NUMERICAL RECIPES

Adaptive Stepsize Runge-Kutta Integration

William H. Press and Saul A. Teukolsky

Once, when we were graduate students, we had a particularly difficult set of differential equations to integrate. We sought advice from an Eminent Professor who was an expert in numerical analysis. The Eminent Professor asked us what method of integration we had tried. "Runge–Kutta," we replied. "Runge–Kutta! Runge–Kutta!" he exclaimed, banging his head. "That's all you physicists know!"

Now, many years later, it is still true that Runge–Kutta is our favorite integration method for ordinary differential equations. The reason is that we often end up having to integrate a "trivial" set of equations, one where evaluating the right-hand side is cheap, and only moderate accuracy ($\lesssim 10^{-5}$) is required. The only progress since our graduate student days is that we now know enough to switch to the Bulirsch–Stoer method or a predictor–corrector method when computational efficiency is a concern because of high accuracy requirements or complicated right-hand sides. We have also learned that writing a good Runge–Kutta routine is not quite as simple as the formulas given in Abramowitz and Stegun[1] would suggest. An excellent discussion of the pitfalls is given by Shampine and Watts.[2]

Any set of ordinary differential equations (ODEs) can be recast as a set of N coupled *first-order* differential equations by defining suitable functions y_i, $i = 1,2,...,N$ (see, for example, Refs. 3–5). The general form of the equations is

$$\frac{dy_i(x)}{dx} = f_i(x,y_1,...,y_N), \quad i = 1,...,N, \tag{1}$$

where the functions f_i on the right-hand side are known. The basic problem is then, given the y_i at some starting value x_s, to find the y_i's at some final point x_f, or at some discrete list of points (for example, at tabulated intervals).

A good ODE integrator should exert some adaptive control over its own progress, making frequent changes in its stepsize h. 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 2; they can sometimes be factors of 10, 100, or more. Sometimes accuracy may be demanded not directly in the solution itself, but in some related conserved quantity that can be monitored.

William H. Press is a professor of astronomy and physics at Harvard University. Saul A. Teukolsky is a professor of physics and astronomy at Cornell University.

Implementation of adaptive stepsize control requires that the stepping algorithm return information about its performance—most importantly, an estimate of its truncation error. Obviously, the calculation of this information will add to the computational overhead, but the investment will generally be repaid handsomely.

By far the most often used Runge–Kutta method invokes the classical *fourth-order Runge–Kutta formula*:

$$\begin{aligned} k_1 &= hf(x_n,y_n), \\ k_2 &= hf(x_n + h/2, y_n + k_1/2) \\ k_3 &= hf(x_n + h/2, y_n + k_2/2), \\ k_4 &= hf(x_n + h, y_n + k_3), \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5). \end{aligned} \tag{2}$$

With fourth-order Runge-Kutta, the most straightforward technique for adaptive stepsize control is *step doubling*.[6] We take each step twice, once as a full step, then, independently, as two half-steps. 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 four evaluations, but the single and double sequences share a starting point; so the total is 11. This is to be compared not to four, but to eight (the two half-steps), because—stepsize control aside—we are achieving the accurarcy 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 (two steps each of size h). Because the basic method is fourth order, the true solution and the two numerical approximations are related by

$$\begin{aligned} y(x + 2h) &= y_1 + (2h)^5\phi + O(h^6) + \cdots, \\ y(x + 2h) &= y_2 + 2(h^5)\phi + O(h^6) + \cdots, \end{aligned} \tag{3}$$

where, to order h^5, the value ϕ remains constant over the step. [Taylor series expansion tells us that the ϕ is a number which has the order of magnitude of $y^{(5)}(x)/5!$]. The first expression in (3) involves $(2h)^5$ because the stepsize is $2h$, whereas the second expression involves $2(h^5)$ because the error on each step is $h^5\phi$. The difference between the two numerical estimates is a convenient indicator of truncation error

$$\Delta \equiv y_2 - y_1. \tag{4}$$

It is this difference that we shall endeavor to keep to a de-

sired 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 (3) to improve our numerical estimate of the true solution $y(x+2h)$, namely,

$$y(x+2h) = y_2 + \Delta/15 + O(h^6). \tag{5}$$

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: Equation (5) may be fifth-order accurate, but we have no way of monitoring *its* truncation error. Higher order is not always higher accuracy! Use of (5) 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 (5). In the technical literature, use of a procedure like (5) is called "local extrapolation."

It was this step-doubling algorithm that we originally advocated in the *Numerical Recipes* routine rkqc.[3–5] 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 4, more than M function evaluations (though never more than $M+2$) are required. This accounts for the 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

TABLE I. Cash–Karp parameters for embedded Runga-Kutta method.

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

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 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 2 more efficient, have superseded algorithms based on step-doubling.

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

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

The embedded fourth-order formula is

$$\begin{aligned} y^*_{n+1} &= y_n + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4 \\ &\quad + c^*_5 k_5 + c^*_6 k_6 + O(h^5) \end{aligned} \tag{7}$$

and so the error estimate is

$$\Delta \equiv y_{n+1} - y^*_{n+1} = \sum_{i=1}^{6} (c_i - c^*_i) k_i. \tag{8}$$

The particular values of the various constants that we favor are those found by Cash and Karp,[7] and given in Table I. These give a more-efficient method than Fehlberg's original values, with somewhat better error properties.

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 (6)–(8), 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

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2}. \tag{9}$$

Henceforth, we will let Δ_0 denote the *desire* accuracy. Then, Eq. (9) is used in two ways: If Δ_1 is larger than Δ_0 in magnitude, the equation tells how much to decrease the stepsize *when we retry the present (failed) step.* If Δ_1 is smaller than Δ_0, on the other hand, then the equation tells how much we can safely increase the stepsize *for the next step.*

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

How is Δ_0, the desired accuracy, related to some looser prescription like "get a solution good to one part in

10^6? " That can be a subtle question, and it depends on exactly what your application is! You may be dealing with a set of equations in which the dependent variables differ enormously in magnitude. In that case, you probably want to use fractional errors, $\Delta_0 = \epsilon|y|$, where ϵ is a number such as 10^{-6}. 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

$$\Delta_0 = \text{eps} \times \text{yscal(i)}. \tag{10}$$

If you desire constant fractional errors, plug y into 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 $|\text{y(i)}| + h \times |\text{dydx(i)}|$.

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.

Box 1.

```
      SUBROUTINE rkck(y,dydx,n,x,h,yout,yerr,derivs)
      INTEGER n,NMAX
      REAL h,x,dydx(n),y(n),yerr(n),yout(n)
      EXTERNAL derivs
      PARAMETER (NMAX=50)                    Set to the maximum number of functions.
C     USES derivs
        Given values for n variables y and their derivatives dydx known at x, use the fifth-order Cash-
        Karp Runge-Kutta method to advance the solution over an interval h and return the incre-
        mented variables as yout. Also return an estimate of the local truncation error in yout using
        the embedded fourth-order method. The user supplies the subroutine derivs(x,y,dydx)
        which returns derivatives dydx at x.
      INTEGER i
      REAL ak2(NMAX),ak3(NMAX),ak4(NMAX),ak5(NMAX),ak6(NMAX),
     *     ytemp(NMAX),A2,A3,A4,A5,A6,B21,B31,B32,B41,B42,B43,B51,
     *     B52,B53,B54,B61,B62,B63,B64,B65,C1,C3,C4,C6,DC1,DC3,
     *     DC4,DC5,DC6
      PARAMETER (A2=.2,A3=.3,A4=.6,A5=1.,A6=.875,B21=.2,B31=3./40.,
     *     B32=9./40.,B41=.3,B42=-.9,B43=1.2,B51=-11./54.,B52=2.5,
     *     B53=-70./27.,B54=35./27.,B61=1631./55296.,B62=175./512.,
     *     B63=575./13824.,B64=44275./110592.,B65=253./4096.,
     *     C1=37./378.,C3=250./621.,C4=125./594.,C6=512./1771.,
     *     DC1=C1-2825./27648.,DC3=C3-18575./48384.,
     *     DC4=C4-13525./55296.,DC5=-277./14336.,DC6=C6-.25)
      do 11 i=1,n                            First step.
        ytemp(i)=y(i)+B21*h*dydx(i)
      enddo 11
      call derivs(x+A2*h,ytemp,ak2)          Second step.
      do 12 i=1,n
        ytemp(i)=y(i)+h*(B31*dydx(i)+B32*ak2(i))
      enddo 12
      call derivs(x+A3*h,ytemp,ak3)          Third step.
      do 13 i=1,n
        ytemp(i)=y(i)+h*(B41*dydx(i)+B42*ak2(i)+B43*ak3(i))
      enddo 13
      call derivs(x+A4*h,ytemp,ak4)          Fourth step.
      do 14 i=1,n
        ytemp(i)=y(i)+h*(B51*dydx(i)+B52*ak2(i)+B53*ak3(i)+
     *       B54*ak4(i))
      enddo 14
      call derivs(x+A5*h,ytemp,ak5)          Fifth step.
      do 15 i=1,n
        ytemp(i)=y(i)+h*(B61*dydx(i)+B62*ak2(i)+B63*ak3(i)+
     *       B64*ak4(i)+B65*ak5(i))
      enddo 15
      call derivs(x+A6*h,ytemp,ak6)          Sixth step.
      do 16 i=1,n                            Accumulate increments with proper weights.
        yout(i)=y(i)+h*(C1*dydx(i)+C3*ak3(i)+C4*ak4(i)+
     *       C6*ak6(i))
      enddo 16
      do 17 i=1,n
      Estimate error as difference between fourth and fifth order methods.
        yerr(i)=h*(DC1*dydx(i)+DC3*ak3(i)+DC4*ak4(i)+DC5*ak5(i)
     *       +DC6*ak6(i))
      enddo 17
      return
      END
```

Box 2.

```
      SUBROUTINE rkqs(y,dydx,n,x,htry,eps,yscal,hdid,hnext,derivs)
      INTEGER n,NMAX
      REAL eps,hdid,hnext,htry,x,dydx(n),y(n),yscal(n)
      EXTERNAL derivs
      PARAMETER (NMAX=50)                    Maximum number of equations.
C     USES derivs,rkck
        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
        subroutine that computes the right-hand side derivatives.
      INTEGER i
      REAL errmax,h,xnew,yerr(NMAX),ytemp(NMAX),SAFETY,PGROW,
     *     PSHRNK,ERRCON
      PARAMETER (SAFETY=0.9,PGROW=-.2,PSHRNK=-.25,ERRCON=1.89e-4)
        The value ERRCON equals (5/SAFETY)**(1/PGROW), see use below.
      h=htry                                 Set stepsize to the initial trial value.
1     call rkck(y,dydx,n,x,h,ytemp,yerr,derivs)   Take a step.
      errmax=0.                              Evaluate accuracy.
      do 11 i=1,n
        errmax=max(errmax,abs(yerr(i)/yscal(i)))
      enddo 11
      errmax=errmax/eps                      Scale relative to required tolerance.
      if(errmax.gt.1.)then                   Truncation error too large, reduce stepsize.
        h=SAFETY*h*(errmax**PSHRNK)
        if(h.lt.0.1*h)then                   No more than a factor of 10.
          h=.1*h
        endif
        xnew=x+h
        if(xnew.eq.x)pause 'stepsize underflow in rkqs'
        goto 1                               For another try.
      else                                   Step succeeded. Compute size of next step.
        if(errmax.gt.ERRCON)then
          hnext=SAFETY*h*(errmax**PGROW)
        else                                 No more than a factor of 5 increase.
          hnext=5.*h
        endif
        hdid=h
        x=x+h
        do 12 i=1,n
          y(i)=ytemp(i)
        enddo 12
        return
      endif
      END
```

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

$$\Delta_0 = \epsilon h \times \text{dydx(i)}. \tag{11}$$

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 (9). If Δ_0 has an implicit scaling with h, then the exponent 0.20 is no longer corrrect: 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 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 (9) is thus replaced by

$$h_0 = \begin{cases} Sh_1 \left| \dfrac{\Delta_0}{\Delta_1} \right|^{0.20}, & \Delta^0 \geqslant \Delta_1, \\ Sh_1 \left| \dfrac{\Delta_0}{\Delta_1} \right|^{0.25}, & \Delta_0 < \Delta_1. \end{cases} \tag{12}$$

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

In the accompanying box we give the stepper program rkqs that takes one "quality-controlled" Runge–Kutta step. The routine rkqs calls the routine rkck to take a Cash–Karp Runge–Kutta step.

You will notice that the routine rkqs requires you to supply not only a function derivs for calculating the right-hand side, but also values of the derivatives at the starting point. Why not let the routine call derivs for this first value? The answer is that the values of the derivatives are often useful in the routine that calls rkqs, for example to help choose an initial stepsize or to scale y.

The routine rkqs is intended to be called by a "driver" routine such as *Numerical Recipes*' odeint. The driver starts and stops the integration, stores intermediate results, and generally acts as an interface with the user. You will usually customize the driver for your particular application. ■

In our next column: Fitting Straight Line Data with Errors in Both Coordinates

References

1. M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions* (National Bureau of Standards, Washington, 1964; reprinted by Dover Publications, New York, 1968).
2. L. F. Shampine and H. A. Watts, "The Art of Writing a Runge–Kutta Code, Part I," in *Mathematical Software III*, edited by J. R. Rice (Academic, New York, 1977), pp. 257–275; "The Art of Writing a Runge–Kutta Code, Part II," Appl. Math. Comput. **5**, 93–121 (1979).
3. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes: The Art of Scientific Computing* (Cambridge U. P., New York, 1986).
4. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing* (Cambridge U. P., New York, 1988).
5. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in Pascal: The Art of Scientific Computing* (Cambridge U. P., New York, 1989).
6. C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations* (Prentice-Hall, Englewood Cliffs, NJ, 1971), §5.4.
7. J. R. Cash and A. H. Karp, "A Variable Order Runge–Kutta Method for Initial Value Problems with Rapidly Varying Right-Hand Sides," ACM Trans. Math. Software **16**, 201–222 (1990).