```
    dv=vector(1,nvar);
    for (i=1;i<=nvar;i++) {          Load starting values.
        v[i]=vstart[i];
        y[i][1]=v[i];
    }
    xx[1]=x1;
    x=x1;
    h=(x2-x1)/nstep;
    for (k=1;k<=nstep;k++) {         Take nstep steps.
        (*derivs)(x,v,dv);
        rk4(v,dv,nvar,x,h,vout,derivs);
        if ((float)(x+h) == x) nrerror("Step size too small in routine rkdumb");
        x += h;
        xx[k+1]=x;                   Store intermediate steps.
        for (i=1;i<=nvar;i++) {
            v[i]=vout[i];
            y[i][k+1]=v[i];
        }
    }
    free_vector(dv,1,nvar);
    free_vector(vout,1,nvar);
    free_vector(v,1,nvar);
}
```

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, Volume 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.5. [1]

Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 2. [2]

Shampine, L.F., and Watts, H.A. 1977, in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, *Applied Mathematics and Computation*, vol. 5, pp. 93–121. [3]

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

16.2 Adaptive Stepsize Control for Runge-Kutta

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

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

With fourth-order Runge-Kutta, the most straightforward technique by far is *step doubling* (see, e.g., [1]). We take each step twice, once as a full step, then, independently, as two half steps (see Figure 16.2.1). How much overhead is this, say in terms of the number of evaluations of the right-hand sides? Each of the three separate Runge-Kutta steps in the procedure requires 4 evaluations, but the single and double sequences share a starting point, so the total is 11. This is to be compared not to 4, but to 8 (the two half-steps), since — stepsize control aside — we are achieving the accuracy of the smaller (half) stepsize. The overhead cost is therefore a factor 1.375. What does it buy us?

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

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

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

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

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

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

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

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

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

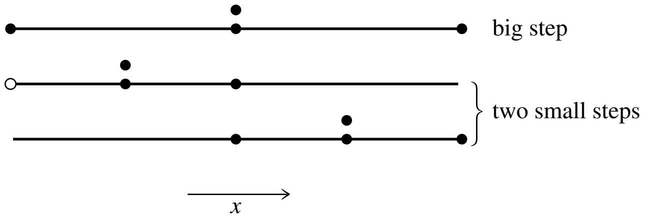

Figure 16.2.1. Step-doubling as a means for adaptive stepsize control in fourth-order Runge-Kutta. Points where the derivative is evaluated are shown as filled circles. The open circle represents the same derivatives as the filled circle immediately above it, so the total number of evaluations is 11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

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

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

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

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

The embedded fourth-order formula is

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

and so the error estimate is

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
#include "nrutil.h"

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

```
{
    int i;
```

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

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

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

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

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

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

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

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

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

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

CITED REFERENCES AND FURTHER READING:

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

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

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

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

16.3 Modified Midpoint Method

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

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

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

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

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