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

CITED REFERENCES AND FURTHER READING:

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

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

16.4 Richardson Extrapolation and the Bulirsch-Stoer Method

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

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

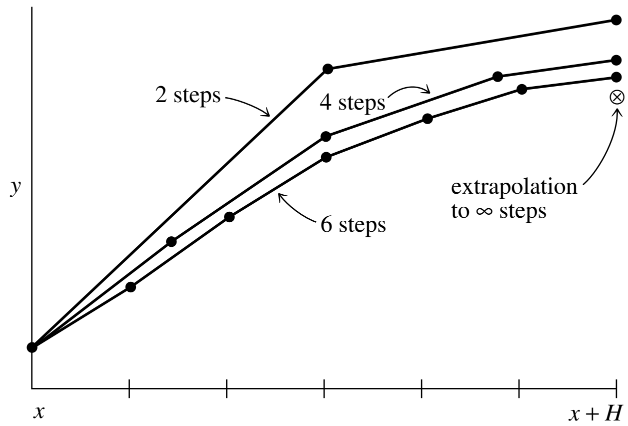

Figure 16.4.1. Richardson extrapolation as used in the Bulirsch-Stoer method. A large interval H is spanned by different sequences of finer and finer substeps. Their results are extrapolated to an answer that is supposed to correspond to infinitely fine substeps. In the Bulirsch-Stoer method, the integrations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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\alpha(k,q) = \epsilon^{\frac{A_{k+1}-A_{q+1}}{(2k+1)(A_{q+1}-A_1+1)}} \qquad \text{for} \quad k < q \tag{16.4.10}$$

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

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

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

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

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

or

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

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

The problem of stepsize reduction is handled by computing stepsize estimates

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
#include "nrutil.h"

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

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

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

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

```
#include "nrutil.h"

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

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

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

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

CITED REFERENCES AND FURTHER READING:

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

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

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

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

16.5 Second-Order Conservative Equations

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

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

As usual, y can denote a vector of values.

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

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