it to be determined by the data set. The formalism for doing this is called "use of a priori covariances."

A related problem occurs in signal processing and control theory, where it is sometimes desired to "track" (i.e. maintain an estimate of) a time-varying signal in the presence of noise. If the signal is known to be characterized by some number of parameters that vary only slowly, then the formalism of *Kalman filtering* tells how the incoming, raw measurements of the signal should be processed to produce best parameter estimates as a function of time. For example, if the signal is a frequency-modulated sine wave, then the slowly varying parameter might be the instantaneous frequency. The Kalman filter for this case is called a *phase-locked loop* and is implemented in the circuitry of good radio receivers.

Consult Bryson and Ho, or Jazwinski for details on these and other techniques.

REFERENCES AND FURTHER READING:

Huber, P.J. 1981, *Robust Statistics* (New York: Wiley).

Launer, R.L., and Wilkinson, G.N., eds. 1979, *Robustness in Statistics* (New York: Academic Press).

Bryson, A. E., and Ho, Y.C. 1969, *Applied Optimal Control* (Waltham, Mass.: Ginn).

Jazwinski, A. H. 1970, *Stochastic Processes and Filtering Theory* (New York: Academic Press).

Chapter 15. Integration of Ordinary Differential Equations

15.0 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

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x) \tag{15.0.1}$$

can be rewritten as two first order equations

$$\begin{aligned} \frac{dy}{dx} &= z(x) \\ \frac{dz}{dx} &= r(x) - q(x)z(x) \end{aligned} \tag{15.0.2}$$

where z is a new variable. This is exemplary of 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

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \ldots, y_N), \qquad i = 1, \ldots, N \tag{15.0.3}$$

where the functions f_i' on the right-hand side are known. (The "prime" here does not mean to take a derivative; it is only a notational reminder that the f' functions *are* the derivatives of the y's.)

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 (15.0.3). 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 16.

The underlying idea of any routine for solving the initial value problem is always this: Rewrite the dy's and dx's in (15.0.3) 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* (15.1.1, 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. When combined with a particular way of taking individual steps (the *modified midpoint method*) and a particular kind of extrapolation (rational function extrapolation), Richardson extrapolation produces the *Bulirsch-Stoer method*.

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. 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 §15.5, 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 returns 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` and `MMID` are algorithm routines; `RKQC` and `BSSTEP` are steppers; `RKDUMB` and `ODEINT` are drivers.

The final section of this chapter is a brief introduction to the subject of *stiff equations*, relevant both to ordinary differential equations and also to partial differential equations (Chapter 17).

REFERENCES AND FURTHER READING:

Gear, C. William. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, N.J.: Prentice-Hall).

Acton, Forman S. 1970, *Numerical Methods That Work* (New York: Harper and Row), Chapter 5.

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

15.1 Runge-Kutta Method

The formula for the Euler method is

$$y_{n+1} = y_n + hf'(x_n, y_n) \tag{15.1.1}$$

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 15.1.1). 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 (15.1.1).

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 §15.6 below).

Consider, however, the use of a step like (15.1.1) 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 15.1.2 illustrates the idea. In equations,

$$\begin{aligned} k_1 &= hf'(x_n, y_n) \\ k_2 &= hf'\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{15.1.2}$$

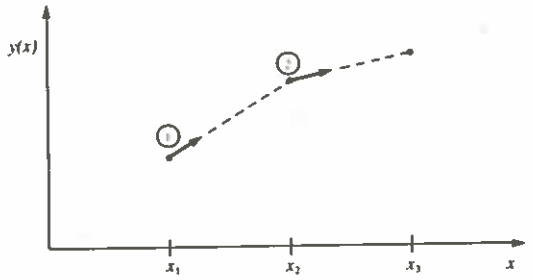

Figure 15.1.1. Euler's method. In this simplest (and least-accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

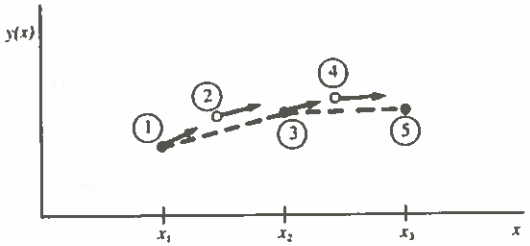

Figure 15.1.2. Midpoint method. Second order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open dots represent function values that are discarded once their derivatives have been calculated and used.

As indicated in the error term, this symmetrization cancels out the first order error term, making the method *second order*. [A method is conventionally called n^{th} order if its error term is $O(h^{n+1})$.] In fact, (15.1.2) 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 $f'(x, y)$ which all agree to first order, but which 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, and Gear, give various specific formulas which derive from this basic idea. By far the most often used, and arguably even most useful, is the *fourth-order Runge-Kutta formula*, which has a certain sleekness of organization about it:

$$\begin{aligned} k_1 &= hf'(x_n, y_n) \\ k_2 &= hf'(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= hf'(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= hf'(x_n + h, y_n + k_3) \end{aligned}$$

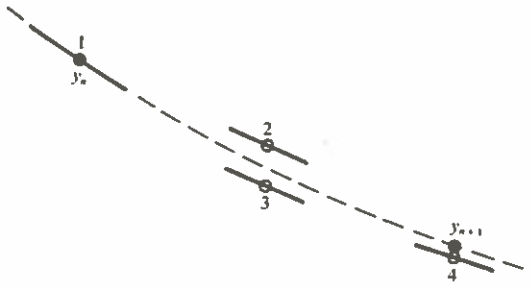

Figure 15.1.3. Fourth-order Runge-Kutta method. In each step the derivative is evaluated four times: once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the final function value (shown as a filled dot) is calculated. (See text for details.)

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5) \tag{15.1.3}$$

The fourth-order Runge-Kutta method requires four evaluations of the right-hand side per step h (see Figure 15.1.3). This will be superior to the midpoint method (15.1.2) *if* at least twice as large a step is possible with (15.1.3) for the same accuracy. Is that so? The answer is: often, perhaps even usually, but surely not always! This takes us back to a central theme, namely that *high order* does not always mean *high accuracy*. The statement "fourth-order Runge-Kutta is generally superior to second-order" is a true one, but you should recognize it as a statement about the contemporary practice of science rather than as a statement about strict mathematics. That is, it reflects the nature of the problems that contemporary scientists like to solve.

By the same token, and with the same caveats, fourth-order Runge-Kutta is generally found superior to *higher*-order Runge-Kutta schemes, and that is why you rarely see those formulas written out. One interesting fact is that, for orders M higher than four, more than M function evaluations (though never more than $M + 2$) are required. Thus fourth-order is a natural breakpoint.

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, as we shall do in the next section. 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.

Here is the routine for carrying out one 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 a routine 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 which follows has, therefore, only three calls to DERIVS.

```
SUBROUTINE RK4(Y,DYDX,N,X,H,YOUT,DERIVS)
    Given values for N variables Y and their derivatives DYDX 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, which need not be a distinct array from Y.  The user supplies the
    subroutine DERIVS(X,Y,DYDX) which returns derivatives DYDX at X.
PARAMETER (NMAX=10)              Set to the maximum number of functions
DIMENSION Y(N),DYDX(N),YOUT(N),YT(NMAX),DYT(NMAX),DYM(NMAX)
HH=H*0.5
H6=H/6.
XH=X+HH
DO 11 I=1,N                      First step
    YT(I)=Y(I)+HH*DYDX(I)
  11 CONTINUE
CALL DERIVS(XH,YT,DYT)           Second step
DO 12 I=1,N
    YT(I)=Y(I)+HH*DYT(I)
  12 CONTINUE
CALL DERIVS(XH,YT,DYM)           Third step
DO 13 I=1,N
    YT(I)=Y(I)+H*DYM(I)
    DYM(I)=DYT(I)+DYM(I)
  13 CONTINUE
CALL DERIVS(X+H,YT,DYT)          Fourth step
DO 14 I=1,N                      Accumulate increments with proper weights.
    YOUT(I)=Y(I)+H6*(DYDX(I)+DYT(I)+2.*DYM(I))
  14 CONTINUE
RETURN
END
```

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 a common block PATH.

```
      SUBROUTINE RKDUMB(VSTART,NVAR,X1,X2,NSTEP,DERIVS)
C     Starting from initial values VSTART for NVAR functions, known at X1 use fourth-order
C     Runge-Kutta to advance NSTEP equal increments to X2.  The user supplied subroutine
C     DERIVS(X,V,DVDX) evaluates derivatives.  Results are stored in the common block PATH.
C     Be sure to dimension the common block appropriately.
      PARAMETER (NMAX=10)                   Set to the maximum number of functions
      COMMON /PATH/ XX(200),Y(10,200)
      DIMENSION VSTART(NVAR),V(NMAX),DV(NMAX)
      DO 11 I=1,NVAR                        Load starting values.
        V(I)=VSTART(I)
        Y(I,1)=V(I)
11    CONTINUE
      XX(1)=X1
      X=X1
      H=(X2-X1)/NSTEP
      DO 13 K=1,NSTEP                       Take NSTEP steps.
        CALL DERIVS(X,V,DV)
        CALL RK4(V,DV,NVAR,X,H,V,DERIVS)
        IF(X+H.EQ.X)PAUSE 'Stepsize not significant in RKDUMB.'
        X=X+H
        XX(K+1)=X                           Store intermediate steps.
        DO 12 I=1,NVAR
          Y(I,K+1)=V(I)
12      CONTINUE
13    CONTINUE
      RETURN
      END
```

REFERENCES AND FURTHER READING:

Gear, C. William. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 2.

Abramowitz, Milton, and Stegun, Irene A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.5.

Rice, J.R. 1983, *Numerical Methods, Software, and Analysis* (New York: McGraw-Hill), §9.2.

15.2 Adaptive Stepsize Control for Runge-Kutta

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

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

With fourth-order Runge-Kutta, the most straightforward technique by far is *step doubling*. We take each step twice, once as a full step, then, independently, as two half steps (see Figure 15.2.1). 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

$$\begin{aligned} y(x+2h) &= y_1 + (2h)^5\phi + O(h^6) + \cdots \\ y(x+2h) &= y_2 + 2(h^5)\phi + O(h^6) + \cdots \end{aligned} \tag{15.2.1}$$

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 (15.2.1) 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

$$\Delta \equiv y_2 - y_1. \tag{15.2.2}$$

It is this difference which 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 (15.2.1) to improve our numerical estimate of the true solution $y(x + 2h)$, namely,

$$y(x+2h) = y_2 + \frac{\Delta}{15} + O(h^6). \tag{15.2.3}$$

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: (15.2.3) may be fifth-order accurate, but we have no way of monitoring *its* truncation

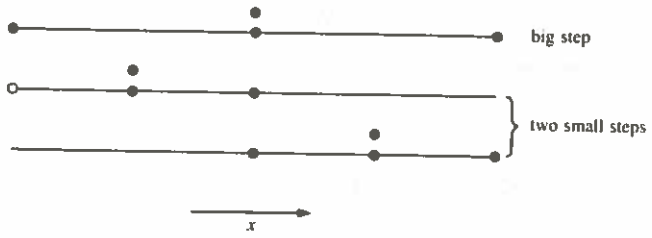

Figure 15.2.1. Step-doubling as a means for adaptive stepsize control in fourth-order Runge-Kutta. Points where the derivative is evaluated are shown as filled circles. The open circle represents the same derivatives as the filled circle immediately above it, so the total number of evaluations is 11 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.

error. Higher order is not always higher accuracy! Use of (15.2.3) 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 (15.2.3).

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 (15.2.1)–(15.2.2), Δ scales as h^5. If we take a step h_1 and produce an error Δ_1, therefore, the step h_0 which *would have given* some other value Δ_0 is readily estimated as

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2} \tag{15.2.4}$$

Henceforth we will let Δ_0 denote the *desired* accuracy. Then equation (15.2.4) 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.*

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`. Let us require the user to specify for each step another, corresponding, vector argument `YSCAL`, and also an overall tolerance level `EPS`. Then the desired accuracy for the i^{th} equation will be taken to be

$$\Delta_0 = \texttt{EPS} \times \texttt{YSCAL}_i \tag{15.2.5}$$

If you desire constant fractional errors, plug `Y` into the `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 $\texttt{YSCAL}_i$ 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

$$\Delta_0 = \epsilon h \times \texttt{DYDX}_i \tag{15.2.6}$$

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 (15.2.4). 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 which 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 (15.2.4) is thus replaced by

$$h_0 = Sh_1 \left|\frac{\Delta_0}{\Delta_1}\right|^{0.20} \qquad \Delta_0 \geq \Delta_1$$
$$= Sh_1 \left|\frac{\Delta_0}{\Delta_1}\right|^{0.25} \qquad \Delta_0 < \Delta_1 \tag{15.2.7}$$

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.

```
      SUBROUTINE RKQC(Y,DYDX,N,X,HTRY,EPS,YSCAL,HDID,HNEXT,DERIVS)
        Fifth-order Runge-Kutta step with monitoring of local truncation error to ensure accuracy
        and adjust stepsize. Input are the dependent variable vector Y of length N and its derivative
        DYDX 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 against which the
        error is scaled. On output, Y and X are replaced by their new values, HDID is the stepsize
        which was actually accomplished, and HNEXT is the estimated next stepsize. DERIVS is the
        user-supplied subroutine that computes the right-hand side derivatives.
      PARAMETER (NMAX=10,PGROW=-0.20,PSHRNK=-0.25,FCOR=1./15.,
     *    ONE=1.,SAFETY=0.9,ERRCON=6.E-4)
        The value ERRCON equals (4/SAFETY)**(1/PGROW), see use below.
      EXTERNAL DERIVS
      DIMENSION Y(N),DYDX(N),YSCAL(N),YTEMP(NMAX),YSAV(NMAX),DYSAV(NMAX)
      XSAV=X                              Save initial values.
      DO 11 I=1,N
        YSAV(I)=Y(I)
        DYSAV(I)=DYDX(I)
      11CONTINUE
      H=HTRY                              Set stepsize to the initial trial value.
    1 HH=0.5*H                            Take two half steps.
      CALL RK4(YSAV,DYSAV,N,XSAV,HH,YTEMP,DERIVS)
      X=XSAV+HH
      CALL DERIVS(X,YTEMP,DYDX)
      CALL RK4(YTEMP,DYDX,N,X,HH,Y,DERIVS)
      X=XSAV+H
      IF(X.EQ.XSAV)PAUSE 'Stepsize not significant in RKQC.'
      CALL RK4(YSAV,DYSAV,N,XSAV,H,YTEMP,DERIVS)      Take the large step.
      ERRMAX=0.                           Evaluate accuracy.
      DO 12 I=1,N
        YTEMP(I)=Y(I)-YTEMP(I)                YTEMP now contains the error estimate.
        ERRMAX=MAX(ERRMAX,ABS(YTEMP(I)/YSCAL(I)))
      12CONTINUE
      ERRMAX=ERRMAX/EPS                   Scale relative to required tolerance.
      IF(ERRMAX.GT.ONE) THEN              Truncation error too large, reduce stepsize.
        H=SAFETY*H*(ERRMAX**PSHRNK)
        GOTO 1                            For another try.
      ELSE                                Step succeeded. Compute size of next step.
        HDID=H
        IF(ERRMAX.GT.ERRCON)THEN
          HNEXT=SAFETY*H*(ERRMAX**PGROW)
        ELSE
          HNEXT=4.*H
        ENDIF
      ENDIF
      DO 13 I=1,N                         Mop up fifth-order truncation error.
        Y(I)=Y(I)+YTEMP(I)*FCOR
```



```
      13CONTINUE
      RETURN
      END
```

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 computer 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 a common block PATH, which can hold up to 200 steps. Because steps occur at unequal intervals results are only stored at intervals greater than DXSAV. Also in the block is KMAX, indicating the number of steps that can be stored. If KMAX=0 there is no intermediate storage, and the rest of the common block need not exist. 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. ODEINT should be customized to the problem at hand.

```
      SUBROUTINE ODEINT(YSTART,NVAR,X1,X2,EPS,H1,HMIN,NOK,NBAD,DERIVS,RKQC)
        Runge-Kutta driver with adaptive stepsize control. Integrate the NVAR starting values
        YSTART from X1 to X2 with accuracy EPS, storing intermediate results in the common block
        /PATH/. 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 subroutine for calculating the right-hand side derivative, while
        RKQC is the name of the stepper routine to be used. PATH contains its own information
        about how often an intermediate value is to be stored.
      PARAMETER (MAXSTP=10000,NMAX=10,TWO=2.0,ZERO=0.0,TINY=1.E-30)
      COMMON /PATH/ KMAX,KOUNT,DXSAV,XP(200),YP(10,200)
        User storage for intermediate results. Preset DXSAV and KMAX.
      DIMENSION YSTART(NVAR),YSCAL(NMAX),Y(NMAX),DYDX(NMAX)
      X=X1
      H=SIGN(H1,X2-X1)
      NOK=0
      NBAD=0
      KOUNT=0
      DO 11 I=1,NVAR
        Y(I)=YSTART(I)
      11CONTINUE
      IF (KMAX.GT.0) XSAV=X-DXSAV*TWO       Assures storage of first step.
      DO 16 NSTP=1,MAXSTP                   Take at most MAXSTP steps.
        CALL DERIVS(X,Y,DYDX)
        DO 12 I=1,NVAR                      Scaling used to monitor accuracy. This general-purpose choice can
          YSCAL(I)=ABS(Y(I))+ABS(H*DYDX(I))+TINY      be modified if need be.
```

```
      12 CONTINUE
    IF(KMAX.GT.0)THEN
        IF(ABS(X-XSAV).GT.ABS(DXSAV)) THEN        Store intermediate results.
            IF(KOUNT.LT.KMAX-1)THEN
                KOUNT=KOUNT+1
                XP(KOUNT)=X
                DO 13 I=1,NVAR
                    YP(I,KOUNT)=Y(I)
                  13 CONTINUE
                XSAV=X
            ENDIF
        ENDIF
    ENDIF
    IF((X+H-X2)*(X+H-X1).GT.ZERO) H=X2-X     If step can overshoot end, cut down stepsize.
    CALL RKQC(Y,DYDX,NVAR,X,H,EPS,YSCAL,HDID,HNEXT,DERIVS)
    IF(HDID.EQ.H)THEN
        NOK=NOK+1
    ELSE
        NBAD=NBAD+1
    ENDIF
    IF((X-X2)*(X2-X1).GE.ZERO)THEN        Are we done?
        DO 14 I=1,NVAR
            YSTART(I)=Y(I)
          14 CONTINUE
        IF(KMAX.NE.0)THEN
            KOUNT=KOUNT+1      Save final step.
            XP(KOUNT)=X
            DO 15 I=1,NVAR
                YP(I,KOUNT)=Y(I)
              15 CONTINUE
        ENDIF
        RETURN                 Normal exit.
    ENDIF
    IF(ABS(HNEXT).LT.HMIN) PAUSE 'Stepsize smaller than minimum.'
    H=HNEXT
  16 CONTINUE
PAUSE 'Too many steps.'
RETURN
END
```

REFERENCES AND FURTHER READING:

Gear, C. William. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, N.J.: Prentice-Hall).

15.3 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,

$$h = H/n \tag{15.3.1}$$

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 §15.4. You can therefore consider this section as a preamble to §15.4.

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

$$\begin{aligned} z_0 &\equiv y(x) \\ z_1 &= z_0 + hf'(x, z_0) \\ z_{m+1} &= z_{m-1} + 2hf'(x + mh, z_m) \qquad \text{for} \quad m = 1, 2, \ldots, n-1 \\ y(x + H) \approx y_n &\equiv \frac{1}{2}[z_n + z_{n-1} + hf'(x + H, z_n)] \end{aligned} \tag{15.3.2}$$

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 15.1.2), except at the first and last points. Those give the qualifier "modified."

The modified midpoint method is a second-order method, like (15.1.2), 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 (15.3.2), 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 fourth-order Runge-Kutta with adaptive stepsize control, as implemented in the preceding section.

The usefulness of the modified midpoint method to the Bulirsch-Stoer technique (§15.4) derives from a "deep" result about equations (15.3.2), due to Gragg. It turns out that the error of (15.3.2), expressed as a power series in h, the stepsize, contains only *even* powers of h,

$$y_n - y(x + H) = \sum_{i=1}^{\infty} \alpha_i h^{2i} \tag{15.3.3}$$

where H is held constant, but h changes by varying n in (15.3.1). 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 (15.3.1) and (15.3.2) with half as many steps, $n \to n/2$. Then the estimate

$$y(x + H) \approx \frac{4y_n - y_{n/2}}{3} \tag{15.3.4}$$

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 (15.3.4), 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 (15.3.4) 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.

```
      SUBROUTINE MMID(Y,DYDX,NVAR,XS,HTOT,NSTEP,YOUT,DERIVS)
        Modified midpoint step. Dependent variable vector Y of length NVAR and its derivative
        vector DYDX are input at XS. 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, which need not be a
        distinct array from Y, if it is distinct, however, then Y and DYDX are returned undamaged.
      PARAMETER (NMAX=10)
      DIMENSION Y(NVAR),DYDX(NVAR),YOUT(NVAR),YM(NMAX),YN(NMAX)
      H=HTOT/NSTEP                    Stepsize this trip.
      DO 11 I=1,NVAR
        YM(I)=Y(I)
        YN(I)=Y(I)+H*DYDX(I)          First step.
11    CONTINUE
      X=XS+H
      CALL DERIVS(X,YN,YOUT)          Will use YOUT for temporary storage of derivatives.
      H2=2.*H
      DO 13 N=2,NSTEP                 General step.
        DO 12 I=1,NVAR
          SWAP=YM(I)+H2*YOUT(I)
          YM(I)=YN(I)
          YN(I)=SWAP
12      CONTINUE
        X=X+H
        CALL DERIVS(X,YN,YOUT)
13    CONTINUE
      DO 14 I=1,NVAR                  Last step.
        YOUT(I)=0.5*(YM(I)+YN(I)+H*YOUT(I))
14    CONTINUE
      RETURN
      END
```

REFERENCES AND FURTHER READING:

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

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

15.4 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 which 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 which 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, rarely encountered in practice, is discussed in the next section.)

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 15.4.1). 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. You might wish at this point to review §3.1–§3.2, where rational function extrapolation was 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 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*. 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

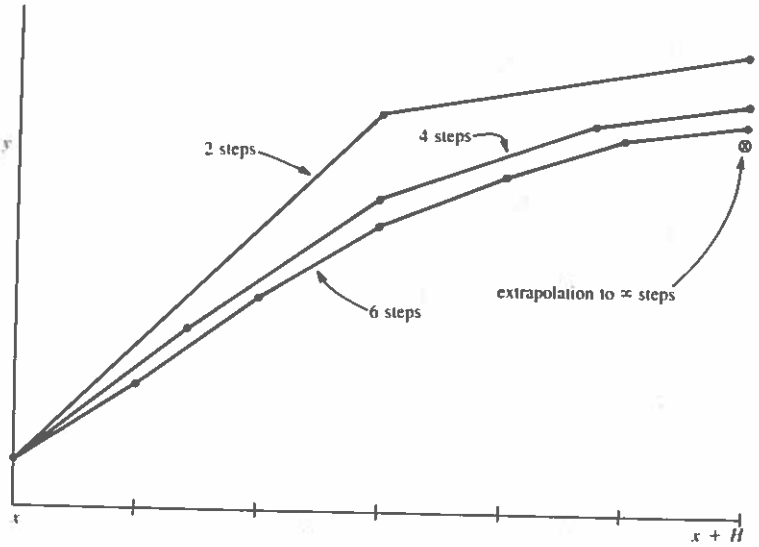

Figure 15.4.1. Richardson extrapolation as used in the Bulirsch-Stoer method. A large interval H is spanned by different sequences of finer and finer substeps. Their results are extrapolated to an answer which is supposed to correspond to infinitely fine substeps. In the Bulirsch-Stoer method, the integrations are done by the modified midpoint method, and the extrapolation technique is rational function extrapolation.

consisting of many (e.g. dozens to hundreds) substeps of modified midpoint method, which are then rational-function 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. A conventional sequence of n's is

$$n = 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, \ldots, [n_j = 2n_{j-2}], \ldots \qquad (15.4.1)$$

(The more obvious choice $n = 2, 4, 8, 16, \ldots$ makes h too small too rapidly.) For each step, we do not know in advance how far up this sequence we will go. After each successive n is tried, a rational function extrapolation is attempted. That extrapolation returns 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 IMAX. We usually take this equal to 11; the $11th$ value of the sequence (15.4.1) is 96, so this is the maximum number of subdivisions of H that we allow.

Another adjustable parameter is the number of previous estimates of the functions at $x + H$ (different values of n) to incorporate into the rational function fit. One possibility would be to use all the estimates available,

$n = 2, 4, 6, \ldots$ etc. Experience shows that, by the time n gets moderately large, the early values are not very relevant. Therefore we usually take the parameter NUSE, the number of estimates to use, equal to 7. (These are values suggested by Gear.)

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 (or more precisely, like that of equations 3.2.6–3.2.7) 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.

Different from Runge-Kutta, and not a fully solved problem in the literature or in practice, is the question of just *how* to increase or decrease the big stepsize H. The problem is not a lack of acceptable ways. The problem is that almost *any* sensible scheme works well, so it is difficult to show that any particular scheme accomplishes the goal of minimizing computation over some universe of hypothetical problems. A key point to keep in mind is that *each* Bulirsch-Stoer step effectively ranges over a factor of, say, up to 96/2 in stepsize (cf. 15.4.1 above). Furthermore, each step can be effectively of very high order in h^2 — if "order" means anything at all for values of h as large as the method often takes! Therefore the range of accuracy already accessible to a step, 48^2 to some high power, is immense. It is a fairly rare event for a Bulirsch-Stoer step to fail at all, and such failure is usually associated with starting or ending transients or internal singularities in the solution.

One desideratum is to keep the number of sequences tried below or at the value NUSE, since after this point early information is thrown away without affecting the solution. As each step succeeds, we learn the value of n at which success occurs. We can then try the next step with H scaled so that the same value h will be reached on, say, sequence number NUSE − 1. When NUSE − 1 succeeds, we might tweak the stepsize up a little bit; while when NUSE succeeds, we might tweak it down a bit.

Of course, if the step truly fails — goes all the way up to IMAX without finding an acceptable error — then we have to take more drastic action, decreasing H substantially and trying the step over.

The following implementation of a Bulirsch-Stoer step has exactly the same calling sequence as the quality-controlled Runge-Kutta stepper RKQC. This means that the driver ODEINT in §15.2 can be used for Bulirsch-Stoer as well as Runge-Kutta: just substitute BSSTEP for RKQC in ODEINT's argument list *and be sure to make the routines consistently either single- or double-precision.* The routine BSSTEP calls MMID to take the modified midpoint sequences, and calls RZEXTR, given below, to do the rational function extrapolation.

```
SUBROUTINE BSSTEP(Y,DYDX,NV,X,HTRY,EPS,YSCAL,HDID,HNEXT,DERIVS)
    Bulirsch-Stoer step with monitoring of local truncation error to ensure accuracy and adjust
    stepsize. Input are the dependent variable vector Y of length NV and its derivative DYDX
    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 against which the error
    is scaled. On output, Y and X are replaced by their new values, HDID is the stepsize
```

```
      which was actually accomplished, and HNEXT is the estimated next stepsize. DERIVS is the
      user-supplied subroutine that computes the right-hand side derivatives.
      PARAMETER (NMAX=10,IMAX=11,NUSE=7,ONE=1.E0,SHRINK=.95E0,GROW=1.2E0)
      DIMENSION Y(NV),DYDX(NV),YSCAL(NV),YERR(NMAX),
     *    YSAV(NMAX),DYSAV(NMAX),YSEQ(NMAX),NSEQ(IMAX)
      DATA NSEQ /2,4,6,8,12,16,24,32,48,64,96/
      H=HTRY
      XSAV=X
      DO 11 I=1,NV                        Save the starting values.
        YSAV(I)=Y(I)
        DYSAV(I)=DYDX(I)
      11 CONTINUE
    1 DO 10 I=1,IMAX                      Evaluate the sequence of modified midpoint integrations.
        CALL MMID(YSAV,DYSAV,NV,XSAV,H,NSEQ(I),YSEQ,DERIVS)
        XEST=(H/NSEQ(I))**2               Squared, since error series is even.
        CALL RZEXTR(I,XEST,YSEQ,Y,YERR,NV,NUSE) Perform rational function extrapolation.
        ERRMAX=0.                         Check local truncation error.
        DO 12 J=1,NV
          ERRMAX=MAX(ERRMAX,ABS(YERR(J)/YSCAL(J)))
        12 CONTINUE
        ERRMAX=ERRMAX/EPS                 Scale accuracy relative to tolerance.
        IF(ERRMAX.LT.ONE) THEN            Step converged.
          X=X+H
          HDID=H
          IF(I.EQ.NUSE)THEN
            HNEXT=H*SHRINK
          ELSE IF(I.EQ.NUSE-1)THEN
            HNEXT=H*GROW
          ELSE
            HNEXT=(H*NSEQ(NUSE-1))/NSEQ(I)
          ENDIF
          RETURN                          Normal return.
        ENDIF
      10 CONTINUE
        If here, then step failed, quite unusual for this method. We reduce the stepsize and try
        again.
      H=0.25*H/2**((IMAX-NUSE)/2)
      IF(X+H.EQ.X)PAUSE 'Step size underflow.'
      GOTO 1
      END
```

The rational function extrapolation routine is based on the same algorithm as RATINT §3.2. 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.

```
      SUBROUTINE RZEXTR(IEST,XEST,YEST,YZ,DY,NV,NUSE)
        Use diagonal rational function extrapolation to evaluate NV functions at X = 0 by fitting
        a diagonal rational function to a sequence of estimates with progressively smaller values
        X = XEST, and corresponding function vectors YEST. This call is number IEST in the
        sequence of calls. The extrapolation uses at most the last NUSE estimates. Extrapolated
        function values are output as YZ, and their estimated error is output as DY.
      PARAMETER (IMAX=11,NMAX=10,NCOL=7)
        Maximum expected value of NUSE is NCOL; of NV is NMAX; of IEST is IMAX.
      DIMENSION X(IMAX),YEST(NV),YZ(NV),DY(NV),D(NMAX,NCOL),FX(NCOL)
      X(IEST)=XEST                        Save current independent variable.
      IF(IEST.EQ.1) THEN
        DO 11 J=1,NV
          YZ(J)=YEST(J)
          D(J,1)=YEST(J)
          DY(J)=YEST(J)
```



```
        11 CONTINUE
      ELSE
        M1=MIN(IEST,NUSE)                 Use at most NUSE previous members.
        DO 12 K=1,M1-1
          FX(K+1)=X(IEST-K)/XEST
        12 CONTINUE
        DO 14 J=1,NV                      Evaluate next diagonal in tableau.
          YY=YEST(J)
          V=D(J,1)
          C=YY
          D(J,1)=YY
          DO 13 K=2,M1
            B1=FX(K)*V
            B=B1-C
            IF(B.NE.0.)  THEN
              B=(C-V)/B
              DDY=C*B
              C=B1*B
            ELSE                          Care needed to avoid division by 0.
              DDY=V
            ENDIF
            IF (K.NE.M1) V=D(J,K)
            D(J,K)=DDY
            YY=YY+DDY
          13 CONTINUE
          DY(J)=DDY
          YZ(J)=YY
        14 CONTINUE
      ENDIF
      RETURN
      END
```

Rational function extrapolation can fail. The extrapolated function might have a pole at the desired evaluation point. Or, more commmonly, there might be two poles that very nearly cancel, so that roundoff becomes a problem. In the above routine, the test for division by zero prevents program crash, but disguises the fact that the quantity computed as `B1-C` may have lost all significance.

You can use Bulirsch-Stoer for years without encountering a failure, or you can be stopped dead on your first attempt. It all depends on the nature of your ODEs, the precision of your machine, and (we sometimes think) whether the fates are smiling on you. The rewards of mastering Bulirsch-Stoer are so great that we urge persistence.

Keep a watch for failed steps, where HDID is returned with a value less than HTRY. If these are not rare, then Bulirsch-Stoer is in trouble. You can try to save it with either of two options:

- Take a couple of quality-controlled Runge-Kutta steps to get over the rough spot. A call to RKQC above is exactly substitutable for a call to BSSTEP. (You might, however, reduce the suggested stepsize HTRY by a factor of 16 or 32, in recognition of Runge-Kutta's necessarily smaller steps. If you don't do this, RKQC will have to seek out the smaller stepsize by itself, with additional effort.)
- If the problem is not a rough spot in your solutions, but purely an artifact in the rational function extrapolation, then a less drastic therapy is to use polynomial extrapolation instead of rational

function extrapolation for a step or two. Polynomial extrapolation does not involve any divisions, and it can be less finicky than rational function extrapolation — also less powerful!

If you are trying the second of the above options, you will want the following polynomial extrapolation routine, which is an exact substitution for RZEXTR above.

```
      SUBROUTINE PZEXTR(IEST,XEST,YEST,YZ,DY,NV,NUSE)
      PARAMETER (IMAX=11,NCOL=7,NMAX=10)
      DIMENSION X(IMAX),YEST(NV),YZ(NV),DY(NV),QCOL(NMAX,NCOL),D(NMAX)
      X(IEST)=XEST                    Store current dependent value.
      DO 11 J=1,NV
        DY(J)=YEST(J)
        YZ(J)=YEST(J)
11    CONTINUE
      IF(IEST.EQ.1) THEN              Store first estimate in first column.
        DO 12 J=1,NV
          QCOL(J,1)=YEST(J)
12      CONTINUE
      ELSE
        M1=MIN(IEST,NUSE)             Use at most NUSE previous estimates.
        DO 13 J=1,NV
          D(J)=YEST(J)
13      CONTINUE
        DO 15 K1=1,M1-1
          DELTA=1./(X(IEST-K1)-XEST)
          F1=XEST*DELTA
          F2=X(IEST-K1)*DELTA
          DO 14 J=1,NV                Propagate tableau 1 diagonal more.
            Q=QCOL(J,K1)
            QCOL(J,K1)=DY(J)
            DELTA=D(J)-Q
            DY(J)=F1*DELTA
            D(J)=F2*DELTA
            YZ(J)=YZ(J)+DY(J)
14        CONTINUE
15      CONTINUE
        DO 16 J=1,NV
          QCOL(J,M1)=DY(J)
16      CONTINUE
      ENDIF
      RETURN
      END
```

REFERENCES AND FURTHER READING:

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

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

15.5 Predictor-Corrector 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. You, the knowledgeable practitioner, had better 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.

Unlike the approaches discussed thus far, *predictor-corrector* methods do not step forward as though each new point were another initial value. Instead these methods record past function values and extrapolate them (via polynomial extrapolation) to *predict* what the next step is going to yield. This is called the *predictor step*. The predictor step does not involve any evaluations of the right-hand side of the equations. Why do it? The answer becomes clear if you 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.

If by some magic we knew the value of the y's that enter into the right-hand side calculation, then integrating an ODE would be exactly the same as integrating a function. We could do the integration by Simpson's rule, or any of the other quadrature rules in §4.1. At whatever value x our chosen rule requires us to evaluate the derivative function f', we do so by evaluating $f'(x, y)$ with the "magically" obtained value of y.

In a predictor-corrector method, the predictor step *is* the desired "magic." Its extrapolated value may not be exact, but if it is accurate to as high an order as the integration will be, then that is good enough. The Simpson-like integration, using the prediction step's value of y in evaluating the right-hand side, is called the *corrector step*. Figure 15.5.1 illustrates the general idea.

Let us here dispose of two silly ideas that might occur to you in an idle moment: (1) Why do any corrector steps at all; why not do predictor step after predictor step? Don't predictor steps alone have the same order as the whole method? Answer: Yes, they do. But order is not the same as accuracy! Repeated extrapolation, with no evaluations of the right-hand

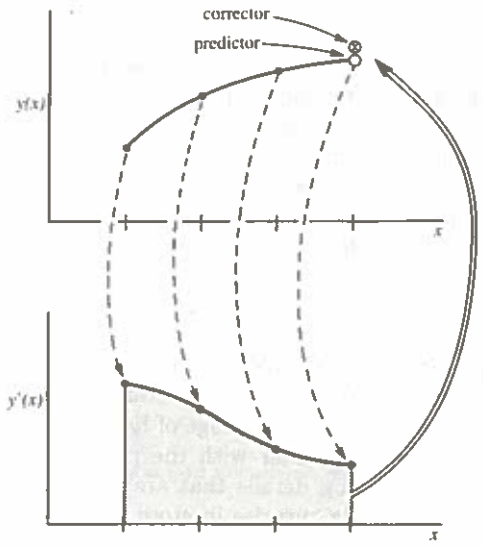

Figure 15.5.1. A predictor-corrector method illustrated schematically. In the upper figure, three already-known points are shown as filled dots. These are polynomial-extrapolated to obtain the predictor, shown as an open dot. The lower figure shows the derivatives as evaluated at each of the upper points, including the predictor. A polynomial is passed through these derivative points and the area under the curve is evaluated. That area, being the integral of y', is the desired corrector point, which becomes a filled dot in the upper figure on the next step. Practical predictor-corrector methods use more complicated combinations of function and derivatives on both predictor and corrector steps.

sides and thus no influx of new information, is wildly unstable and hence wholly inaccurate. (2) 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 only changing 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 the so-called Adams-Bashforth-Moulton schemes, which have good stability properties. The

Adams-Bashforth part is the predictor. For example, the 4^{th} order case is

$$\text{predictor:} \qquad y_1 = y_0 + \frac{h}{24}(55y'_0 - 59y'_{-1} + 37y'_{-2} - 9y'_{-3}) + O(h^5) \quad (15.5.1)$$

Here information at the current point x_0, together with the three previous points x_{-1}, x_{-2} and x_{-3} (assumed equally spaced), is used to predict the value y_1 at the next point, x_1. The prime means calculate the derivative using as the dependent variable the quantity that is primed.

The Adams-Moulton part is the corrector. For the same order as the predictor, the corrector has the same number of terms. The 4^{th} order case is

$$\text{corrector:} \qquad y_1 = y_0 + \frac{h}{24}(9y'_1 + 19y'_0 - 5y'_{-1} + y'_{-2}) + O(h^5) \quad (15.5.2)$$

Without the trial value of y_1 from the predictor step to insert on the right-hand side, the corrector would be a nasty implicit equation for y_1.

There are actually three separate processes occurring in this method: the predictor step, which we call P, the evaluation of the derivative y'_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 $\mathrm{P(EC)}^m$. 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.

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. Unfortunately, predictor-corrector methods are not very flexible as far as stepsize adjustment is concerned.

Since the formulas require results from four equally spaced steps, halving and doubling adjustments to stepsize are the most simply realized. Suppose that we attempt to take a step, but the error indicated by the difference "predictor minus corrector" exceeds the required tolerance. We can use interpolation to generate "old" results at half the current stepsize, and then try again with $h/2$ as the new step.

If the routine is working particularly well, on the other hand, then we can double the stepsize simply by discarding every other previous point and using $2h$. However, we can only do this if we had the foresight to save the necessary information at the last *seven* points for a 4^{th} order method. Thus, the predictor-corrector requires some considerable bookkeeping on prior information, and some checks to allow doubling only when enough steps are available. Also, it is wasteful to double prematurely and then be forced immediately to halve again. For this reason it is best to set a fairly strict accuracy criterion for doubling.

Starting and stopping pose obvious problems for predictor-corrector methods. For starting, we need the initial values plus three previous steps to prime the pump. Stopping is a problem because equal steps are unlikely

to land directly on the desired termination point. The solution is to use Runge-Kutta (or any other available procedure) to start and stop.

The reason we have chosen not to give an implementation of a PC method is that, because of the bookkeeping complexity, the code is about twice as long as the comparable Runge-Kutta or Bulirsch-Stoer routines given earlier. Moreover, such a routine is no better than Bulirsch-Stoer, and in fact is often less efficient.

There do exist even more complicated PC methods that adaptively change the *order* of the method as the integration proceeds, as well as the stepsize. They also use more complicated integration schemes with unequal spacing of steps. For very smooth functions, very high order methods get invoked. 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 a nightmare 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 or Shampine and Gordon.

Our prediction 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.

REFERENCES AND FURTHER READING:

Acton, Forman S. 1970, *Numerical Methods That Work* (New York: Harper and Row), Chapter 5.

Gear, C. William. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, N.J.: Prentice-Hall), Chapter 9.

Hamming, R.W. 1962, *Numerical Methods for Engineers and Scientists* (New York: McGraw-Hill), Chapters 14, 15.

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

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

15.6 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 second-order equation

$$y'' = f(x)y \tag{15.6.1}$$

where $f >> 0$. (This is equivalent to a set of two first-order equations by the usual substitution $y_1 = y$, $y_2 = y'$.) For large values of $f(x)$, y is approximately the sum of two exponentials. For example, if we approximate f as a constant, say $f = 100$, then

$$y = Ae^{-10x} + Be^{10x}, \qquad A, B = \text{constant} \tag{15.6.2}$$

Often the desired solution is the dying exponential. Thus if we integrated equation (15.6.1) with the boundary conditions

$$y(0) = 1 \qquad y'(0) = -10 \tag{15.6.3}$$

the true solution is $y = e^{-10x}$. However, the integration methods given so far in this chapter would give a numerical solution that would start off decaying as e^{-10x}, but would then "explode" as e^{10x} as x becomes large. The reason is clear: any roundoff or truncation error as we start the integration, no matter how small, is equivalent to a small admixture near the origin of the unwanted other solution, e^{10x}. Thus

$$y_{numerical} \approx e^{-10x} + \epsilon e^{10x} \tag{15.6.4}$$

No matter how small ϵ is made (e.g. by taking a very small stepsize), sooner or later the second term in (15.6.4) dominates.

"Simple" stiff equations like (15.6.1) can be handled by a change of variable called the Riccati transformation (see Acton, p. 148). Before considering a general strategy for stiff equations, let us consider an example which shows that stiff systems need not have divergent solutions. Gear gives the following set of equations:

$$\begin{aligned} u' &= 998u + 1998v \\ v' &= -999u - 1999v \end{aligned} \tag{15.6.5}$$

with boundary conditions

$$u(0) = 1 \qquad v(0) = 0 \tag{15.6.6}$$

By means of the transformation

$$u = 2y - z \qquad v = -y + z \tag{15.6.7}$$

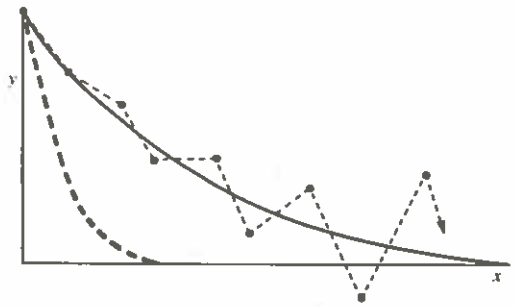

Figure 15.6.1. Example of an instability encountered in integrating a stiff equation (schematic). Here it is supposed that the equation has two solutions, shown as solid and dashed lines. Although the initial conditions are such as to give the solid solution, the stability of the integration (shown as 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.

we find the solution

$$u = 2e^{-x} - e^{-1000x}$$
$$v = -e^{-x} + e^{-1000x} \tag{15.6.8}$$

If we integrated the system (15.6.5) 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 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 15.6.1).

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

$$y' = -cy \tag{15.6.9}$$

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

$$y_{n+1} = y_n + hy'_n = (1 - ch)y_n \tag{15.6.10}$$

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:

$$y_{n+1} = y_n + hy'_{n+1} \tag{15.6.11}$$

or

$$y_{n+1} = \frac{y_n}{1 + ch} \tag{15.6.12}$$

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:

$$\mathbf{y}' = -\mathbf{C} \cdot \mathbf{y} \tag{15.6.13}$$

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

$$\mathbf{y}_{n+1} = (1 - \mathbf{C}h) \cdot \mathbf{y}_n \tag{15.6.14}$$

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 $1 - \mathbf{C}h$ is less than 1, or in other words

$$h < \frac{2}{\lambda_{max}} \tag{15.6.15}$$

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

On the other hand, implicit differencing gives

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{y}'_{n+1} \tag{15.6.16}$$

or

$$\mathbf{y}_{n+1} = (1 + \mathbf{C}h)^{-1} \cdot \mathbf{y}_n \tag{15.6.17}$$

If the eigenvalues of $\mathbf{C}$ are λ, then the eigenvalues of $(1+\mathbf{C}h)^{-1}$ 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

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}) \tag{15.6.18}$$

(note that the notation differs from previous sections by omission of the reminder "prime" on the right-hand side) implicit differencing gives

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(x_{n+1}, \mathbf{y}_{n+1}) \tag{15.6.19}$$

In general this is some nasty set of nonlinear equations that has to be solved iteratively at each step. Usually we can get away with linearizing the equations:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\left[\mathbf{f}(x_{n+1}, \mathbf{y}_n) + \left.\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right|_{\mathbf{y}_n} \cdot (\mathbf{y}_{n+1} - \mathbf{y}_n)\right] \tag{15.6.20}$$

Here $\partial \mathbf{f}/\partial \mathbf{y}$ is the matrix of the partial derivatives of the right-hand side, and so at each step we have to invert the matrix

$$1 - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}} \tag{15.6.21}$$

to find $\mathbf{y}_{n+1}$. This procedure is called a "semi-implicit" 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 tolerate higher order methods. We can easily construct a second-order method by taking the average of the explicit and implicit first-order methods:

$$y_{n+1} = y_n + \frac{h}{2}(y'_{n+1} + y'_n) \tag{15.6.22}$$

This is simply the trapezoidal rule if y' does not depend on y. For our model equation (15.6.9), we find

$$y_{n+1} = \frac{1 - ch/2}{1 + ch/2} y_n \tag{15.6.23}$$

showing stability. For the system (15.6.18), we can construct a semi-implicit scheme analogous to (15.6.20):

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{2}\left[\mathbf{f}(x_{n+1}, \mathbf{y}_n) + \left.\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right|_{\mathbf{y}_n} \cdot (\mathbf{y}_{n+1} - \mathbf{y}_n) + \mathbf{f}(x_n, \mathbf{y}_n)\right] \tag{15.6.24}$$

It is quite complicated to design higher order implicit methods, especially with a good scheme for automatic stepsize adjustment. Higher order methods analogous to Runge-Kutta and predictor-corrector methods have been developed. Gear gives a well tested routine in his book, and references to more recent developments can be found in Stoer and Bulirsch.

REFERENCES AND FURTHER READING:

Acton, Forman S. 1970, *Numerical Methods That Work* (New York: Harper and Row).

Gear, C. William. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, N.J.: Prentice-Hall).

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