```
    for (Int i=0;i<n;i++) yt[i]=y[i]+hh*dydx[i];          First step.
    derivs(xh,yt,dyt);                                    Second step.
    for (Int i=0;i<n;i++) yt[i]=y[i]+hh*dyt[i];
    derivs(xh,yt,dym);                                    Third step.
    for (Int i=0;i<n;i++) {
        yt[i]=y[i]+h*dym[i];
        dym[i] += dyt[i];
    }
    derivs(x+h,yt,dyt);                                   Fourth step.
    for (Int i=0;i<n;i++)                                 Accumulate increments with
        yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);          proper weights.
}
```

The Runge-Kutta method treats every step in a sequence of steps in an identical manner. Prior behavior of a solution is not used in its propagation. This is mathematically proper, since any point along the trajectory of an ordinary differential equation can serve as an initial point. The fact that all steps are treated identically also makes it easy to incorporate Runge-Kutta into relatively simple "driver" schemes.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions* (Washington: National Bureau of Standards); reprinted 1968 (New York: Dover); online at http://numerical.recipes/aands, §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, "The Art of Writing a Runge-Kutta Code, Part I," in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, "The Art of Writing a Runge-Kutta Code. II," *Applied Mathematics and Computation*, vol. 5, pp. 93–121.[3]

17.2 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 17.2.1). 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 (two steps each of size h). Since the basic method is fourth order, the true solution and the two numerical approximations are related by

$$\begin{aligned} y(x + 2h) &= y_1 + (2h)^5\phi + O(h^6) + \dots \\ y(x + 2h) &= y_2 + 2(h^5)\phi + O(h^6) + \dots \end{aligned} \tag{17.2.1}$$

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 (17.2.1) 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,

$$\Delta \equiv y_2 - y_1 \tag{17.2.2}$$

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

$$y(x + 2h) = y_2 + \frac{\Delta}{15} + O(h^6) \tag{17.2.3}$$

This estimate is accurate to *fifth order*, one order higher than the original Runge-Kutta steps (Richardson extrapolation again!). However, we can't have our cake and eat it too: (17.2.3) may be fifth-order accurate, but we have no way of monitoring *its* truncation error. Higher order is not always higher accuracy! Use of (17.2.3) 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 (17.2.3). In the technical literature, use of a procedure like (17.2.3) is called "local extrapolation."

Step doubling has been superseded by a more efficient stepsize adjustment algorithm based on *embedded Runge-Kutta formulas*, originally invented by Merson and popularized in a method of Fehlberg. An interesting fact about Runge-Kutta formulas is that for orders M higher than four, more than M function evaluations 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 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

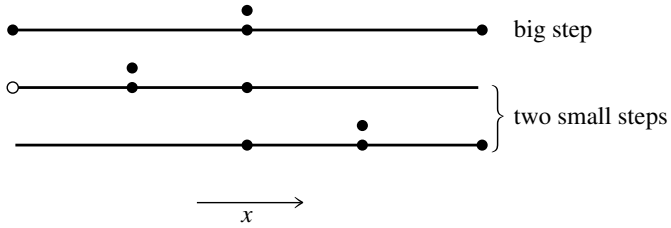

Figure 17.2.1. Step doubling as a means for adaptive stepsize control in fourth-order Runge-Kutta. Points where the derivative is evaluated are shown as filled circles. The open circle represents the same derivatives as the filled circle immediately above it, so the total number of evaluations is 11 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.

stepsize. Since Fehlberg's original formula, many other embedded Runge-Kutta formulas have been found.

As an aside, the general question of how many function evaluations are required for a Runge-Kutta method of a given order is still open. Order 5 requires 6 function evaluations, order 6 requires 7, order 7 requires 9, order 8 requires 11. It is known that for order $M \geq 8$, at least $M + 3$ evaluations are required. The highest order explicitly constructed method so far is order 10, with 17 evaluations. The calculation of the coefficients of these high-order methods is very complicated.

We will spend most of this section setting up an efficient fifth-order Runge-Kutta method, coded in the routine `StepperDopr5`. This will allow us to explore the various issues that have to be dealt with in any Runge-Kutta scheme. However, ultimately you should not use this routine except for low accuracy requirements ($\lesssim 10^{-3}$) or trivial problems. Use the more efficient higher-order Runge-Kutta code `StepperDopr853` or the Bulirsch-Stoer code `StepperBS`.

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

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + c_2 h, y_n + a_{21} k_1) \\ &\cdots \\ k_6 &= hf(x_n + c_6 h, y_n + a_{61} k_1 + \cdots + a_{65} k_5) \\ y_{n+1} &= y_n + b_1 k_1 + b_2 k_2 + b_3 k_3 + b_4 k_4 + b_5 k_5 + b_6 k_6 + O(h^6) \end{aligned} \tag{17.2.4}$$

The embedded fourth-order formula is

$$y^*_{n+1} = y_n + b^*_1 k_1 + b^*_2 k_2 + b^*_3 k_3 + b^*_4 k_4 + b^*_5 k_5 + b^*_6 k_6 + O(h^5) \tag{17.2.5}$$

and so the error estimate is

$$\Delta \equiv y_{n+1} - y^*_{n+1} = \sum_{i=1}^{6} (b_i - b^*_i) k_i \tag{17.2.6}$$

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

We said that the Dormand-Prince method needs six function evaluations per step, yet the table on the next page shows seven and the sums in equations (17.2.5) and (17.2.6) should really go up to $i = 7$. What's going on? The idea is to use

Dormand-Prince 5(4) Parameters for Embedded Runga-Kutta Method

i	c_i	a_{ij}						b_i	b_i^*
1								$\frac{35}{384}$	$\frac{5179}{57600}$
2	$\frac{1}{5}$	$\frac{1}{5}$						0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					$\frac{500}{1113}$	$\frac{7571}{16695}$
4	$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				$\frac{125}{192}$	$\frac{393}{640}$
5	$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			$-\frac{2187}{6784}$	$-\frac{92097}{339200}$
6	1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$		$\frac{11}{84}$	$\frac{187}{2100}$
7	1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0	$\frac{1}{40}$
$j =$		1	2	3	4	5	6		

y_{n+1} itself to provide a seventh stage. Because $f(x_n + h, y_{n+1})$ has to be evaluated anyway to start the next step, this costs nothing (unless the step is rejected because the error is too big). This trick is called FSAL (first-same-as-last). You can see in the table that the coefficients in the last row are the same as the b_i column.

Now that we know, at least approximately, what our error is, we need to consider how to keep it within desired bounds. We require

$$|\Delta| = |y_{n+1} - y^*_{n+1}| \leq \texttt{scale} \tag{17.2.7}$$

where

$$\texttt{scale} = \texttt{atol} + |y|\,\texttt{rtol} \tag{17.2.8}$$

Here `atol` is the absolute error tolerance and `rtol` is the relative error tolerance. (Practical detail: In a code, you use $\max(|y_n|, |y_{n+1}|)$ for $|y|$ in the above formula in case one of them is close to zero.)

Our notation hides the fact that Δ is actually a vector of desired accuracies, Δ_i, one for each equation in the set of ODEs. In practice one takes some norm of the vector Δ. While taking the maximum component value is fine (i.e., rescaling the stepsize according to the needs of the "worst-offender" equation), we will use the usual Euclidean norm. Also, while `atol` and `rtol` could be different for each component of y, we will take them as constant. So define

$$\texttt{err} = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}\left(\frac{\Delta_i}{\texttt{scale}_i}\right)^2} \tag{17.2.9}$$

and accept the step if $\texttt{err} \leq 1$, otherwise reject it.

What is the relation between the scaled error `err` and h? According to (17.2.4) – (17.2.5), Δ scales as h^5 and hence so does `err`. If we take a step h_1 and produce an error $\texttt{err}_1$, therefore, the step h_0 that *would have given* some other value $\texttt{err}_0$ is readily estimated as

$$h_0 = h_1 \left|\frac{\texttt{err}_0}{\texttt{err}_1}\right|^{1/5} \tag{17.2.10}$$

Let $\texttt{err}_0$ denote the desired error, which is 1 in an efficient integration. Then

equation (17.2.10) is used in two ways: If $\texttt{err}_1$ is larger than 1 in magnitude, the equation tells how much to decrease the stepsize *when we retry the present (failed) step*. If $\texttt{err}_1$ is smaller than 1, on the other hand, then the equation tells how much we can safely increase the stepsize *for the next step*. Local extrapolation means that we use the fifth-order value y_{n+1}, even though the error estimate actually applies to the fourth-order value y^*_{n+1}.

How is the quantity `err` 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, $\texttt{atol} = 0$, $\texttt{rtol} = \epsilon$, 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 $\texttt{atol} = \texttt{rtol} = \epsilon$. This latter choice is the safest in general, and should usually be your first choice.

Here is a more technical point. 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 *more steps* between your starting and ending values of x. In such a case you might want to set `scale` proportional to h, typically to something like

$$\texttt{scale} = \epsilon h \times \texttt{dydx[i]} \tag{17.2.11}$$

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 (17.2.10). The exponent $1/5$ 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 `scale` is also altered to this new h_1 value. Instead of $1/5$, we must scale by the exponent $1/4$ for things to work out.

Error control that tries to constrain the global error by setting the scale factor proportional to h is called "error per unit step," as opposed to the original "error per step" method. As a point of principle, controlling the global error by controlling the local error is very difficult. The global error at any point is the sum of the global error up to the start of the last step plus the local error of that step. This cumulative nature of the global error means it depends on things that cannot always be controlled, like stability properties of the differential equation. Accordingly, we recommend the straightforward "error per step" method in most cases. If you want to estimate the global error of your solution, you have to integrate again with a reduced tolerance and use the change in the solution as an estimate of the global error. This works *if* the stepsize controller produces errors roughly proportional to the tolerance, which is not always guaranteed.

Because our error estimates are not exact, but only accurate to the leading order in h, we are advised to put in a safety factor S that is a few percent smaller than unity. Equation (17.2.10) (with $\texttt{err}_0 = 1$ and the subscripts $1 \to n$ and $0 \to n+1$) is thus replaced by

$$h_{n+1} = S h_n \left(\frac{1}{\texttt{err}_n} \right)^{1/5} \tag{17.2.12}$$

Moreover, experience shows that it is not wise to let the stepsize increase or decrease too fast, and not to let the stepsize increase at all if the previous step was rejected. In `StepperDopr5`, the stepsize cannot increase by more than a factor of 10 nor decrease by more than a factor of 5 in a single step.

17.2.1 PI Stepsize Control

One situation in which the above stepsize controller has difficulty is when the stepsize is being limited by the stability properties of the integration method, rather than the accuracy of the individual steps. (We will see more about this in §17.5 on stiff differential equations.) The stepsize increases slowly as successive steps are accepted, until the method becomes unstable. The controller responds to the sudden increase in the error by cutting the stepsize drastically, and the cycle repeats itself. Similar problems can occur when the solution to the differential equation enters a region with drastically different behavior than the previous region. A long sequence of alternating accepted and rejected steps ensues. Since rejected steps are expensive, it is worth improving the stepsize control.

The most effective way to do this seems to be to use ideas from *control theory*. The integration routine and the differential equation play the role of the *process*, like a chemical plant manufacturing a product. The stepsize h is the input and the error estimate `err` is the output. (The numerical solution is also output, but it is not used for stepsize control.) The *controller* is the stepsize control algorithm. It tries to hold the error at the prescribed tolerance by varying the stepsize. Deriving an improved stepsize controller from control theory ideas is beyond our scope here, so we will introduce some basic concepts and then refer you to the literature for derivations and a fuller explanation [6-8].

The standard stepsize controller (17.2.12), when expressed in the language of control theory, is known as an *integrating controller*, with $\log h$ as the discrete control variable. This means that the control variable is obtained by "integrating" the control error signal. It is well known in control theory that more stable control can be achieved by adding an additional term *proportional* to the control error. This is called a PI controller, where the P stands for proportional feedback and the I for integral feedback. Instead of (17.2.12), the resulting algorithm takes the simple form

$$h_{n+1} = S h_n \mathtt{err}_n^{-\alpha} \mathtt{err}_{n-1}^{\beta} \tag{17.2.13}$$

Typically α and β should be scaled as $1/k$, where k is the exponent of h in `err` ($k = 5$ for a fifth-order method). Setting $\alpha = 1/k$, $\beta = 0$ recovers the classical controller (17.2.12). Nonzero β improves the stability but loses some efficiency for "easy" parts of the solution. A good compromise [6] is to set

$$\beta \approx 0.4/k, \qquad \alpha \approx 0.7/k = 1/k - 0.75\beta \tag{17.2.14}$$

17.2.2 Dense Output

Adaptive stepsize control means the algorithm marches along producing y values at x's that it chooses itself. What if you want output at values that you specify? The simplest option is just to integrate from one desired output point to the next. But if you specify a lot of output points, this is inefficient: The code has to take steps

based on where you want output, rather than the "natural" stepsizes it would like to choose. High-order methods like to take large steps for smooth solutions, so the problem is especially acute in this case.

The solution is to find an *interpolation* method that uses information produced during the integration and is of an order comparable to the order of the method so that full accuracy of the solution is preserved. This is called providing a *dense output* method.

For example, any method has available y and $dy/dx = f$ at the beginning and end of the step. These four quantities specify a cubic interpolating polynomial:

$$y(x_n+\theta h) = (1-\theta)y_n+\theta y_{n+1}+\theta(\theta-1)[(1-2\theta)(y_{n+1}-y_n)+(\theta-1)hf_n+\theta hf_{n+1}] \tag{17.2.15}$$

where $0 \le \theta \le 1$. Evaluating this polynomial at any θ in the interval gives a value of y that is third-order accurate, as you can verify by Taylor expansion in h. (Equation 17.2.15 is an example of *Hermite interpolation*, which uses both function and derivative values.)

We are interested, however, in integration methods with order higher than three, so higher-order dense output formulas are needed. The general approach for Runge-Kutta methods is to regard the b_i coefficients in (17.2.4) as polynomials in θ instead of constants. This defines a continuous solution,

$$y(x_n + \theta h) = y_n + b_1(\theta)k_1 + b_2(\theta)k_2 + b_3(\theta)k_3 + b_4(\theta)k_4 + b_5(\theta)k_5 + b_6(\theta)k_6 \tag{17.2.16}$$

and we require the polynomials $b_i(\theta)$ to approximate the true solution to the required order. Equation (17.2.15) is a special case of this.

The Dormand-Prince fifth-order method allows dense output of order four without any further function evaluations. This is usually sufficient: The number of steps to get to a typical point scales as $1/h$, so the global error at that point is typically $O(h^5)$ (fourth order). (Fifth-order dense output, needed, for example, for full accuracy in $y'(x_n + \theta h)$, turns out to need two extra function evaluations per step.) `StepperDopr5` contains a dense output option based on the formulas in [3] as simplified in [4].

Dense output simplifies problems where you don't know in advance how far to integrate. You want to locate the position x_c where some condition is satisfied. Examples include integrating the equations of stellar structure out from the center of the star until the pressure goes to zero at the surface, or the study of limit cycles when one integrates until the solution reaches the Poincaré section for the first time. Write the condition as finding the zero of some function:

$$g\,(x, y_i\,(x)) = 0 \tag{17.2.17}$$

Monitor g in the output routine. When g changes sign between two steps, use the dense output routine to supply function values to your favorite root-finding routine, such as bisection or Newton's method.

17.2.3 Implementation

Here follows the implementation of the fifth-order Dormand-Prince method.

stepperdopr5.h

```
template <class D>
struct StepperDopr5 : StepperBase {
Dormand-Prince fifth-order Runge-Kutta step with monitoring of local truncation error to ensure
accuracy and adjust stepsize.
    typedef D Dtype;                          Make the type of derivs available to Odeint.
    VecDoub k2,k3,k4,k5,k6;
    VecDoub rcont1,rcont2,rcont3,rcont4,rcont5;
    VecDoub dydxnew;
    StepperDopr5(VecDoub_IO &yy, VecDoub_IO &dydxx, Doub &xx,
        const Doub atoll, const Doub rtoll, bool dens);
    void step(const Doub htry,D &derivs);
    void dy(const Doub h,D &derivs);
    void prepare_dense(const Doub h,D &derivs);
    Doub dense_out(const Int i, const Doub x, const Doub h);
    Doub error();
    struct Controller {
        Doub hnext,errold;
        bool reject;
        Controller();
        bool success(const Doub err, Doub &h);
    };
    Controller con;
};
```

The constructor simply invokes the base class instructor and initializes variables:

stepperdopr5.h

```
template <class D>
StepperDopr5<D>::StepperDopr5(VecDoub_IO &yy,VecDoub_IO &dydxx,Doub &xx,
    const Doub atoll,const Doub rtoll,bool dens) :
    StepperBase(yy,dydxx,xx,atoll,rtoll,dens), k2(n),k3(n),k4(n),k5(n),k6(n),
    rcont1(n),rcont2(n),rcont3(n),rcont4(n),rcont5(n),dydxnew(n) {
Input to the constructor are the dependent variable y[0..n-1] and its derivative dydx[0..n-1]
at the starting value of the independent variable x. Also input are the absolute and relative
tolerances, atol and rtol, and the boolean dense, which is true if dense output is required.
    EPS=numeric_limits<Doub>::epsilon();
}
```

The `step` method is the actual stepper. It attempts a step, invokes the controller to decide whether to accept the step or try again with a smaller stepsize, and sets up the coefficients in case dense output is needed between x and $x + h$.

stepperdopr5.h

```
template <class D>
void StepperDopr5<D>::step(const Doub htry,D &derivs) {
Attempts a step with stepsize htry. 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.
    Doub h=htry;                              Set stepsize to the initial trial value.
    for (;;) {
        dy(h,derivs);                         Take a step.
        Doub err=error();                     Evaluate accuracy.
        if (con.success(err,h)) break;        Step rejected. Try again with reduced h set
        if (abs(h) <= abs(x)*EPS)                 by controller.
            throw("stepsize underflow in StepperDopr5");
    }
    if (dense)                                Step succeeded. Compute coefficients for dense
        prepare_dense(h,derivs);                  output.
    dydx=dydxnew;                             Reuse last derivative evaluation for next step.
    y=yout;
    xold=x;                                   Used for dense output.
    x += (hdid=h);
    hnext=con.hnext;
}
```

The algorithm routine dy does the six steps plus the seventh FSAL step, and computes y_{n+1} and the error Δ.

stepperdopr5.h

```
template <class D>
void StepperDopr5<D>::dy(const Doub h,D &derivs) {
Given values for n variables y[0..n-1] and their derivatives dydx[0..n-1] known at x, use the
fifth-order Dormand-Prince Runge-Kutta method to advance the solution over an interval h and
store the incremented variables in yout[0..n-1]. Also store an estimate of the local truncation
error in yerr using the embedded fourth-order method.
    static const Doub c2=0.2,c3=0.3,c4=0.8,c5=8.0/9.0,a21=0.2,a31=3.0/40.0,
    a32=9.0/40.0,a41=44.0/45.0,a42=-56.0/15.0,a43=32.0/9.0,a51=19372.0/6561.0,
    a52=-25360.0/2187.0,a53=64448.0/6561.0,a54=-212.0/729.0,a61=9017.0/3168.0,
    a62=-355.0/33.0,a63=46732.0/5247.0,a64=49.0/176.0,a65=-5103.0/18656.0,
    a71=35.0/384.0,a73=500.0/1113.0,a74=125.0/192.0,a75=-2187.0/6784.0,
    a76=11.0/84.0,e1=71.0/57600.0,e3=-71.0/16695.0,e4=71.0/1920.0,
    e5=-17253.0/339200.0,e6=22.0/525.0,e7=-1.0/40.0;
    VecDoub ytemp(n);
    Int i;
    for (i=0;i<n;i++)                          First step.
        ytemp[i]=y[i]+h*a21*dydx[i];
    derivs(x+c2*h,ytemp,k2);                   Second step.
    for (i=0;i<n;i++)
        ytemp[i]=y[i]+h*(a31*dydx[i]+a32*k2[i]);
    derivs(x+c3*h,ytemp,k3);                   Third step.
    for (i=0;i<n;i++)
        ytemp[i]=y[i]+h*(a41*dydx[i]+a42*k2[i]+a43*k3[i]);
    derivs(x+c4*h,ytemp,k4);                   Fourth step.
    for (i=0;i<n;i++)
        ytemp[i]=y[i]+h*(a51*dydx[i]+a52*k2[i]+a53*k3[i]+a54*k4[i]);
    derivs(x+c5*h,ytemp,k5);                   Fifth step.
    for (i=0;i<n;i++)
        ytemp[i]=y[i]+h*(a61*dydx[i]+a62*k2[i]+a63*k3[i]+a64*k4[i]+a65*k5[i]);
    Doub xph=x+h;
    derivs(xph,ytemp,k6);                      Sixth step.
    for (i=0;i<n;i++)                          Accumulate increments with proper weights.
        yout[i]=y[i]+h*(a71*dydx[i]+a73*k3[i]+a74*k4[i]+a75*k5[i]+a76*k6[i]);
    derivs(xph,yout,dydxnew);                  Will also be first evaluation for next step.
    for (i=0;i<n;i++) {
        Estimate error as difference between fourth- and fifth-order methods.
        yerr[i]=h*(e1*dydx[i]+e3*k3[i]+e4*k4[i]+e5*k5[i]+e6*k6[i]+e7*dydxnew[i]);
    }
}
```

The routine prepare_dense uses the coefficients of [4] to set up the dense output quantities. Our coding of the dense output is closely based on that of the Fortran code DOPRI5 of [5].

stepperdopr5.h

```
template <class D>
void StepperDopr5<D>::prepare_dense(const Doub h,D &derivs) {
Store coefficients of interpolating polynomial for dense output in rcont1...rcont5.
    VecDoub ytemp(n);
    static const Doub d1=-12715105075.0/11282082432.0,
    d3=87487479700.0/32700410799.0, d4=-10690763975.0/1880347072.0,
    d5=701980252875.0/199316789632.0, d6=-1453857185.0/822651844.0,
    d7=69997945.0/29380423.0;
    for (Int i=0;i<n;i++) {
        rcont1[i]=y[i];
        Doub ydiff=yout[i]-y[i];
        rcont2[i]=ydiff;
        Doub bspl=h*dydx[i]-ydiff;
        rcont3[i]=bspl;
        rcont4[i]=ydiff-h*dydxnew[i]-bspl;
```

```
        rcont5[i]=h*(d1*dydx[i]+d3*k3[i]+d4*k4[i]+d5*k5[i]+d6*k6[i]+
            d7*dydxnew[i]);
    }
}
```

The next routine, `dense_out`, uses the coefficients stored by the previous routine to evaluate the solution at an arbitrary point.

stepperdopr5.h

```
template <class D>
Doub StepperDopr5<D>::dense_out(const Int i,const Doub x,const Doub h) {
Evaluate interpolating polynomial for y[i] at location x, where xold ≤ x ≤ xold + h.
    Doub s=(x-xold)/h;
    Doub s1=1.0-s;
    return rcont1[i]+s*(rcont2[i]+s1*(rcont3[i]+s*(rcont4[i]+s1*rcont5[i])));
}
```

The `error` routine converts Δ into the scaled quantity `err`.

stepperdopr5.h

```
template <class D>
Doub StepperDopr5<D>::error() {
Use yerr to compute norm of scaled error estimate. A value less than one means the step was
successful.
    Doub err=0.0,sk;
    for (Int i=0;i<n;i++) {
        sk=atol+rtol*MAX(abs(y[i]),abs(yout[i]));
        err += SQR(yerr[i]/sk);
    }
    return sqrt(err/n);
}
```

Finally, the `controller` tests whether `err` ≤ 1 and adjusts the stepsize. The default setting is `beta` $= 0$ (no PI control). Set `beta` to 0.04 or 0.08 to turn on PI control.

stepperdopr5.h

```
template <class D>
StepperDopr5<D>::Controller::Controller() : reject(false), errold(1.0e-4) {}
Step size controller for fifth-order Dormand-Prince method.
template <class D>
bool StepperDopr5<D>::Controller::success(const Doub err,Doub &h) {
Returns true if err ≤ 1, false otherwise. If step was successful, sets hnext to the estimated
optimal stepsize for the next step. If the step failed, reduces h appropriately for another try.
    static const Doub beta=0.0,alpha=0.2-beta*0.75,safe=0.9,minscale=0.2,
        maxscale=10.0;
    Set beta to a nonzero value for PI control. beta = 0.04–0.08 is a good default.
    Doub scale;
    if (err <= 1.0) {                       Step succeeded. Compute hnext.
        if (err == 0.0)
            scale=maxscale;
        else {                              PI control if beta ≠ 0.
            scale=safe*pow(err,-alpha)*pow(errold,beta);
            if (scale<minscale) scale=minscale;   Ensure minscale ≤ hnext/h ≤ maxscale.
            if (scale>maxscale) scale=maxscale;
        }
        if (reject)                         Don't let step increase if last one was re-
            hnext=h*MIN(scale,1.0);                                   jected.
        else
            hnext=h*scale;
        errold=MAX(err,1.0e-4);             Bookkeeping for next call.
        reject=false;
        return true;
    } else {                                Truncation error too large, reduce stepsize.
```

```
            scale=MAX(safe*pow(err,-alpha),minscale);
            h *= scale;
            reject=true;
            return false;
        }
    }
```

A warning: Don't be too greedy in specifying `atol` and `rtol`. 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. (On a supercomputer, you guard against this catastrophic loss of your time allocation by signaling on abnormally small stepsizes or on the dependent variable vector remaining unchanged from step to step. On a desktop computer, you guard against it by not taking too long a lunch hour while the program is running.)

17.2.4 Dopr853 — An Eighth-Order Method

Once you understand the above implementation of `StepperDopr5`, then you have the framework for essentially any Runge-Kutta method. For production work, we recommend that you use the following method, `StepperDopr853`. It is again a Dormand-Prince embedded method, this time of eighth order that uses 12 function evaluations. The original version used a sixth-order embedded method for error estimation. However, it turned out that the error estimation was not robust in certain circumstances because the last evaluation point happened not to be used in computing the error. Accordingly, Hairer, Nörsett, and Wanner [5] constructed both fifth-order and third-order embedded methods that use the last point. Then the error is estimated as

$$\mathtt{err} = \mathtt{err}_5 \frac{\mathtt{err}_5}{\sqrt{0.01(\mathtt{err}_3)^2 + (\mathtt{err}_5)^2}} \tag{17.2.18}$$

Most of the time, $\mathtt{err}_5 \ll \mathtt{err}_3$, so $\mathtt{err} = O(h^8)$. If the estimation breaks down so that either $\mathtt{err}_3$ gets small or $\mathtt{err}_5$ gets large, then `err` will still give a reasonable basis for stepsize control. This method has worked well in practice and is the basis for the "853" in the name of the method.

For an eighth-order method we would like seventh-order dense output. It turns out this requires three more function evaluations. Our coding of the dense output follows closely the Fortran implementation of [5]. Since the code is somewhat lengthy, but basically similar to `StepperDopr5`, we give it as `StepperDopr853` in a Webnote [9].

CITED REFERENCES AND FURTHER READING:

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

Dormand, J.R, and Prince, P.J. 1980, "A Family of Embedded Runge-Kutta Formulae," *Journal of Computational and Applied Mathematics*, vol. 6, pp. 19–26.[2]

Shampine, L.F., and Watts, H.A. 1977, "The Art of Writing a Runge-Kutta Code, Part I," in *Mathematical Software III*, J.R. Rice, ed. (New York: Academic Press), pp. 257–275; 1979, "The Art of Writing a Runge-Kutta Code. II," *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).

Dormand, J.R, and Prince, P.J. 1986, "Runge-Kutta Triples," *Computers and Mathematics with Applications*, vol. 12A, pp. 1007–1017.[3]

Shampine, L.F. 1986, "Some Practical Runge-Kutta Formulas," *Mathematics of Computation*, vol. 46, pp. 135–150.[4]

Hairer, E., Nørsett, S.P., and Wanner, G. 1993, *Solving Ordinary Differential Equations I. Nonstiff Problems*, 2nd ed. (New York: Springer). Fortran codes at http://www.unige.ch/~hairer/software.html.[5]

Gustafsson, K. 1991, "Control Theoretic Techniques for Stepsize Selection in Explicit Runge-Kutta Methods," *ACM Transactions on Mathematical Software*, vol. 17, pp. 533-554.[6]

Hairer, E., Nørsett, S.P., and Wanner, G. 1996, *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*, 2nd ed. (New York: Springer), p. 28.[7]

Söderlind, G. 2003, "Digital Filters in Adaptive Time-stepping," *ACM Transactions on Mathematical Software*, vol. 29, pp. 1–26.[8]

Numerical Recipes Software 2007, "Routine Implementing an Eighth-order Runge-Kutta Method," *Numerical Recipes Webnote No. 20*, at http://numerical.recipes/webnotes?20 [9]

17.3 Richardson Extrapolation and the Bulirsch-Stoer Method

The techniques in this section are for differential equations containing smooth functions. With just three caveats, we believe that the Bulirsch-Stoer method, discussed here, is the best-known way to obtain high accuracy solutions to ordinary differential equations with minimal computational effort. The caveats are these:

- If you have a nonsmooth problem, for example, a differential equation whose right-hand side involves a function that is evaluated by table look-up and interpolation, go back to Runge-Kutta with an 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 a quick-and-dirty, low accuracy solution of a set of equations.
- 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 but touched on in §18.6.
- There *may* be a few problems that are both very smooth and have right-hand sides that are very expensive to evaluate, for which predictor-corrector methods, discussed in §17.6, are the methods of choice.

The methods in this section involve three key ideas. 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