with a factor of H and write as

$$W_k = \frac{A_{k+1}}{H_k} H \tag{16.4.7}$$

$$= A_{k+1} \left(\frac{\epsilon_{k+1,k}}{\epsilon} \right)^{1/(2k+1)} \tag{16.4.8}$$

The quantities W_k can be calculated during the integration. The optimal column index q is then defined by

$$W_q = \min_{k=1,\ldots,k_f} W_k \tag{16.4.9}$$

where k_f is the final column, in which the error criterion (16.4.3) was satisfied. The q determined from (16.4.9) defines the stepsize H_q to be used as the next basic stepsize, so that we can expect to get convergence in the optimal column q.

Two important refinements have to be made to the strategy outlined so far:

- If the current H is "too small," then k_f will be "too small," and so q remains "too small." It may be desirable to increase H and aim for convergence in a column $q > k_f$.
- If the current H is "too big," we may not converge at all on the current step and we will have to decrease H. We would like to detect this by monitoring the quantities $\epsilon_{k+1,k}$ for each k so we can stop the current step as soon as possible.

Deuflhard's prescription for dealing with these two problems uses ideas from communication theory to determine the "average expected convergence behavior" of the extrapolation. His model produces certain correction factors $\alpha(k,q)$ by which H_k is to be multiplied to try to get convergence in column q. The factors $\alpha(k,q)$ depend only on ϵ and the sequence $\{n_i\}$ and so can be computed once during initialization:

$$\alpha(k,q) = \epsilon^{\frac{A_{k+1} - A_{q+1}}{(2k+1)(A_{q+1} - A_{1+1})}} \qquad \text{for} \quad k < q \tag{16.4.10}$$

with $\alpha(q,q) = 1$.

Now to handle the first problem, suppose convergence occurs in column $q = k_f$. Then rather than taking H_q for the next step, we might aim to increase the stepsize to get convergence in column $q+1$. Since we don't have H_{q+1} available from the computation, we estimate it as

$$H_{q+1} = H_q \alpha(q, q+1) \tag{16.4.11}$$

By equation (16.4.7) this replacement is efficient, i.e., reduces the work per unit step, if

$$\frac{A_{q+1}}{H_q} > \frac{A_{q+2}}{H_{q+1}} \tag{16.4.12}$$

or

$$A_{q+1} \alpha(q, q+1) > A_{q+2} \tag{16.4.13}$$

During initialization, this inequality can be checked for $q = 1, 2, \ldots$ to determine $k_{\max}$, the largest allowed column. Then when (16.4.12) is satisfied it will always be efficient to use H_{q+1}. (In practice we limit $k_{\max}$ to 8 even when ϵ is very small as there is very little further gain in efficiency whereas roundoff can become a problem.)

The problem of stepsize reduction is handled by computing stepsize estimates

$$\bar{H}_k \equiv H_k \alpha(k,q), \qquad k = 1, \ldots, q-1 \tag{16.4.14}$$

during the current step. The $\bar{H}$'s are estimates of the stepsize to get convergence in the optimal column q. If any $\bar{H}_k$ is "too small," we abandon the current step and restart using $\bar{H}_k$. The criterion of being "too small" is taken to be

$$H_k \alpha(k, q+1) < H \tag{16.4.15}$$

The α's satisfy $\alpha(k, q+1) > \alpha(k,q)$.

During the first step, when we have no information about the solution, the stepsize reduction check is made for all k. Afterwards, we test for convergence and for possible stepsize reduction only in an "order window"

$$\max(1, q-1) \le k \le \min(k_{\max}, q+1) \tag{16.4.16}$$

The rationale for the order window is that if convergence appears to occur for $k < q-1$ it is often spurious, resulting from some fortuitously small error estimate in the extrapolation. On the other hand, if you need to go beyond $k = q+1$ to obtain convergence, your local model of the convergence behavior is obviously not very good and you need to cut the stepsize and reestablish it.

In the routine `bsstep`, these various tests are actually carried out using quantities

$$\epsilon(k) \equiv \frac{H}{H_k} = \left(\frac{\epsilon_{k+1,k}}{\epsilon}\right)^{1/(2k+1)} \tag{16.4.17}$$

called `err[k]` in the code. As usual, we include a "safety factor" in the stepsize selection. This is implemented by replacing ϵ by 0.25ϵ. Other safety factors are explained in the program comments.

Note that while the optimal convergence column is restricted to increase by at most one on each step, a sudden drop in order is allowed by equation (16.4.9). This gives the method a degree of robustness for problems with discontinuities.

Let us remind you once again that *scaling* of the variables is often crucial for successful integration of differential equations. The scaling "trick" suggested in the discussion following equation (16.2.8) is a good general purpose choice, but not foolproof. Scaling by the maximum values of the variables is more robust, but requires you to have some prior information.

The following implementation of a Bulirsch-Stoer step has exactly the same calling sequence as the quality-controlled Runge-Kutta stepper `rkqs`. This means that the driver `odeint` in §16.2 can be used for Bulirsch-Stoer as well as Runge-Kutta: Just substitute `bsstep` for `rkqs` in `odeint`'s argument list. The routine `bsstep` calls `mmid` to take the modified midpoint sequences, and calls `pzextr`, given below, to do the polynomial extrapolation.

```
#include <math.h>
#include "nrutil.h"
#define KMAXX 8                          Maximum row number used in the extrapola-
                                           tion.
#define IMAXX (KMAXX+1)
#define SAFE1 0.25                       Safety factors.
#define SAFE2 0.7
#define REDMAX 1.0e-5                    Maximum factor for stepsize reduction.
#define REDMIN 0.7                       Minimum factor for stepsize reduction.
#define TINY 1.0e-30                     Prevents division by zero.
#define SCALMX 0.1                       1/SCALMX is the maximum factor by which a
                                           stepsize can be increased.
float **d,*x;
Pointers to matrix and vector used by pzextr or rzextr.

void bsstep(float y[], float dydx[], int nv, float *xx, float htry, float eps,
    float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []))
Bulirsch-Stoer step with monitoring of local truncation error to ensure accuracy and adjust
stepsize. Input are the dependent variable vector y[1..nv] and its derivative dydx[1..nv]
at the starting value of the independent variable x. Also input are the stepsize to be attempted
htry, the required accuracy eps, and the vector yscal[1..nv] against which the error is
scaled. On output, y and x are replaced by their new values, hdid is the stepsize that was
actually accomplished, and hnext is the estimated next stepsize. derivs is the user-supplied
routine that computes the right-hand side derivatives. Be sure to set htry on successive steps
```

to the value of hnext returned from the previous step, as is the case if the routine is called by odeint.

```
{
    void mmid(float y[], float dydx[], int nvar, float xs, float htot,
        int nstep, float yout[], void (*derivs)(float, float[], float[]));
    void pzextr(int iest, float xest, float yest[], float yz[], float dy[],
        int nv);
    int i,iq,k,kk,km;
    static int first=1,kmax,kopt;
    static float epsold = -1.0,xnew;
    float eps1,errmax,fact,h,red,scale,work,wrkmin,xest;
    float *err,*yerr,*ysav,*yseq;
    static float a[IMAXX+1];
    static float alf[KMAXX+1][KMAXX+1];
    static int nseq[IMAXX+1]={0,2,4,6,8,10,12,14,16,18};
    int reduct,exitflag=0;

    d=matrix(1,nv,1,KMAXX);
    err=vector(1,KMAXX);
    x=vector(1,KMAXX);
    yerr=vector(1,nv);
    ysav=vector(1,nv);
    yseq=vector(1,nv);
    if (eps != epsold) {                          A new tolerance, so reinitialize.
        *hnext = xnew = -1.0e29;                  "Impossible" values.
        eps1=SAFE1*eps;
        a[1]=nseq[1]+1;                           Compute work coefficients A_k.
        for (k=1;k<=KMAXX;k++) a[k+1]=a[k]+nseq[k+1];
        for (iq=2;iq<=KMAXX;iq++) {               Compute α(k, q).
            for (k=1;k<iq;k++)
                alf[k][iq]=pow(eps1,(a[k+1]-a[iq+1])/
                    ((a[iq+1]-a[1]+1.0)*(2*k+1)));
        }
        epsold=eps;
        for (kopt=2;kopt<KMAXX;kopt++)            Determine optimal row number for
            if (a[kopt+1] > a[kopt]*alf[kopt-1][kopt]) break;      convergence.
        kmax=kopt;
    }
    h=htry;
    for (i=1;i<=nv;i++) ysav[i]=y[i];             Save the starting values.
    if (*xx != xnew || h != (*hnext)) {           A new stepsize or a new integration:
        first=1;                                      re-establish the order window.
        kopt=kmax;
    }
    reduct=0;
    for (;;) {
        for (k=1;k<=kmax;k++) {                   Evaluate the sequence of modified
            xnew=(*xx)+h;                             midpoint integrations.
            if (xnew == (*xx)) nrerror("step size underflow in bsstep");
            mmid(ysav,dydx,nv,*xx,h,nseq[k],yseq,derivs);
            xest=SQR(h/nseq[k]);                  Squared, since error series is even.
            pzextr(k,xest,yseq,y,yerr,nv);        Perform extrapolation.
            if (k != 1) {                         Compute normalized error estimate
                errmax=TINY;                          ε(k).
                for (i=1;i<=nv;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
                errmax /= eps;                    Scale error relative to tolerance.
                km=k-1;
                err[km]=pow(errmax/SAFE1,1.0/(2*km+1));
            }
            if (k != 1 && (k >= kopt-1 || first)) {      In order window.
                if (errmax < 1.0) {                      Converged.
                    exitflag=1;
                    break;
                }
```

```
                if (k == kmax || k == kopt+1) {          Check for possible stepsize
                    red=SAFE2/err[km];                       reduction.
                    break;
                }
                else if (k == kopt && alf[kopt-1][kopt] < err[km]) {
                        red=1.0/err[km];
                        break;
                    }
                else if (kopt == kmax && alf[km][kmax-1] < err[km]) {
                        red=alf[km][kmax-1]*SAFE2/err[km];
                        break;
                    }
                else if (alf[km][kopt] < err[km]) {
                    red=alf[km][kopt-1]/err[km];
                    break;
                }
            }
        }
        if (exitflag) break;
        red=FMIN(red,REDMIN);                    Reduce stepsize by at least REDMIN
        red=FMAX(red,REDMAX);                        and at most REDMAX.
        h *= red;
        reduct=1;
    }                                            Try again.
    *xx=xnew;                                    Successful step taken.
    *hdid=h;
    first=0;
    wrkmin=1.0e35;                               Compute optimal row for convergence
    for (kk=1;kk<=km;kk++) {                         and corresponding stepsize.
        fact=FMAX(err[kk],SCALMX);
        work=fact*a[kk+1];
        if (work < wrkmin) {
            scale=fact;
            wrkmin=work;
            kopt=kk+1;
        }
    }
    *hnext=h/scale;
    if (kopt >= k && kopt != kmax && !reduct) {
    Check for possible order increase, but not if stepsize was just reduced.
        fact=FMAX(scale/alf[kopt-1][kopt],SCALMX);
        if (a[kopt+1]*fact <= wrkmin) {
            *hnext=h/fact;
            kopt++;
        }
    }
    free_vector(yseq,1,nv);
    free_vector(ysav,1,nv);
    free_vector(yerr,1,nv);
    free_vector(x,1,KMAXX);
    free_vector(err,1,KMAXX);
    free_matrix(d,1,nv,1,KMAXX);
}
```

The polynomial extrapolation routine is based on the same algorithm as `polint` §3.1. It is simpler in that it is always extrapolating to zero, rather than to an arbitrary value. However, it is more complicated in that it must individually extrapolate each component of a vector of quantities.

```
#include "nrutil.h"

extern float **d,*x;                          Defined in bsstep.

void pzextr(int iest, float xest, float yest[], float yz[], float dy[], int nv)
Use polynomial extrapolation to evaluate nv functions at x = 0 by fitting a polynomial to a
sequence of estimates with progressively smaller values x = xest, and corresponding function
vectors yest[1..nv]. This call is number iest in the sequence of calls. Extrapolated function
values are output as yz[1..nv], and their estimated error is output as dy[1..nv].
{
    int k1,j;
    float q,f2,f1,delta,*c;

    c=vector(1,nv);
    x[iest]=xest;                             Save current independent variable.
    for (j=1;j<=nv;j++) dy[j]=yz[j]=yest[j];
    if (iest == 1) {                          Store first estimate in first column.
        for (j=1;j<=nv;j++) d[j][1]=yest[j];
    } else {
        for (j=1;j<=nv;j++) c[j]=yest[j];
        for (k1=1;k1<iest;k1++) {
            delta=1.0/(x[iest-k1]-xest);
            f1=xest*delta;
            f2=x[iest-k1]*delta;
            for (j=1;j<=nv;j++) {             Propagate tableau 1 diagonal more.
                q=d[j][k1];
                d[j][k1]=dy[j];
                delta=c[j]-q;
                dy[j]=f1*delta;
                c[j]=f2*delta;
                yz[j] += dy[j];
            }
        }
        for (j=1;j<=nv;j++) d[j][iest]=dy[j];
    }
    free_vector(c,1,nv);
}
```

Current wisdom favors polynomial extrapolation over rational function extrapolation in the Bulirsch-Stoer method. However, our feeling is that this view is guided more by the kinds of problems used for tests than by one method being actually "better." Accordingly, we provide the optional routine `rzextr` for rational function extrapolation, an exact substitution for `pzextr` above.

```
#include "nrutil.h"

extern float **d,*x;                          Defined in bsstep.

void rzextr(int iest, float xest, float yest[], float yz[], float dy[], int nv)
Exact substitute for pzextr, but uses diagonal rational function extrapolation instead of poly-
nomial extrapolation.
{
    int k,j;
    float yy,v,ddy,c,b1,b,*fx;

    fx=vector(1,iest);
    x[iest]=xest;                             Save current independent variable.
    if (iest == 1)
        for (j=1;j<=nv;j++) {
            yz[j]=yest[j];
            d[j][1]=yest[j];
```

```
            dy[j]=yest[j];
        }
    else {
        for (k=1;k<iest;k++)
            fx[k+1]=x[iest-k]/xest;
        for (j=1;j<=nv;j++) {          Evaluate next diagonal in tableau.
            v=d[j][1];
            d[j][1]=yy=c=yest[j];
            for (k=2;k<=iest;k++) {
                b1=fx[k]*v;
                b=b1-c;
                if (b) {
                    b=(c-v)/b;
                    ddy=c*b;
                    c=b1*b;
                } else                 Care needed to avoid division by 0.
                    ddy=v;
                if (k != iest) v=d[j][k];
                d[j][k]=ddy;
                yy += ddy;
            }
            dy[j]=ddy;
            yz[j]=yy;
        }
    }
    free_vector(fx,1,iest);
}
```

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §7.2.14. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.2.

Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 399–422. [2]

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535. [3]

16.5 Second-Order Conservative Equations

Usually when you have a system of high-order differential equations to solve it is best to reformulate them as a system of first-order equations, as discussed in §16.0. There is a particular class of equations that occurs quite frequently in practice where you can gain about a factor of two in efficiency by differencing the equations directly. The equations are second-order systems where the derivative does not appear on the right-hand side:

$$y'' = f(x, y), \qquad y(x_0) = y_0, \qquad y'(x_0) = z_0 \tag{16.5.1}$$

As usual, y can denote a vector of values.

Stoermer's rule, dating back to 1907, has been a popular method for discretizing such systems. With $h = H/m$ we have

$$\begin{aligned} y_1 &= y_0 + h[z_0 + \tfrac{1}{2}hf(x_0, y_0)] \\ y_{k+1} - 2y_k + y_{k-1} &= h^2 f(x_0 + kh, y_k), \qquad k = 1, \ldots, m-1 \\ z_m &= (y_m - y_{m-1})/h + \tfrac{1}{2}hf(x_0 + H, y_m) \end{aligned} \tag{16.5.2}$$

Here z_m is $y'(x_0 + H)$. Henrici showed how to rewrite equations (16.5.2) to reduce roundoff error by using the quantities $\Delta_k \equiv y_{k+1} - y_k$. Start with

$$\begin{aligned} \Delta_0 &= h[z_0 + \tfrac{1}{2}hf(x_0, y_0)] \\ y_1 &= y_0 + \Delta_0 \end{aligned} \tag{16.5.3}$$

Then for $k = 1, \ldots, m-1$, set

$$\begin{aligned} \Delta_k &= \Delta_{k-1} + h^2 f(x_0 + kh, y_k) \\ y_{k+1} &= y_k + \Delta_k \end{aligned} \tag{16.5.4}$$

Finally compute the derivative from

$$z_m = \Delta_{m-1}/h + \tfrac{1}{2}hf(x_0 + H, y_m) \tag{16.5.5}$$

Gragg again showed that the error series for equations (16.5.3)–(16.5.5) contains only even powers of h, and so the method is a logical candidate for extrapolation à la Bulirsch-Stoer. We replace `mmid` by the following routine `stoerm`:

```
#include "nrutil.h"

void stoerm(float y[], float d2y[], int nv, float xs, float htot, int nstep,
    float yout[], void (*derivs)(float, float [], float []))
Stoermer's rule for integrating y'' = f(x,y) for a system of n = nv/2 equations. On input
y[1..nv] contains y in its first n elements and y' in its second n elements, all evaluated at
xs. d2y[1..nv] contains the right-hand side function f (also evaluated at xs) in its first n
elements. Its second n elements are not referenced. Also input is htot, the total step to be
taken, and nstep, the number of substeps to be used. The output is returned as yout[1..nv],
with the same storage arrangement as y. derivs is the user-supplied routine that calculates f.
{
    int i,n,neqns,nn;
    float h,h2,halfh,x,*ytemp;

    ytemp=vector(1,nv);
    h=htot/nstep;                          Stepsize this trip.
    halfh=0.5*h;
    neqns=nv/2;                            Number of equations.
    for (i=1;i<=neqns;i++) {               First step.
        n=neqns+i;
        ytemp[i]=y[i]+(ytemp[n]=h*(y[n]+halfh*d2y[i]));
    }
    x=xs+h;
    (*derivs)(x,ytemp,yout);               Use yout for temporary storage of derivatives.
    h2=h*h;
    for (nn=2;nn<=nstep;nn++) {            General step.
        for (i=1;i<=neqns;i++)
            ytemp[i] += (ytemp[(n=neqns+i)] += h2*yout[i]);
        x += h;
        (*derivs)(x,ytemp,yout);
    }
    for (i=1;i<=neqns;i++) {               Last step.
        n=neqns+i;
        yout[n]=ytemp[n]/h+halfh*yout[i];
        yout[i]=ytemp[i];
    }
    free_vector(ytemp,1,nv);
}
```

Note that for compatibility with `bsstep` the arrays `y` and `d2y` are of length $2n$ for a system of n second-order equations. The values of y are stored in the first n elements of `y`, while the first derivatives are stored in the second n elements. The right-hand side f is stored in the first n elements of the array `d2y`; the second n elements are unused. With this storage arrangement you can use `bsstep` simply by replacing the call to `mmid` with one to `stoerm` using the same arguments; just be sure that the argument `nv` of `bsstep` is set to $2n$. You should also use the more efficient sequence of stepsizes suggested by Deuflhard:

$$n = 1, 2, 3, 4, 5, \ldots \tag{16.5.6}$$

and set `KMAXX` = 12 in `bsstep`.

CITED REFERENCES AND FURTHER READING:
Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535.

16.6 Stiff Sets of Equations

As soon as one deals with more than one first-order differential equation, the possibility of a *stiff* set of equations arises. Stiffness occurs in a problem where there are two or more very different scales of the independent variable on which the dependent variables are changing. For example, consider the following set of equations [1]:

$$\begin{aligned} u' &= 998u + 1998v \\ v' &= -999u - 1999v \end{aligned} \tag{16.6.1}$$

with boundary conditions

$$u(0) = 1 \qquad v(0) = 0 \tag{16.6.2}$$

By means of the transformation

$$u = 2y - z \qquad v = -y + z \tag{16.6.3}$$

we find the solution

$$\begin{aligned} u &= 2e^{-x} - e^{-1000x} \\ v &= -e^{-x} + e^{-1000x} \end{aligned} \tag{16.6.4}$$

If we integrated the system (16.6.1) with any of the methods given so far in this chapter, the presence of the e^{-1000x} term would require a stepsize $h \ll 1/1000$ for the method to be stable (the reason for this is explained below). This is so even

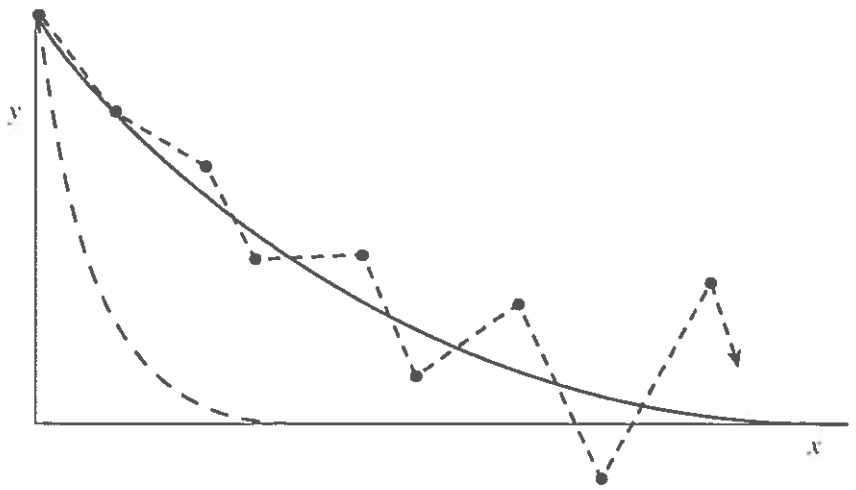

Figure 16.6.1. Example of an instability encountered in integrating a stiff equation (schematic). Here it is supposed that the equation has two solutions, shown as solid and dashed lines. Although the initial conditions are such as to give the solid solution, the stability of the integration (shown as the unstable dotted sequence of segments) is determined by the more rapidly varying dashed solution, even after that solution has effectively died away to zero. Implicit integration methods are the cure.

though the e^{-1000x} term is completely negligible in determining the values of u and v as soon as one is away from the origin (see Figure 16.6.1).

This is the generic disease of stiff equations: we are required to follow the variation in the solution on the shortest length scale to maintain stability of the integration, even though accuracy requirements allow a much larger stepsize.

To see how we might cure this problem, consider the single equation

$$y' = -cy \tag{16.6.5}$$

where $c > 0$ is a constant. The explicit (or *forward*) Euler scheme for integrating this equation with stepsize h is

$$y_{n+1} = y_n + hy'_n = (1 - ch)y_n \tag{16.6.6}$$

The method is called explicit because the new value y_{n+1} is given explicitly in terms of the old value y_n. Clearly the method is unstable if $h > 2/c$, for then $|y_n| \to \infty$ as $n \to \infty$.

The simplest cure is to resort to *implicit* differencing, where the right-hand side is evaluated at the *new* y location. In this case, we get the *backward Euler* scheme:

$$y_{n+1} = y_n + hy'_{n+1} \tag{16.6.7}$$

or

$$y_{n+1} = \frac{y_n}{1 + ch} \tag{16.6.8}$$

The method is absolutely stable: even as $h \to \infty$, $y_{n+1} \to 0$, which is in fact the correct solution of the differential equation. If we think of x as representing time, then the implicit method converges to the true equilibrium solution (i.e., the solution at late times) for large stepsizes. This nice feature of implicit methods holds only for linear systems, but even in the general case implicit methods give better stability.

Of course, we give up *accuracy* in following the evolution towards equilibrium if we use large stepsizes, but we maintain *stability*.

These considerations can easily be generalized to sets of linear equations with constant coefficients:

$$\mathbf{y}' = -\mathbf{C} \cdot \mathbf{y} \tag{16.6.9}$$

where $\mathbf{C}$ is a positive definite matrix. Explicit differencing gives

$$\mathbf{y}_{n+1} = (\mathbf{1} - \mathbf{C}h) \cdot \mathbf{y}_n \tag{16.6.10}$$

Now a matrix $\mathbf{A}^n$ tends to zero as $n \to \infty$ only if the largest eigenvalue of $\mathbf{A}$ has magnitude less than unity. Thus $\mathbf{y}_n$ is bounded as $n \to \infty$ only if the largest eigenvalue of $\mathbf{1} - \mathbf{C}h$ is less than 1, or in other words

$$h < \frac{2}{\lambda_{\max}} \tag{16.6.11}$$

where $\lambda_{\max}$ is the largest eigenvalue of $\mathbf{C}$.

On the other hand, implicit differencing gives

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{y}'_{n+1} \tag{16.6.12}$$

or

$$\mathbf{y}_{n+1} = (\mathbf{1} + \mathbf{C}h)^{-1} \cdot \mathbf{y}_n \tag{16.6.13}$$

If the eigenvalues of $\mathbf{C}$ are λ, then the eigenvalues of $(\mathbf{1} + \mathbf{C}h)^{-1}$ are $(1 + \lambda h)^{-1}$, which has magnitude less than one for all h. (Recall that all the eigenvalues of a positive definite matrix are nonnegative.) Thus the method is stable for all stepsizes h. The penalty we pay for this stability is that we are required to invert a matrix at each step.

Not all equations are linear with constant coefficients, unfortunately! For the system

$$\mathbf{y}' = \mathbf{f}(\mathbf{y}) \tag{16.6.14}$$

implicit differencing gives

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(\mathbf{y}_{n+1}) \tag{16.6.15}$$

In general this is some nasty set of nonlinear equations that has to be solved iteratively at each step. Suppose we try linearizing the equations, as in Newton's method:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left[\mathbf{f}(\mathbf{y}_n) + \left.\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right|_{\mathbf{y}_n} \cdot (\mathbf{y}_{n+1} - \mathbf{y}_n)\right] \tag{16.6.16}$$

Here $\partial \mathbf{f}/\partial \mathbf{y}$ is the matrix of the partial derivatives of the right-hand side (the Jacobian matrix). Rearrange equation (16.6.16) into the form

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot \mathbf{f}(\mathbf{y}_n) \tag{16.6.17}$$

If h is not too big, only one iteration of Newton's method may be accurate enough to solve equation (16.6.15) using equation (16.6.17). In other words, at each step we have to invert the matrix

$$\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}} \tag{16.6.18}$$

to find $\mathbf{y}_{n+1}$. Solving implicit methods by linearization is called a "semi-implicit" method, so equation (16.6.17) is the *semi-implicit Euler method*. It is not guaranteed to be stable, but it usually is, because the behavior is locally similar to the case of a constant matrix $\mathbf{C}$ described above.

So far we have dealt only with implicit methods that are first-order accurate. While these are very robust, most problems will benefit from higher-order methods. There are three important classes of higher-order methods for stiff systems:

- Generalizations of the Runge-Kutta method, of which the most useful are the Rosenbrock methods. The first practical implementation of these ideas was by Kaps and Rentrop, and so these methods are also called Kaps-Rentrop methods.
- Generalizations of the Bulirsch-Stoer method, in particular a semi-implicit extrapolation method due to Bader and Deuflhard.
- Predictor-corrector methods, most of which are descendants of Gear's backward differentiation method.

We shall give implementations of the first two methods. Note that systems where the right-hand side depends explicitly on x, $\mathbf{f}(\mathbf{y}, x)$, can be handled by adding x to the list of dependent variables so that the system to be solved is

$$\begin{pmatrix} \mathbf{y} \\ x \end{pmatrix}' = \begin{pmatrix} \mathbf{f} \\ 1 \end{pmatrix} \tag{16.6.19}$$

In both the routines to be given in this section, we have explicitly carried out this replacement for you, so the routines can handle right-hand sides of the form $\mathbf{f}(\mathbf{y}, x)$ without any special effort on your part.

We now mention an important point: *It is absolutely crucial to scale your variables properly when integrating stiff problems with automatic stepsize adjustment.* As in our nonstiff routines, you will be asked to supply a vector $\mathbf{y}_{\text{scal}}$ with which the error is to be scaled. For example, to get constant fractional errors, simply set $\mathbf{y}_{\text{scal}} = |\mathbf{y}|$. You can get constant absolute errors relative to some maximum values by setting $\mathbf{y}_{\text{scal}}$ equal to those maximum values. In stiff problems, there are often strongly decreasing pieces of the solution which you are not particularly interested in following once they are small. You can control the relative error above some threshold $\mathbf{C}$ and the absolute error below the threshold by setting

$$\mathbf{y}_{\text{scal}} = \max(\mathbf{C}, |\mathbf{y}|) \tag{16.6.20}$$

If you are using appropriate nondimensional units, then each component of $\mathbf{C}$ should be of order unity. If you are not sure what values to take for $\mathbf{C}$, simply try setting each component equal to unity. *We strongly advocate the choice (16.6.20) for stiff problems.*

One final warning: Solving stiff problems can sometimes lead to catastrophic precision loss. Be alert for situations where double precision is necessary.

Rosenbrock Methods

These methods have the advantage of being relatively simple to understand and implement. For moderate accuracies ($\epsilon \lesssim 10^{-4} - 10^{-5}$ in the error criterion) and moderate-sized systems ($N \lesssim 10$), they are competitive with the more complicated algorithms. For more stringent parameters, Rosenbrock methods remain reliable; they merely become less efficient than competitors like the semi-implicit extrapolation method (see below).

A Rosenbrock method seeks a solution of the form

$$\mathbf{y}(x_0 + h) = \mathbf{y}_0 + \sum_{i=1}^{s} c_i \mathbf{k}_i \tag{16.6.21}$$

where the corrections $\mathbf{k}_i$ are found by solving s linear equations that generalize the structure in (16.6.17):

$$(\mathbf{1} - \gamma h \mathbf{f}') \cdot \mathbf{k}_i = h\mathbf{f}\left(\mathbf{y}_0 + \sum_{j=1}^{i-1} \alpha_{ij}\mathbf{k}_j\right) + h\mathbf{f}' \cdot \sum_{j=1}^{i-1} \gamma_{ij}\mathbf{k}_j, \qquad i = 1, \ldots, s \tag{16.6.22}$$

Here we denote the Jacobian matrix by $\mathbf{f}'$. The coefficients γ, c_i, α_{ij}, and γ_{ij} are fixed constants independent of the problem. If $\gamma = \gamma_{ij} = 0$, this is simply a Runge-Kutta scheme. Equations (16.6.22) can be solved successively for $\mathbf{k}_1, \mathbf{k}_2, \ldots$.

Crucial to the success of a stiff integration scheme is an automatic stepsize adjustment algorithm. Kaps and Rentrop [2] discovered an *embedded* or Runge-Kutta-Fehlberg method as described in §16.2: Two estimates of the form (16.6.21) are computed, the "real" one $\mathbf{y}$ and a lower-order estimate $\widehat{\mathbf{y}}$ with different coefficients $\hat{c}_i, i = 1, \ldots, \hat{s}$, where $\hat{s} < s$ but the $\mathbf{k}_i$ are the same. The difference between $\mathbf{y}$ and $\widehat{\mathbf{y}}$ leads to an estimate of the local truncation error, which can then be used for stepsize control. Kaps and Rentrop showed that the smallest value of s for which embedding is possible is $s = 4$, $\hat{s} = 3$, leading to a fourth-order method.

To minimize the matrix-vector multiplications on the right-hand side of (16.6.22), we rewrite the equations in terms of quantities

$$\mathbf{g}_i = \sum_{j=1}^{i-1} \gamma_{ij}\mathbf{k}_j + \gamma \mathbf{k}_i \tag{16.6.23}$$

The equations then take the form

$$\begin{aligned}
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_1 &= \mathbf{f}(\mathbf{y}_0) \\
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_2 &= \mathbf{f}(\mathbf{y}_0 + a_{21}\mathbf{g}_1) + c_{21}\mathbf{g}_1/h \\
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_3 &= \mathbf{f}(\mathbf{y}_0 + a_{31}\mathbf{g}_1 + a_{32}\mathbf{g}_2) + (c_{31}\mathbf{g}_1 + c_{32}\mathbf{g}_2)/h \\
(\mathbf{1}/\gamma h - \mathbf{f}') \cdot \mathbf{g}_4 &= \mathbf{f}(\mathbf{y}_0 + a_{41}\mathbf{g}_1 + a_{42}\mathbf{g}_2 + a_{43}\mathbf{g}_3) + (c_{41}\mathbf{g}_1 + c_{42}\mathbf{g}_2 + c_{43}\mathbf{g}_3)/h
\end{aligned} \tag{16.6.24}$$

In our implementation `stiff` of the Kaps-Rentrop algorithm, we have carried out the replacement (16.6.19) explicitly in equations (16.6.24), so you need not concern yourself about it. Simply provide a routine (called `derivs` in `stiff`) that returns $\mathbf{f}$ (called `dydx`) as a function of x and $\mathbf{y}$. Also supply a routine `jacobn` that returns $\mathbf{f}'$ (`dfdy`) and $\partial \mathbf{f}/\partial x$ (`dfdx`) as functions of x and $\mathbf{y}$. If x does not occur explicitly on the right-hand side, then `dfdx` will be zero. Usually the Jacobian matrix will be available to you by analytic differentiation of the right-hand side $\mathbf{f}$. If not, your routine will have to compute it by numerical differencing with appropriate increments $\Delta \mathbf{y}$.

Kaps and Rentrop gave two different sets of parameters, which have slightly different stability properties. Several other sets have been proposed. Our default choice is that of Shampine [3], but we also give you one of the Kaps-Rentrop sets as an option. Some proposed parameter sets require function evaluations outside the domain of integration; we prefer to avoid that complication.

The calling sequence of `stiff` is exactly the same as the nonstiff routines given earlier in this chapter. It is thus "plug-compatible" with them in the general ODE integrating routine

odeint. This compatibility requires, unfortunately, one slight anomaly: While the user-supplied routine derivs is a dummy argument (which can therefore have any actual name), the other user-supplied routine is *not* an argument and must be named (exactly) jacobn.

stiff begins by saving the initial values, in case the step has to be repeated because the error tolerance is exceeded. The linear equations (16.6.24) are solved by first computing the LU decomposition of the matrix $1/\gamma h - \mathbf{f}'$ using the routine ludcmp. Then the four $\mathbf{g}_i$ are found by back-substitution of the four different right-hand sides using lubksb. Note that each step of the integration requires one call to jacobn and three calls to derivs (one call to get dydx before calling stiff, and two calls inside stiff). The reason only three calls are needed and not four is that the parameters have been chosen so that the last two calls in equation (16.6.24) are done with the same arguments. Counting the evaluation of the Jacobian matrix as roughly equivalent to N evaluations of the right-hand side $\mathbf{f}$, we see that the Kaps-Rentrop scheme involves about $N + 3$ function evaluations per step. Note that if N is large and the Jacobian matrix is sparse, you should replace the LU decomposition by a suitable sparse matrix procedure.

Stepsize control depends on the fact that

$$\begin{aligned} \mathbf{y}_{\text{exact}} &= \mathbf{y} + O(h^5) \\ \mathbf{y}_{\text{exact}} &= \hat{\mathbf{y}} + O(h^4) \end{aligned} \tag{16.6.25}$$

Thus

$$|\mathbf{y} - \hat{\mathbf{y}}| = O(h^4) \tag{16.6.26}$$

Referring back to the steps leading from equation (16.2.4) to equation (16.2.10), we see that the new stepsize should be chosen as in equation (16.2.10) but with the exponents 1/4 and 1/5 replaced by 1/3 and 1/4, respectively. Also, experience shows that it is wise to prevent too large a stepsize change in one step, otherwise we will probably have to undo the large change in the next step. We adopt 0.5 and 1.5 as the maximum allowed decrease and increase of h in one step.

```
#include <math.h>
#include "nrutil.h"
#define SAFETY 0.9
#define GROW 1.5
#define PGROW -0.25
#define SHRNK 0.5
#define PSHRNK (-1.0/3.0)
#define ERRCON 0.1296
#define MAXTRY 40
```

Here NMAX is the maximum value of n; GROW and SHRNK are the largest and smallest factors by which stepsize can change in one step; ERRCON equals (GROW/SAFETY) raised to the power (1/PGROW) and handles the case when errmax $\simeq 0$.

```
#define GAM (1.0/2.0)
#define A21 2.0
#define A31 (48.0/25.0)
#define A32 (6.0/25.0)
#define C21 -8.0
#define C31 (372.0/25.0)
#define C32 (12.0/5.0)
#define C41 (-112.0/125.0)
#define C42 (-54.0/125.0)
#define C43 (-2.0/5.0)
#define B1 (19.0/9.0)
#define B2 (1.0/2.0)
#define B3 (25.0/108.0)
#define B4 (125.0/108.0)
#define E1 (17.0/54.0)
#define E2 (7.0/36.0)
#define E3 0.0
#define E4 (125.0/108.0)
```

```
#define C1X (1.0/2.0)
#define C2X (-3.0/2.0)
#define C3X (121.0/50.0)
#define C4X (29.0/250.0)
#define A2X 1.0
#define A3X (3.0/5.0)

void stiff(float y[], float dydx[], int n, float *x, float htry, float eps,
    float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []))
```

Fourth-order Rosenbrock step for integrating stiff o.d.e.'s, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector `y[1..n]` and its derivative `dydx[1..n]` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal[1..n]` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is a user-supplied routine that computes the derivatives of the right-hand side with respect to `x`, while `jacobn` (a fixed name) is a user-supplied routine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of `y`.

```
{
    void jacobn(float x, float y[], float dfdx[], float **dfdy, int n);
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,j,jtry,*indx;
    float d,errmax,h,xsav,**a,*dfdx,**dfdy,*dysav,*err;
    float *g1,*g2,*g3,*g4,*ysav;

    indx=ivector(1,n);
    a=matrix(1,n,1,n);
    dfdx=vector(1,n);
    dfdy=matrix(1,n,1,n);
    dysav=vector(1,n);
    err=vector(1,n);
    g1=vector(1,n);
    g2=vector(1,n);
    g3=vector(1,n);
    g4=vector(1,n);
    ysav=vector(1,n);
    xsav=(*x);                          Save initial values.
    for (i=1;i<=n;i++) {
        ysav[i]=y[i];
        dysav[i]=dydx[i];
    }
    jacobn(xsav,ysav,dfdx,dfdy,n);
    The user must supply this routine to return the n-by-n matrix dfdy and the vector dfdx.
    h=htry;                             Set stepsize to the initial trial value.
    for (jtry=1;jtry<=MAXTRY;jtry++) {
        for (i=1;i<=n;i++) {            Set up the matrix 1 − γhf′.
            for (j=1;j<=n;j++) a[i][j] = -dfdy[i][j];
            a[i][i] += 1.0/(GAM*h);
        }
        ludcmp(a,n,indx,&d);            LU decomposition of the matrix.
        for (i=1;i<=n;i++)              Set up right-hand side for g1.
            g1[i]=dysav[i]+h*C1X*dfdx[i];
        lubksb(a,n,indx,g1);            Solve for g1.
        for (i=1;i<=n;i++)              Compute intermediate values of y and x.
            y[i]=ysav[i]+A21*g1[i];
        *x=xsav+A2X*h;
        (*derivs)(*x,y,dydx);           Compute dydx at the intermediate values.
        for (i=1;i<=n;i++)              Set up right-hand side for g2.
            g2[i]=dydx[i]+h*C2X*dfdx[i]+C21*g1[i]/h;
        lubksb(a,n,indx,g2);            Solve for g2.
        for (i=1;i<=n;i++)              Compute intermediate values of y and x.
            y[i]=ysav[i]+A31*g1[i]+A32*g2[i];
```

```
        *x=xsav+A3X*h;
        (*derivs)(*x,y,dydx);          Compute dydx at the intermediate values.
        for (i=1;i<=n;i++)             Set up right-hand side for g3.
            g3[i]=dydx[i]+h*C3X*dfdx[i]+(C31*g1[i]+C32*g2[i])/h;
        lubksb(a,n,indx,g3);           Solve for g3.
        for (i=1;i<=n;i++)             Set up right-hand side for g4.
            g4[i]=dydx[i]+h*C4X*dfdx[i]+(C41*g1[i]+C42*g2[i]+C43*g3[i])/h;
        lubksb(a,n,indx,g4);           Solve for g4.
        for (i=1;i<=n;i++) {           Get fourth-order estimate of y and error estimate.
            y[i]=ysav[i]+B1*g1[i]+B2*g2[i]+B3*g3[i]+B4*g4[i];
            err[i]=E1*g1[i]+E2*g2[i]+E3*g3[i]+E4*g4[i];
        }
        *x=xsav+h;
        if (*x == xsav) nrerror("stepsize not significant in stiff");
        errmax=0.0;                    Evaluate accuracy.
        for (i=1;i<=n;i++) errmax=FMAX(errmax,fabs(err[i]/yscal[i]));
        errmax /= eps;                 Scale relative to required tolerance.
        if (errmax <= 1.0) {           Step succeeded. Compute size of next step and re-
            *hdid=h;                       turn.
            *hnext=(errmax > ERRCON ? SAFETY*h*pow(errmax,PGROW) : GROW*h);
            free_vector(ysav,1,n);
            free_vector(g4,1,n);
            free_vector(g3,1,n);
            free_vector(g2,1,n);
            free_vector(g1,1,n);
            free_vector(err,1,n);
            free_vector(dysav,1,n);
            free_matrix(dfdy,1,n,1,n);
            free_vector(dfdx,1,n);
            free_matrix(a,1,n,1,n);
            free_ivector(indx,1,n);
            return;
        } else {                       Truncation error too large, reduce stepsize.
            *hnext=SAFETY*h*pow(errmax,PSHRNK);
            h=(h >= 0.0 ? FMAX(*hnext,SHRNK*h) : FMIN(*hnext,SHRNK*h));
        }
    }                                  Go back and re-try step.
    nrerror("exceeded MAXTRY in stiff");
}
```

Here are the Kaps-Rentrop parameters, which can be substituted for those of Shampine simply by replacing the `#define` statements:

```
#define GAM 0.231
#define A21 2.0
#define A31 4.52470820736
#define A32 4.16352878860
#define C21 -5.07167533877
#define C31 6.02015272865
#define C32 0.159750684673
#define C41 -1.856343618677
#define C42 -8.50538085819
#define C43 -2.08407513602
#define B1 3.95750374663
#define B2 4.62489238836
#define B3 0.617477263873
#define B4 1.282612945268
#define E1 -2.30215540292
#define E2 -3.07363448539
#define E3 0.873280801802
#define E4 1.282612945268
#define C1X GAM
```

```
#define C2X -0.396296677520e-01
#define C3X 0.550778939579
#define C4X -0.553509845700e-01
#define A2X 0.462
#define A3X 0.880208333333
```

As an example of how `stiff` is used, one can solve the system

$$\begin{aligned} y_1' &= -.013y_1 - 1000y_1y_3 \\ y_2' &= -2500y_2y_3 \\ y_3' &= -.013y_1 - 1000y_1y_3 - 2500y_2y_3 \end{aligned} \tag{16.6.27}$$

with initial conditions

$$y_1(0) = 1, \qquad y_2(0) = 1, \qquad y_3(0) = 0 \tag{16.6.28}$$

(This is test problem D4 in [4].) We integrate the system up to $x = 50$ with an initial stepsize of $h = 2.9 \times 10^{-4}$ using `odeint`. The components of $\mathbf{C}$ in (16.6.20) are all set to unity. The routines `derivs` and `jacobn` for this problem are given below. Even though the ratio of largest to smallest decay constants for this problem is around 10^6, `stiff` succeeds in integrating this set in only 29 steps with $\epsilon = 10^{-4}$. By contrast, the Runge-Kutta routine `rkqs` requires 51,012 steps!

```
void jacobn(float x, float y[], float dfdx[], float **dfdy, int n)
{
    int i;

    for (i=1;i<=n;i++) dfdx[i]=0.0;
    dfdy[1][1] = -0.013-1000.0*y[3];
    dfdy[1][2]=0.0;
    dfdy[1][3] = -1000.0*y[1];
    dfdy[2][1]=0.0;
    dfdy[2][2] = -2500.0*y[3];
    dfdy[2][3] = -2500.0*y[2];
    dfdy[3][1] = -0.013-1000.0*y[3];
    dfdy[3][2] = -2500.0*y[3];
    dfdy[3][3] = -1000.0*y[1]-2500.0*y[2];
}

void derivs(float x, float y[], float dydx[])
{
    dydx[1] = -0.013*y[1]-1000.0*y[1]*y[3];
    dydx[2] = -2500.0*y[2]*y[3];
    dydx[3] = -0.013*y[1]-1000.0*y[1]*y[3]-2500.0*y[2]*y[3];
}
```

Semi-implicit Extrapolation Method

The Bulirsch-Stoer method, which discretizes the differential equation using the modified midpoint rule, does not work for stiff problems. Bader and Deuflhard [5] discovered a semi-implicit discretization that works very well and that lends itself to extrapolation exactly as in the original Bulirsch-Stoer method.

The starting point is an implicit form of the midpoint rule:

$$\mathbf{y}_{n+1} - \mathbf{y}_{n-1} = 2h\mathbf{f}\left(\frac{\mathbf{y}_{n+1} + \mathbf{y}_{n-1}}{2}\right) \tag{16.6.29}$$

Convert this equation into semi-implicit form by linearizing the right-hand side about $\mathbf{f}(\mathbf{y}_n)$. The result is the *semi-implicit midpoint rule*:

$$\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right] \cdot \mathbf{y}_{n+1} = \left[\mathbf{1} + h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right] \cdot \mathbf{y}_{n-1} + 2h\left[\mathbf{f}(\mathbf{y}_n) - \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \cdot \mathbf{y}_n\right] \tag{16.6.30}$$

It is used with a special first step, the semi-implicit Euler step (16.6.17), and a special "smoothing" last step in which the last $\mathbf{y}_n$ is replaced by

$$\bar{\mathbf{y}}_n \equiv \tfrac{1}{2}(\mathbf{y}_{n+1} + \mathbf{y}_{n-1}) \tag{16.6.31}$$

Bader and Deuflhard showed that the error series for this method once again involves only even powers of h.

For practical implementation, it is better to rewrite the equations using $\Delta_k \equiv \mathbf{y}_{k+1} - \mathbf{y}_k$. With $h = H/m$, start by calculating

$$\begin{aligned} \Delta_0 &= \left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot h\mathbf{f}(\mathbf{y}_0) \\ \mathbf{y}_1 &= \mathbf{y}_0 + \Delta_0 \end{aligned} \tag{16.6.32}$$

Then for $k = 1, \ldots, m-1$, set

$$\begin{aligned} \Delta_k &= \Delta_{k-1} + 2\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot [h\mathbf{f}(\mathbf{y}_k) - \Delta_{k-1}] \\ \mathbf{y}_{k+1} &= \mathbf{y}_k + \Delta_k \end{aligned} \tag{16.6.33}$$

Finally compute

$$\begin{aligned} \Delta_m &= \left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right]^{-1} \cdot [h\mathbf{f}(\mathbf{y}_m) - \Delta_{m-1}] \\ \bar{\mathbf{y}}_m &= \mathbf{y}_m + \Delta_m \end{aligned} \tag{16.6.34}$$

It is easy to incorporate the replacement (16.6.19) in the above formulas. The additional terms in the Jacobian that come from $\partial \mathbf{f}/\partial x$ all cancel out of the semi-implicit midpoint rule (16.6.30). In the special first step (16.6.17), and in the corresponding equation (16.6.32), the term $h\mathbf{f}$ becomes $h\mathbf{f} + h^2\partial \mathbf{f}/\partial x$. The remaining equations are all unchanged.

This algorithm is implemented in the routine `simpr`:

```
#include "nrutil.h"

void simpr(float y[], float dydx[], float dfdx[], float **dfdy, int n,
    float xs, float htot, int nstep, float yout[],
    void (*derivs)(float, float [], float []))
Performs one step of semi-implicit midpoint rule. Input are the dependent variable y[1..n], its
derivative dydx[1..n], the derivative of the right-hand side with respect to x, dfdx[1..n],
and the Jacobian dfdy[1..n][1..n] at xs. Also input are htot, the total step to be taken,
and nstep, the number of substeps to be used. The output is returned as yout[1..n].
derivs is the user-supplied routine that calculates dydx.
{
    void lubksb(float **a, int n, int *indx, float b[]);
    void ludcmp(float **a, int n, int *indx, float *d);
    int i,j,nn,*indx;
    float d,h,x,**a,*del,*ytemp;

    indx=ivector(1,n);
    a=matrix(1,n,1,n);
    del=vector(1,n);
    ytemp=vector(1,n);
    h=htot/nstep;                          Stepsize this trip.
    for (i=1;i<=n;i++) {                   Set up the matrix 1 - hf'.
        for (j=1;j<=n;j++) a[i][j] = -h*dfdy[i][j];
```

```
        ++a[i][i];
    }
    ludcmp(a,n,indx,&d);                LU decomposition of the matrix.
    for (i=1;i<=n;i++)                  Set up right-hand side for first step. Use yout
        yout[i]=h*(dydx[i]+h*dfdx[i]);      for temporary storage.
    lubksb(a,n,indx,yout);
    for (i=1;i<=n;i++)                  First step.
        ytemp[i]=y[i]+(del[i]=yout[i]);
    x=xs+h;
    (*derivs)(x,ytemp,yout);            Use yout for temporary storage of derivatives.
    for (nn=2;nn<=nstep;nn++) {         General step.
        for (i=1;i<=n;i++)              Set up right-hand side for general step.
            yout[i]=h*yout[i]-del[i];
        lubksb(a,n,indx,yout);
        for (i=1;i<=n;i++)
            ytemp[i] += (del[i] += 2.0*yout[i]);
        x += h;
        (*derivs)(x,ytemp,yout);
    }
    for (i=1;i<=n;i++)                  Set up right-hand side for last step.
        yout[i]=h*yout[i]-del[i];
    lubksb(a,n,indx,yout);
    for (i=1;i<=n;i++)                  Take last step.
        yout[i] += ytemp[i];
    free_vector(ytemp,1,n);
    free_vector(del,1,n);
    free_matrix(a,1,n,1,n);
    free_ivector(indx,1,n);
}
```

The routine `simpr` is intended to be used in a routine `stifbs` that is almost exactly the same as `bsstep`. The only differences are:

- The stepsize sequence is

$$n = 2, 6, 10, 14, 22, 34, 50, \ldots, \tag{16.6.35}$$

  where each member differs from its predecessor by the smallest multiple of 4 that makes the ratio of successive terms be $\leq \frac{5}{7}$. The parameter `KMAXX` is taken to be 7.
- The work per unit step now includes the cost of Jacobian evaluations as well as function evaluations. We count one Jacobian evaluation as equivalent to N function evaluations, where N is the number of equations.
- Once again the user-supplied routine `derivs` is a dummy argument and so can have any name. However, to maintain "plug-compatibility" with `rkqs`, `bsstep` and `stiff`, the routine `jacobn` is not an argument and *must* have exactly this name. It is called once per step to return $\mathbf{f}'$ (`dfdy`) and $\partial \mathbf{f}/\partial x$ (`dfdx`) as functions of x and $\mathbf{y}$.

Here is the routine, with comments pointing out only the differences from `bsstep`:

```
#include <math.h>
#include "nrutil.h"
#define KMAXX 7
#define IMAXX (KMAXX+1)
#define SAFE1 0.25
#define SAFE2 0.7
#define REDMAX 1.0e-5
#define REDMIN 0.7
#define TINY 1.0e-30
#define SCALMX 0.1

float **d,*x;

void stifbs(float y[], float dydx[], int nv, float *xx, float htry, float eps,
    float yscal[], float *hdid, float *hnext,
    void (*derivs)(float, float [], float []))
```

Semi-implicit extrapolation step for integrating stiff o.d.e.'s, with monitoring of local truncation error to adjust stepsize. Input are the dependent variable vector `y[1..nv]` and its derivative `dydx[1..nv]` at the starting value of the independent variable `x`. Also input are the stepsize to be attempted `htry`, the required accuracy `eps`, and the vector `yscal[1..nv]` against which the error is scaled. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize. `derivs` is a user-supplied routine that computes the derivatives of the right-hand side with respect to `x`, while `jacobn` (a fixed name) is a user-supplied routine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of `y`. Be sure to set `htry` on successive steps to the value of `hnext` returned from the previous step, as is the case if the routine is called by `odeint`.

```
{
    void jacobn(float x, float y[], float dfdx[], float **dfdy, int n);
    void simpr(float y[], float dydx[], float dfdx[], float **dfdy,
        int n, float xs, float htot, int nstep, float yout[],
        void (*derivs)(float, float [], float []));
    void pzextr(int iest, float xest, float yest[], float yz[], float dy[],
        int nv);
    int i,iq,k,kk,km;
    static int first=1,kmax,kopt,nvold = -1;
    static float epsold = -1.0,xnew;
    float eps1,errmax,fact,h,red,scale,work,wrkmin,xest;
    float *dfdx,**dfdy,*err,*yerr,*ysav,*yseq;
    static float a[IMAXX+1];
    static float alf[KMAXX+1][KMAXX+1];
    static int nseq[IMAXX+1]={0,2,6,10,14,22,34,50,70};      Sequence is different from
    int reduct,exitflag=0;                                       bsstep.

    d=matrix(1,nv,1,KMAXX);
    dfdx=vector(1,nv);
    dfdy=matrix(1,nv,1,nv);
    err=vector(1,KMAXX);
    x=vector(1,KMAXX);
    yerr=vector(1,nv);
    ysav=vector(1,nv);
    yseq=vector(1,nv);
    if(eps != epsold || nv != nvold) {          Reinitialize also if nv has changed.
        *hnext = xnew = -1.0e29;
        eps1=SAFE1*eps;
        a[1]=nseq[1]+1;
        for (k=1;k<=KMAXX;k++) a[k+1]=a[k]+nseq[k+1];
        for (iq=2;iq<=KMAXX;iq++) {
            for (k=1;k<iq;k++)
                alf[k][iq]=pow(eps1,((a[k+1]-a[iq+1])/
                    ((a[iq+1]-a[1]+1.0)*(2*k+1))));
        }
        epsold=eps;
        nvold=nv;                               Save nv.
        a[1] += nv;                             Add cost of Jacobian evaluations to work
        for (k=1;k<=KMAXX;k++) a[k+1]=a[k]+nseq[k+1];      coefficients.
        for (kopt=2;kopt<KMAXX;kopt++)
            if (a[kopt+1] > a[kopt]*alf[kopt-1][kopt]) break;
        kmax=kopt;
    }
    h=htry;
    for (i=1;i<=nv;i++) ysav[i]=y[i];
    jacobn(*xx,y,dfdx,dfdy,nv);                 Evaluate Jacobian.
    if (*xx != xnew || h != (*hnext)) {
        first=1;
        kopt=kmax;
    }
    reduct=0;
    for (;;) {
        for (k=1;k<=kmax;k++) {
```

```
            xnew=(*xx)+h;
            if (xnew == (*xx)) nrerror("step size underflow in stifbs");
            simpr(ysav,dydx,dfdx,dfdy,nv,*xx,h,nseq[k],yseq,derivs);
            Semi-implicit midpoint rule.
            xest=SQR(h/nseq[k]);                The rest of the routine is identical to
            pzextr(k,xest,yseq,y,yerr,nv);        bsstep.
            if (k != 1) {
                errmax=TINY;
                for (i=1;i<=nv;i++) errmax=FMAX(errmax,fabs(yerr[i]/yscal[i]));
                errmax /= eps;
                km=k-1;
                err[km]=pow(errmax/SAFE1,1.0/(2*km+1));
            }
            if (k != 1 && (k >= kopt-1 || first)) {
                if (errmax < 1.0) {
                    exitflag=1;
                    break;
                }
                if (k == kmax || k == kopt+1) {
                    red=SAFE2/err[km];
                    break;
                }
                else if (k == kopt && alf[kopt-1][kopt] < err[km]) {
                        red=1.0/err[km];
                        break;
                    }
                else if (kopt == kmax && alf[km][kmax-1] < err[km]) {
                        red=alf[km][kmax-1]*SAFE2/err[km];
                        break;
                    }
                else if (alf[km][kopt] < err[km]) {
                    red=alf[km][kopt-1]/err[km];
                    break;
                }
            }
        }
        if (exitflag) break;
        red=FMIN(red,REDMIN);
        red=FMAX(red,REDMAX);
        h *= red;
        reduct=1;
    }
    *xx=xnew;
    *hdid=h;
    first=0;
    wrkmin=1.0e35;
    for (kk=1;kk<=km;kk++) {
        fact=FMAX(err[kk],SCALMX);
        work=fact*a[kk+1];
        if (work < wrkmin) {
            scale=fact;
            wrkmin=work;
            kopt=kk+1;
        }
    }
    *hnext=h/scale;
    if (kopt >= k && kopt != kmax && !reduct) {
        fact=FMAX(scale/alf[kopt-1][kopt],SCALMX);
        if (a[kopt+1]*fact <= wrkmin) {
            *hnext=h/fact;
            kopt++;
        }
    }
    free_vector(yseq,1,nv);
```

```
    free_vector(ysav,1,nv);
    free_vector(yerr,1,nv);
    free_vector(x,1,KMAXX);
    free_vector(err,1,KMAXX);
    free_matrix(dfdy,1,nv,1,nv);
    free_vector(dfdx,1,nv);
    free_matrix(d,1,nv,1,KMAXX);
}
```

The routine `stifbs` is an excellent routine for all stiff problems, competitive with the best Gear-type routines. `stiff` is comparable in execution time for moderate N and $\epsilon \lesssim 10^{-4}$. By the time $\epsilon \sim 10^{-8}$, `stifbs` is roughly an order of magnitude faster. There are further improvements that could be applied to `stifbs` to make it even more robust. For example, very occasionally `ludcmp` in `simpr` will encounter a singular matrix. You could arrange for the stepsize to be reduced, say by a factor of the current `nseq[k]`. There are also certain stability restrictions on the stepsize that come into play on some problems. For a discussion of how to implement these automatically, see [6].

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall). [1]

Kaps, P., and Rentrop, P. 1979, *Numerische Mathematik*, vol. 33, pp. 55–68. [2]

Shampine, L.F. 1982, *ACM Transactions on Mathematical Software*, vol. 8, pp. 93–113. [3]

Enright, W.H., and Pryce, J.D. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 1–27. [4]

Bader, G., and Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 373–398. [5]

Deuflhard, P. 1983, *Numerische Mathematik*, vol. 41, pp. 399–422.

Deuflhard, P. 1985, *SIAM Review*, vol. 27, pp. 505–535.

Deuflhard, P. 1987, "Uniqueness Theorems for Stiff ODE Initial Value Problems," *Preprint SC-87-3* (Berlin: Konrad Zuse Zentrum für Informationstechnik). [6]

Enright, W.H., Hull, T.E., and Lindberg, B. 1975, *BIT*, vol. 15, pp. 10–48.

Wanner, G. 1988, in *Numerical Analysis 1987*, Pitman Research Notes in Mathematics, vol. 170, D.F. Griffiths and G.A. Watson, eds. (Harlow, Essex, U.K.: Longman Scientific and Technical).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag).

16.7 Multistep, Multivalue, and Predictor-Corrector Methods

The terms multistep and multivalue describe two different ways of implementing essentially the same integration technique for ODEs. Predictor-corrector is a particular subcategrory of these methods — in fact, the most widely used. Accordingly, the name predictor-corrector is often loosely used to denote all these methods.

We suspect that predictor-corrector integrators have had their day, and that they are no longer the method of choice for most problems in ODEs. For high-precision applications, or applications where evaluations of the right-hand sides are expensive, Bulirsch-Stoer dominates. For convenience, or for low precision, adaptive-stepsize Runge-Kutta dominates. Predictor-corrector methods have been, we think, squeezed

out in the middle. There is possibly only one exceptional case: high-precision solution of very smooth equations with very complicated right-hand sides, as we will describe later.

Nevertheless, these methods have had a long historical run. Textbooks are full of information on them, and there are a lot of standard ODE programs around that are based on predictor-corrector methods. Many capable researchers have a lot of experience with predictor-corrector routines, and they see no reason to make a precipitous change of habit. It is not a bad idea for you to be familiar with the principles involved, and even with the sorts of bookkeeping details that are the bane of these methods. Otherwise there will be a big surprise in store when you first have to fix a problem in a predictor-corrector routine.

Let us first consider the multistep approach. Think about how integrating an ODE is different from finding the integral of a function: For a function, the integrand has a known dependence on the independent variable x, and can be evaluated at will. For an ODE, the "integrand" is the right-hand side, which depends both on x and on the dependent variables y. Thus to advance the solution of $y' = f(x, y)$ from x_n to x, we have

$$y(x) = y_n + \int_{x_n}^{x} f(x', y)\, dx' \tag{16.7.1}$$

In a single-step method like Runge-Kutta or Bulirsch-Stoer, the value y_{n+1} at x_{n+1} depends only on y_n. In a multistep method, we approximate $f(x, y)$ by a polynomial passing through *several* previous points $x_n, x_{n-1}, \ldots$ and possibly also through x_{n+1}. The result of evaluating the integral (16.7.1) at $x = x_{n+1}$ is then of the form

$$y_{n+1} = y_n + h(\beta_0 y'_{n+1} + \beta_1 y'_n + \beta_2 y'_{n-1} + \beta_3 y'_{n-2} + \cdots) \tag{16.7.2}$$

where y'_n denotes $f(x_n, y_n)$, and so on. If $\beta_0 = 0$, the method is explicit; otherwise it is implicit. The order of the method depends on how many previous steps we use to get each new value of y.

Consider how we might solve an implicit formula of the form (16.7.2) for y_{n+1}. Two methods suggest themselves: *functional iteration* and *Newton's method*. In functional iteration, we take some initial guess for y_{n+1}, insert it into the right-hand side of (16.7.2) to get an updated value of y_{n+1}, insert this updated value back into the right-hand side, and continue iterating. But how are we to get an initial guess for y_{n+1}? Easy! Just use some *explicit* formula of the same form as (16.7.2). This is called the *predictor step*. In the predictor step we are essentially *extrapolating* the polynomial fit to the derivative from the previous points to the new point x_{n+1} and then doing the integral (16.7.1) in a Simpson-like manner from x_n to x_{n+1}. The subsequent Simpson-like integration, using the prediction step's value of y_{n+1} to *interpolate* the derivative, is called the *corrector step*. The difference between the predicted and corrected function values supplies information on the local truncation error that can be used to control accuracy and to adjust stepsize.

If one corrector step is good, aren't many better? Why not use each corrector as an improved predictor and iterate to convergence on each step? Answer: Even if you had a *perfect* predictor, the step would still be accurate only to the finite order of the corrector. This incurable error term is on the same order as that which your iteration is supposed to cure, so you are at best changing only the coefficient in front

of the error term by a fractional amount. So dubious an improvement is certainly not worth the effort. Your extra effort would be better spent in taking a smaller stepsize.

As described so far, you might think it desirable or necessary to predict several intervals ahead at each step, then to use all these intervals, with various weights, in a Simpson-like corrector step. That is not a good idea. Extrapolation is the least stable part of the procedure, and it is desirable to minimize its effect. Therefore, the integration steps of a predictor-corrector method are overlapping, each one involving several stepsize intervals h, but extending just one such interval farther than the previous ones. Only that one extended interval is extrapolated by each predictor step.

The most popular predictor-corrector methods are probably the Adams-Bashforth-Moulton schemes, which have good stability properties. The Adams-Bashforth part is the predictor. For example, the third-order case is

$$\text{predictor:} \qquad y_{n+1} = y_n + \frac{h}{12}(23y'_n - 16y'_{n-1} + 5y'_{n-2}) + O(h^4) \tag{16.7.3}$$

Here information at the current point x_n, together with the two previous points x_{n-1} and x_{n-2} (assumed equally spaced), is used to predict the value y_{n+1} at the next point, x_{n+1}. The Adams-Moulton part is the corrector. The third-order case is

$$\text{corrector:} \qquad y_{n+1} = y_n + \frac{h}{12}(5y'_{n+1} + 8y'_n - y'_{n-1}) + O(h^4) \tag{16.7.4}$$

Without the trial value of y_{n+1} from the predictor step to insert on the right-hand side, the corrector would be a nasty implicit equation for y_{n+1}.

There are actually three separate processes occurring in a predictor-corrector method: the predictor step, which we call P, the evaluation of the derivative y'_{n+1} from the latest value of y, which we call E, and the corrector step, which we call C. In this notation, iterating m times with the corrector (a practice we inveighed against earlier) would be written P(EC)^m. One also has the choice of finishing with a C or an E step. The lore is that a final E is superior, so the strategy usually recommended is PECE.

Notice that a PC method with a fixed number of iterations (say, one) is an explicit method! When we fix the number of iterations in advance, then the final value of y_{n+1} can be written as some complicated function of known quantities. Thus fixed iteration PC methods lose the strong stability properties of implicit methods and *should only be used for nonstiff problems.*

For stiff problems we *must* use an implicit method if we want to avoid having tiny stepsizes. (Not all implicit methods are good for stiff problems, but fortunately some good ones such as the Gear formulas are known.) We then appear to have two choices for solving the implicit equations: functional iteration to convergence, or Newton iteration. However, it turns out that for stiff problems functional iteration will not even converge unless we use tiny stepsizes, no matter how close our prediction is! Thus Newton iteration is usually an essential part of a multistep stiff solver. For convergence, Newton's method doesn't particularly care what the stepsize is, as long as the prediction is accurate enough.

Multistep methods, as we have described them so far, suffer from two serious difficulties when one tries to implement them:

- Since the formulas require results from equally spaced steps, adjusting the stepsize is difficult.

- Starting and stopping present problems. For starting, we need the initial values plus several previous steps to prime the pump. Stopping is a problem because equal steps are unlikely to land directly on the desired termination point.

Older implementations of PC methods have various cumbersome ways of dealing with these problems. For example, they might use Runge-Kutta to start and stop. Changing the stepsize requires considerable bookkeeping to do some kind of interpolation procedure. Fortunately both these drawbacks disappear with the multivalue approach.

For multivalue methods the basic data available to the integrator are the first few terms of the Taylor series expansion of the solution at the current point x_n. The aim is to advance the solution and obtain the expansion coefficients at the next point x_{n+1}. This is in contrast to multistep methods, where the data are the values of the solution at $x_n, x_{n-1}, \ldots$. We'll illustrate the idea by considering a four-value method, for which the basic data are

$$\mathbf{y}_n \equiv \begin{pmatrix} y_n \\ hy'_n \\ (h^2/2)y''_n \\ (h^3/6)y'''_n \end{pmatrix} \tag{16.7.5}$$

It is also conventional to scale the derivatives with the powers of $h = x_{n+1} - x_n$ as shown. Note that here we use the vector notation $\mathbf{y}$ to denote the solution and its first few derivatives at a point, not the fact that we are solving a system of equations with many components y.

In terms of the data in (16.7.5), we can approximate the value of the solution y at some point x:

$$y(x) = y_n + (x - x_n)y'_n + \frac{(x - x_n)^2}{2}y''_n + \frac{(x - x_n)^3}{6}y'''_n \tag{16.7.6}$$

Set $x = x_{n+1}$ in equation (16.7.6) to get an approximation to y_{n+1}. Differentiate equation (16.7.6) and set $x = x_{n+1}$ to get an approximation to y'_{n+1}, and similarly for y''_{n+1} and y'''_{n+1}. Call the resulting approximation $\tilde{\mathbf{y}}_{n+1}$, where the tilde is a reminder that all we have done so far is a polynomial extrapolation of the solution and its derivatives; we have not yet used the differential equation. You can easily verify that

$$\tilde{\mathbf{y}}_{n+1} = \mathbf{B} \cdot \mathbf{y}_n \tag{16.7.7}$$

where the matrix $\mathbf{B}$ is

$$\mathbf{B} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{16.7.8}$$

We now write the actual approximation to $\mathbf{y}_{n+1}$ that we will use by adding a correction to $\tilde{\mathbf{y}}_{n+1}$:

$$\mathbf{y}_{n+1} = \tilde{\mathbf{y}}_{n+1} + c\mathbf{r} \tag{16.7.9}$$

Here $\mathbf{r}$ will be a fixed vector of numbers, in the same way that $\mathbf{B}$ is a fixed matrix. We fix α by requiring that the differential equation

$$y'_{n+1} = f(x_{n+1}, y_{n+1}) \tag{16.7.10}$$

be satisfied. The second of the equations in (16.7.9) is

$$hy'_{n+1} = h\tilde{y}'_{n+1} + \alpha r_2 \tag{16.7.11}$$

and this will be consistent with (16.7.10) provided

$$r_2 = 1, \qquad \alpha = hf(x_{n+1}, y_{n+1}) - h\tilde{y}'_{n+1} \tag{16.7.12}$$

The values of r_1, r_3, and r_4 are free for the inventor of a given four-value method to choose. Different choices give different orders of method (i.e., through what order in h the final expression 16.7.9 actually approximates the solution), and different stability properties.

An interesting result, not obvious from our presentation, is that multivalue and multistep methods are entirely equivalent. In other words, the value y_{n+1} given by a multivalue method with given $\mathbf{B}$ and $\mathbf{r}$ is exactly the same value given by some multistep method with given β's in equation (16.7.2). For example, it turns out that the Adams-Bashforth formula (16.7.3) corresponds to a four-value method with $r_1 = 0$, $r_3 = 3/4$, and $r_4 = 1/6$. The method is explicit because $r_1 = 0$. The Adams-Moulton method (16.7.4) corresponds to the implicit four-value method with $r_1 = 5/12$, $r_3 = 3/4$, and $r_4 = 1/6$. Implicit multivalue methods are solved the same way as implicit multistep methods: either by a predictor-corrector approach using an explicit method for the predictor, or by Newton iteration for stiff systems.

Why go to all the trouble of introducing a whole new method that turns out to be equivalent to a method you already knew? The reason is that multivalue methods allow an easy solution to the two difficulties we mentioned above in actually implementing multistep methods.

Consider first the question of stepsize adjustment. To change stepsize from h to h' at some point x_n, simply multiply the components of $\mathbf{y}_n$ in (16.7.5) by the appropriate powers of h'/h, and you are ready to continue to $x_n + h'$.

Multivalue methods also allow a relatively easy change in the *order* of the method: Simply change $\mathbf{r}$. The usual strategy for this is first to determine the new stepsize with the current order from the error estimate. Then check what stepsize would be predicted using an order one greater and one smaller than the current order. Choose the order that allows you to take the biggest next step. Being able to change order also allows an easy solution to the starting problem: Simply start with a first-order method and let the order automatically increase to the appropriate level.

For low accuracy requirements, a Runge-Kutta routine like `rkqs` is almost always the most efficient choice. For high accuracy, `bsstep` is both robust and efficient. For very smooth functions, a variable-order PC method can invoke very high orders. If the right-hand side of the equation is relatively complicated, so that the expense of evaluating it outweighs the bookkeeping expense, then the best PC packages can outperform Bulirsch-Stoer on such problems. As you can imagine, however, such a variable-stepsize, variable-order method is not trivial to program. If

you suspect that your problem is suitable for this treatment, we recommend use of a canned PC package. For further details consult Gear [1] or Shampine and Gordon [2].

Our prediction, nevertheless, is that, as extrapolation methods like Bulirsch-Stoer continue to gain sophistication, they will eventually beat out PC methods in all applications. We are willing, however, to be corrected.

CITED REFERENCES AND FURTHER READING:

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9. [1]

Shampine, L.F., and Gordon, M.K. 1975, *Computer Solution of Ordinary Differential Equations. The Initial Value Problem.* (San Francisco: W.H Freeman). [2]

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

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 8.

Hamming, R.W. 1962, *Numerical Methods for Engineers and Scientists*; reprinted 1986 (New York: Dover), Chapters 14–15.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 7.