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

CITED REFERENCES AND FURTHER READING:

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

Cash, J.R., and Karp, A.H. 1990, *ACM Transactions on Mathematical Software*, vol. 16, pp. 201–222. [2]

Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121.

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

16.3 Modified Midpoint Method

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
#include "nrutil.h"

void mmid(float y[], float dydx[], int nvar, float xs, float htot, int nstep,
    float yout[], void (*derivs)(float, float[], float[]))
```
Modified midpoint step. At `xs`, input the dependent variable vector `y[1..nvar]` and its derivative vector `dydx[1..nvar]`. Also input is `htot`, the total step to be made, and `nstep`, the number of substeps to be used. The output is returned as `yout[1..nvar]`, which need not be a distinct array from `y`; if it is distinct, however, then `y` and `dydx` are returned undamaged.
```
{
    int n,i;
    float x,swap,h2,h,*ym,*yn;
```

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

CITED REFERENCES AND FURTHER READING:

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

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

16.4 Richardson Extrapolation and the Bulirsch-Stoer Method

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

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