```
        ytemp(i)=y(i)+ytemp(n)
    enddo 11
    x=xs+h
    call derivs(x,ytemp,yout)          Use yout for temporary storage of derivatives.
    h2=h*h
    do 13 nn=2,nstep                   General step.
        do 12 i=1,neqns
            n=neqns+i
            ytemp(n)=ytemp(n)+h2*yout(i)
            ytemp(i)=ytemp(i)+ytemp(n)
        enddo 12
        x=x+h
        call derivs(x,ytemp,yout)
    enddo 13
    do 14 i=1,neqns                    Last step.
        n=neqns+i
        yout(n)=ytemp(n)/h+halfh*yout(i)
        yout(i)=ytemp(i)
    enddo 14
    return
    END
```

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

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

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

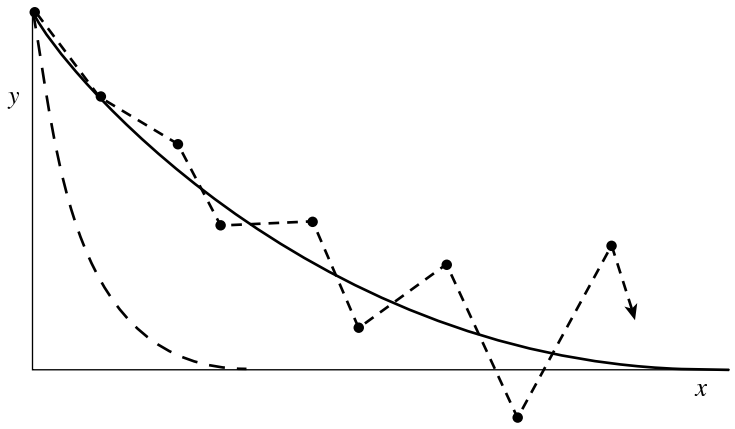

Figure 16.6.1. Example of an instability encountered in integrating a stiff equation (schematic). Here it is supposed that the equation has two solutions, shown as solid and dashed lines. Although the initial conditions are such as to give the solid solution, the stability of the integration (shown as 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.

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 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 $n \to \infty$.

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

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

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

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

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 $\mathbf{y}_{\rm scal}$ with which the error is to be scaled. For example, to get constant fractional errors, simply set $\mathbf{y}_{\rm scal} = |\mathbf{y}|$. You can get constant absolute errors relative to some maximum values

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

by setting $\mathbf{y}_{\rm scal}$ 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

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

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

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

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

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 subroutine (called `derivs` in `stiff`) that returns **f** (called `dydx`) as a function of x and **y**. Also supply a subroutine `jacobn` that returns $\mathbf{f}'$ (`dfdy`) and $\partial\mathbf{f}/\partial x$ (`dfdx`) as functions of x and **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 **f**. If not, your subroutine 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 $\mathbf{1}/\gamma h - \mathbf{f}'$ 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 **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

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

Thus

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

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.

```
      SUBROUTINE stiff(y,dydx,n,x,htry,eps,yscal,hdid,hnext,derivs)
      INTEGER n,NMAX,MAXTRY
      REAL eps,hdid,hnext,htry,x,dydx(n),y(n),yscal(n),SAFETY,GROW,
     *     PGROW,SHRNK,PSHRNK,ERRCON,GAM,A21,A31,A32,A2X,A3X,C21,
     *     C31,C32,C41,C42,C43,B1,B2,B3,B4,E1,E2,E3,E4,C1X,C2X,C3X,
     *     C4X
      EXTERNAL derivs
      PARAMETER (NMAX=50,SAFETY=0.9,GROW=1.5,PGROW=-.25,
     *     SHRNK=0.5,PSHRNK=-1./3.,ERRCON=.1296,MAXTRY=40)
      PARAMETER (GAM=1./2.,A21=2.,A31=48./25.,A32=6./25.,C21=-8.,
     *     C31=372./25.,C32=12./5.,C41=-112./125.,C42=-54./125.,
     *     C43=-2./5.,B1=19./9.,B2=1./2.,B3=25./108.,B4=125./108.,
     *     E1=17./54.,E2=7./36.,E3=0.,E4=125./108.,C1X=1./2.,
```

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```
*       C2X=-3./2.,C3X=121./50.,C4X=29./250.,A2X=1.,A3X=3./5.)
C   USES derivs,jacobn,lubksb,ludcmp
```

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 subroutine that computes the derivatives of the right-hand side with respect to `x`, while `jacobn` (a fixed name) is a user-supplied subroutine that computes the Jacobi matrix of derivatives of the right-hand side with respect to the components of `y`. Parameters: `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=(GROW/SAFETY)**(1/PGROW)` and handles the case when `errmax` $\simeq 0$.

```
      INTEGER i,j,jtry,indx(NMAX)
      REAL d,errmax,h,xsav,a(NMAX,NMAX),dfdx(NMAX),dfdy(NMAX,NMAX),
     *     dysav(NMAX),err(NMAX),g1(NMAX),g2(NMAX),g3(NMAX),
     *     g4(NMAX),ysav(NMAX)
      xsav=x                                  Save initial values.
      do 11 i=1,n
        ysav(i)=y(i)
        dysav(i)=dydx(i)
      enddo 11
      call jacobn(xsav,ysav,dfdx,dfdy,n,NMAX)
```

The user must supply this subroutine to return the `n`-by-`n` matrix `dfdy` and the vector `dfdx`.

```
      h=htry                                  Set stepsize to the initial trial value.
      do 23 jtry=1,MAXTRY
        do 13 i=1,n                           Set up the matrix 1 − γhf′.
          do 12 j=1,n
            a(i,j)=-dfdy(i,j)
          enddo 12
          a(i,i)=1./(GAM*h)+a(i,i)
        enddo 13
        call ludcmp(a,n,NMAX,indx,d)          LU decomposition of the matrix.
        do 14 i=1,n                           Set up right-hand side for g1.
          g1(i)=dysav(i)+h*C1X*dfdx(i)
        enddo 14
        call lubksb(a,n,NMAX,indx,g1)         Solve for g1.
        do 15 i=1,n                           Compute intermediate values of y and x.
          y(i)=ysav(i)+A21*g1(i)
        enddo 15
        x=xsav+A2X*h
        call derivs(x,y,dydx)                 Compute dydx at the intermediate values.
        do 16 i=1,n                           Set up right-hand side for g2.
          g2(i)=dydx(i)+h*C2X*dfdx(i)+C21*g1(i)/h
        enddo 16
        call lubksb(a,n,NMAX,indx,g2)         Solve for g2.
        do 17 i=1,n                           Compute intermediate values of y and x.
          y(i)=ysav(i)+A31*g1(i)+A32*g2(i)
        enddo 17
        x=xsav+A3X*h
        call derivs(x,y,dydx)                 Compute dydx at the intermediate values.
        do 18 i=1,n                           Set up right-hand side for g3.
          g3(i)=dydx(i)+h*C3X*dfdx(i)+(C31*g1(i)+
     *         C32*g2(i))/h
        enddo 18
        call lubksb(a,n,NMAX,indx,g3)         Solve for g3.
        do 19 i=1,n                           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
        enddo 19
        call lubksb(a,n,NMAX,indx,g4)         Solve for g4.
        do 21 i=1,n                           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)
```

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```
            err(i)=E1*g1(i)+E2*g2(i)+E3*g3(i)+E4*g4(i)
        enddo 21
        x=xsav+h
        if(x.eq.xsav)pause 'stepsize not significant in stiff'
        errmax=0.                          Evaluate accuracy.
        do 22 i=1,n
            errmax=max(errmax,abs(err(i)/yscal(i)))
        enddo 22
        errmax=errmax/eps                  Scale relative to required tolerance.
        if(errmax.le.1.)then               Step succeeded. Compute size of next step and re-
            hdid=h                             turn.
            if(errmax.gt.ERRCON)then
                hnext=SAFETY*h*errmax**PGROW
            else
                hnext=GROW*h
            endif
            return
        else                               Truncation error too large, reduce stepsize.
            hnext=SAFETY*h*errmax**PSHRNK
            h=sign(max(abs(hnext),SHRNK*abs(h)),h)
        endif
    enddo 23                               Go back and re-try step.
    pause 'exceeded MAXTRY in stiff'
    END
```

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

```
      PARAMETER (GAM=.231,A21=2.,A31=4.52470820736,A32=4.16352878860,
     *     C21=-5.07167533877,C31=6.02015272865,C32=.159750684673,
     *     C41=-1.856343618677,C42=-8.50538085819,C43=
     *     -2.08407513602,B1=3.95750374663,B2=4.62489238836,B3=
     *     .617477263873,B4=1.282612945268,E1=-2.30215540292,
     *     E2=-3.07363448539,E3=.873280801802,E4=1.282612945268,
     *     C1X=GAM,C2X=-.396296677520e-01,C3X=.550778939579,
     *     C4X=-.553509845700e-01,A2X=.462,A3X=.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!

```
      SUBROUTINE jacobn(x,y,dfdx,dfdy,n,nmax)
      INTEGER n,nmax,i
      REAL x,y(*),dfdx(*),dfdy(nmax,nmax)
      do 11 i=1,3
          dfdx(i)=0.
      enddo 11
```

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```
dfdy(1,1)=-.013-1000.*y(3)
dfdy(1,2)=0.
dfdy(1,3)=-1000.*y(1)
dfdy(2,1)=0.
dfdy(2,2)=-2500.*y(3)
dfdy(2,3)=-2500.*y(2)
dfdy(3,1)=-.013-1000.*y(3)
dfdy(3,2)=-2500.*y(3)
dfdy(3,3)=-1000.*y(1)-2500.*y(2)
return
END

SUBROUTINE derivs(x,y,dydx)
REAL x,y(*),dydx(*)
dydx(1)=-.013*y(1)-1000.*y(1)*y(3)
dydx(2)=-2500.*y(2)*y(3)
dydx(3)=-.013*y(1)-1000.*y(1)*y(3)-2500.*y(2)*y(3)
return
END
```

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

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

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

Finally compute

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

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:

```
      SUBROUTINE simpr(y,dydx,dfdx,dfdy,nmax,n,xs,htot,nstep,yout,
     *     derivs)
      INTEGER n,nmax,nstep,NMAXX
      REAL htot,xs,dfdx(n),dfdy(nmax,nmax),dydx(n),y(n),yout(n)
      EXTERNAL derivs
      PARAMETER (NMAXX=50)                  Maximum expected value of n.
C     USES derivs,lubksb,ludcmp
        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:nmax,1:nmax) 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 subroutine that calculates dydx.
      INTEGER i,j,nn,indx(NMAXX)
      REAL d,h,x,a(NMAXX,NMAXX),del(NMAXX),ytemp(NMAXX)
      h=htot/nstep                          Stepsize this trip.
      do 12 i=1,n                           Set up the matrix 1 − hf′.
        do 11 j=1,n
          a(i,j)=-h*dfdy(i,j)
        enddo 11
        a(i,i)=a(i,i)+1.
      enddo 12
      call ludcmp(a,n,NMAXX,indx,d)         LU decomposition of the matrix.
      do 13 i=1,n                           Set up right-hand side for first step. Use yout for
        yout(i)=h*(dydx(i)+h*dfdx(i))           temporary storage.
      enddo 13
      call lubksb(a,n,NMAXX,indx,yout)
      do 14 i=1,n                           First step.
        del(i)=yout(i)
        ytemp(i)=y(i)+del(i)
      enddo 14
      x=xs+h
      call derivs(x,ytemp,yout)             Use yout for temporary storage of derivatives.
      do 17 nn=2,nstep                      General step.
        do 15 i=1,n                         Set up right-hand side for general step.
          yout(i)=h*yout(i)-del(i)
        enddo 15
        call lubksb(a,n,NMAXX,indx,yout)
        do 16 i=1,n
          del(i)=del(i)+2.*yout(i)
          ytemp(i)=ytemp(i)+del(i)
        enddo 16
        x=x+h
        call derivs(x,ytemp,yout)
      enddo 17
      do 18 i=1,n                           Set up right-hand side for last step.
        yout(i)=h*yout(i)-del(i)
      enddo 18
      call lubksb(a,n,NMAXX,indx,yout)
      do 19 i=1,n                           Take last step.
        yout(i)=ytemp(i)+yout(i)
```

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```
      enddo 19
      return
      END
```

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:

```
      SUBROUTINE stifbs(y,dydx,nv,x,htry,eps,yscal,hdid,hnext,derivs)
      INTEGER nv,NMAX,KMAXX,IMAX
      REAL eps,hdid,hnext,htry,x,dydx(nv),y(nv),yscal(nv),SAFE1,
     *     SAFE2,REDMAX,REDMIN,TINY,SCALMX
      EXTERNAL derivs
      PARAMETER (NMAX=50,KMAXX=7,IMAX=KMAXX+1,SAFE1=.25,SAFE2=.7,
     *     REDMAX=1.e-5,REDMIN=.7,TINY=1.e-30,SCALMX=.1)
C     USES derivs,jacobn,simpr,pzextr
        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 subroutine that computes the derivatives of the right-hand side with respect to x, while jacobn (a fixed name) is a user-supplied subroutine 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.
      INTEGER i,iq,k,kk,km,kmax,kopt,nvold,nseq(IMAX)
      REAL eps1,epsold,errmax,fact,h,red,scale,work,wrkmin,xest,xnew,
     *     a(IMAX),alf(KMAXX,KMAXX),dfdx(NMAX),dfdy(NMAX,NMAX),
     *     err(KMAXX),yerr(NMAX),ysav(NMAX),yseq(NMAX)
      LOGICAL first,reduct
      SAVE a,alf,epsold,first,kmax,kopt,nseq,nvold,xnew
      DATA first/.true./,epsold/-1./,nvold/-1/
      DATA nseq /2,6,10,14,22,34,50,70/          Sequence is different from bsstep.
      if(eps.ne.epsold.or.nv.ne.nvold)then       Reinitialize also if nv has changed.
        hnext=-1.e29
        xnew=-1.e29
        eps1=SAFE1*eps
        a(1)=nseq(1)+1
        do 11 k=1,KMAXX
          a(k+1)=a(k)+nseq(k+1)
        enddo 11
        do 13 iq=2,KMAXX
          do 12 k=1,iq-1
            alf(k,iq)=eps1**((a(k+1)-a(iq+1))/
     *           ((a(iq+1)-a(1)+1.)*(2*k+1)))
          enddo 12
        enddo 13
```

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```
        epsold=eps
        nvold=nv                                  Save nv.
        a(1)=nv+a(1)                              Add cost of Jacobian evaluations to work co-
        do 14 k=1,KMAXX                               efficients.
            a(k+1)=a(k)+nseq(k+1)
        enddo 14
        do 15 kopt=2,KMAXX-1
            if(a(kopt+1).gt.a(kopt)*alf(kopt-1,kopt))goto 1
        enddo 15
1       kmax=kopt
    endif
    h=htry
    do 16 i=1,nv
        ysav(i)=y(i)
    enddo 16
    call jacobn(x,y,dfdx,dfdy,nv,nmax)            Evaluate Jacobian.
    if(h.ne.hnext.or.x.ne.xnew)then
        first=.true.
        kopt=kmax
    endif
    reduct=.false.
2   do 18 k=1,kmax
        xnew=x+h
        if(xnew.eq.x)pause 'stepsize underflow in stifbs'
        call simpr(ysav,dydx,dfdx,dfdy,nmax,nv,x,h,nseq(k),yseq,
*            derivs)                              Semi-implicit midpoint rule.
        xest=(h/nseq(k))**2                       The rest of the routine is identical to bsstep.
        call pzextr(k,xest,yseq,y,yerr,nv)
        if(k.ne.1)then
            errmax=TINY
            do 17 i=1,nv
                errmax=max(errmax,abs(yerr(i)/yscal(i)))
            enddo 17
            errmax=errmax/eps
            km=k-1
            err(km)=(errmax/SAFE1)**(1./(2*km+1))
        endif
        if(k.ne.1.and.(k.ge.kopt-1.or.first))then
            if(errmax.lt.1.)goto 4
            if(k.eq.kmax.or.k.eq.kopt+1)then
                red=SAFE2/err(km)
                goto 3
            else if(k.eq.kopt)then
                if(alf(kopt-1,kopt).lt.err(km))then
                    red=1./err(km)
                    goto 3
                endif
            else if(kopt.eq.kmax)then
                if(alf(km,kmax-1).lt.err(km))then
                    red=alf(km,kmax-1)*
*                        SAFE2/err(km)
                    goto 3
                endif
            else if(alf(km,kopt).lt.err(km))then
                red=alf(km,kopt-1)/err(km)
                goto 3
            endif
        endif
    enddo 18
3   red=min(red,REDMIN)
    red=max(red,REDMAX)
    h=h*red
    reduct=.true.
    goto 2
```

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

```
4   x=xnew
    hdid=h
    first=.false.
    wrkmin=1.e35
    do 19 kk=1,km
        fact=max(err(kk),SCALMX)
        work=fact*a(kk+1)
        if(work.lt.wrkmin)then
            scale=fact
            wrkmin=work
            kopt=kk+1
        endif
    enddo 19
    hnext=h/scale
    if(kopt.ge.k.and.kopt.ne.kmax.and..not.reduct)then
        fact=max(scale/alf(kopt-1,kopt),SCALMX)
        if(a(kopt+1)*fact.le.wrkmin)then
            hnext=h/fact
            kopt=kopt+1
        endif
    endif
    return
    END
```

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).

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).

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

Sample page from NUMERICAL RECIPES IN FORTRAN 77: THE ART OF SCIENTIFIC COMPUTING (ISBN 0-521-43064-X)
Copyright (C) 1986-1992 by Cambridge University Press. Programs Copyright (C) 1986-1992 by Numerical Recipes Software.
Permission is granted for internet users to make one paper copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited. To order Numerical Recipes books or CDROMs, visit website
http://www.nr.com or call 1-800-872-7423 (North America only), or send email to directcustserv@cambridge.org (outside North America).