Integration of Ordinary Differential Equations

17.0 Introduction

Problems involving ordinary differential equations (ODEs) can always be reduced to the study of sets of first-order differential equations. For example the second-order equation

\[
\frac{d^2 y}{dx^2} + q(x) \frac{dy}{dx} = r(x)
\]  

(17.0.1)
can be rewritten as two first-order equations,

\[
\frac{dy}{dx} = z(x)
\]

\[
\frac{dz}{dx} = r(x) - q(x)z(x)
\]  

(17.0.2)

where \( z \) is a new variable. This exemplifies the procedure for an arbitrary ODE. The usual choice for the new variables is to let them be just derivatives of each other (and of the original variable). Occasionally, it is useful to incorporate into their definition some other factors in the equation, or some powers of the independent variable, for the purpose of mitigating singular behavior that could result in overflows or increased roundoff error. Let common sense be your guide: If you find that the original variables are smooth in a solution, while your auxiliary variables are doing crazy things, then figure out why and choose different auxiliary variables.

The generic problem in ordinary differential equations is thus reduced to the study of a set of \( N \) coupled first-order differential equations for the functions \( y_i, \ i = 0, 1, \ldots, N - 1 \), having the general form

\[
\frac{dy_i(x)}{dx} = f_i(x, y_0, \ldots, y_{N-1}), \quad i = 0, \ldots, N - 1
\]  

(17.0.3)

where the functions \( f_i \) on the right-hand side are known.

A problem involving ODEs is not completely specified by its equations. Even more crucial in determining how to attack the problem numerically is the nature of
the problem's boundary conditions. Boundary conditions are algebraic conditions on the values of the functions \( y_i \) in (17.0.3). In general they can be satisfied at discrete specified points, but do not hold between those points, i.e., are not preserved automatically by the differential equations. Boundary conditions can be as simple as requiring that certain variables have certain numerical values, or as complicated as a set of nonlinear algebraic equations among the variables.

Usually, it is the nature of the boundary conditions that determines which numerical methods will be feasible. Boundary conditions divide into two broad categories.

- In **initial value problems** all the \( y_i \) are given at some starting value \( x_s \), and it is desired to find the \( y_i \)'s at some final point \( x_f \), or at some discrete list of points (for example, at tabulated intervals).
- In **two-point boundary value problems**, on the other hand, boundary conditions are specified at more than one \( x \). Typically, some of the conditions will be specified at \( x_s \) and the remainder at \( x_f \).

This chapter will consider exclusively the initial value problem, deferring two-point boundary value problems, which are generally more difficult, to Chapter 18.

The underlying idea of any routine for solving the initial value problem is always this: Rewrite the \( dy \)'s and \( dx \)'s in (17.0.3) as finite steps \( \Delta y \) and \( \Delta x \), and multiply the equations by \( \Delta x \). This gives algebraic formulas for the change in the functions when the independent variable \( x \) is "stepped" by one "stepsize" \( \Delta x \). In the limit of making the stepsize very small, a good approximation to the underlying differential equation is achieved. Literal implementation of this procedure results in Euler's method (equation 17.1.1, below), which is, however, not recommended for any practical use. Euler's method is conceptually important, however; one way or another, practical methods all come down to this same idea: Add small increments to your functions corresponding to derivatives (right-hand sides of the equations) multiplied by stepsizes.

In this chapter we consider three major types of practical numerical methods for solving initial value problems for ODEs:

- **Runge-Kutta methods**
- **Richardson extrapolation and its particular implementation as the Bulirsch-Stoer method**
- **predictor-corrector methods, also known as multistep methods.**

A brief description of each of these types follows.

1. **Runge-Kutta** methods propagate a solution over an interval by combining the information from several Euler-style steps (each involving one evaluation of the right-hand \( f \)'s), and then using the information obtained to match a Taylor series expansion up to some higher order.

2. **Richardson extrapolation** uses the powerful idea of extrapolating a computed result to the value that would have been obtained if the stepsize had been very much smaller than it actually was. In particular, extrapolation to zero stepsize is the desired goal. The first practical ODE integrator that implemented this idea was developed by Bulirsch and Stoer, and so extrapolation methods are often called Bulirsch-Stoer methods.

3. **Predictor-corrector methods** or **multistep methods** store the solution along the way, and use those results to extrapolate the solution one step advanced; they
then correct the extrapolation using derivative information at the new point. These are best for very smooth functions.

Runge-Kutta used to be what you used when (i) you didn’t know any better, or (ii) you had an intransigent problem where Bulirsch-Stoer was failing, or (iii) you had a trivial problem where computational efficiency was of no concern. However, advances in Runge-Kutta methods, particularly the development of higher-order methods, have made Runge-Kutta competitive with the other methods in many cases. Runge-Kutta succeeds virtually always; it is usually the fastest method when evaluating $f_i$ is cheap and the accuracy requirement is not ultra-stringent ($\lesssim 10^{-10}$), or in general when moderate accuracy ($\lesssim 10^{-5}$) is required. Predictor-corrector methods have a relatively high overhead and so come into their own only when evaluating $f_i$ is expensive. However, for many smooth problems, they are computationally more efficient than Runge-Kutta. In recent years, Bulirsch-Stoer has been replacing predictor-corrector in many applications, but it is too soon to say that predictor-corrector is dominated in all cases. However, it appears that only rather sophisticated predictor-corrector routines are competitive. Accordingly, we have chosen not to give an implementation of predictor-corrector in this book. We discuss predictor-corrector further in §17.6, so that you can use a packaged routine knowledgeably should you encounter a suitable problem. In our experience, the relatively simple Runge-Kutta and Bulirsch-Stoer routines we give are adequate for most problems.

Each of the three types of methods can be organized to monitor internal consistency. This allows numerical errors, which are inevitably introduced into the solution, to be controlled by automatic (adaptive) changing of the fundamental stepsize. We always recommend that adaptive stepsize control be implemented, and we will do so below.

In general, all three types of methods can be applied to any initial value problem. Each comes with its own set of debits and credits that must be understood before it is used.

Section 17.5 of this chapter treats the subject of stiff equations, relevant both to ordinary differential equations and also to partial differential equations (Chapter 20).

17.0.1 Organization of the Routines in This Chapter

We have organized the routines in this chapter into three nested levels, enabling modularity and sharing common code wherever possible.

The highest level is the driver object, which starts and stops the integration, stores intermediate results, and generally acts as an interface with the user. There is nothing canonical about our driver object, `odeint`. You should consider it to be an example, and you can customize it for your particular application.

The next level down is a stepper object. The stepper oversees the actual incrementing of the independent variable $x$. It knows how to call the underlying algorithm routine. It may reject the result, set a smaller stepsize, and call the algorithm routine again, until compatibility with a predetermined accuracy criterion has been achieved. The stepper’s fundamental task is to take the largest stepsize consistent with specified performance. Only when this is accomplished does the true power of an algorithm come to light.

All our steppers are derived from a base object called `StepperBase`: `StepperDopr5` and `StepperDopr853` (two Runge-Kutta routines), `StepperBS` and `StepperStoerm` (two Bulirsch-Stoer routines), and `StepperRoss` and `StepperSIE`
Chapter 17. Integration of Ordinary Differential Equations

(for so-called stiff equations).

Standing apart from the stepper, but interacting with it at the same level, is an Output object. This is basically a container into which the stepper writes the output of the integration, but it has some intelligence of its own: It can save, or not save, intermediate results according to several different prescriptions that are specified by its constructor. In particular, it has the option to provide so-called dense output, that is, output at user-specified intermediate points without loss of efficiency.

The lowest or "nitty-gritty" level is the piece we call the algorithm routine. This implements the basic formulas of the method, starts with dependent variables $y_j$ at $x$, and calculates new values of the dependent variables at the value $x + h$. The algorithm routine also yields some information about the quality of the solution after the step. The routine is dumb, however, in that it is unable to make any adaptive decision about whether the solution is of acceptable quality. Each algorithm routine is implemented as a member function $dy(\cdot)$ in its corresponding stepper object.

### 17.0.2 The Odeint Object

It is a real time saver to have a single high-level interface to what are otherwise quite diverse methods. We use the Odeint driver for a variety of problems, notably including garden-variety ODEs or sets of ODEs, and definite integrals (augmenting the methods of Chapter 4). The Odeint driver is template on the stepper. This means that you can usually change from one ODE method to another in just a few keystrokes. For example, changing from the Dormand-Prince fifth-order Runge-Kutta method to Bulirsch-Stoer is as simple as changing the template parameter from StepperDopr5 to StepperBS.

The Odeint constructor simply initializes a bunch of things, including a call to the stepper constructor. The meat is in the integrate routine, which repeatedly invokes the step routine of the stepper to advance the solution from $x_1$ to $x_2$. It also calls the functions of the Output object to save the results at appropriate points.

odeint.h

```
template<class Stepper>
struct Odeint {
  Driver for ODE solvers with adaptive stepsize control. The template parameter should be one of the derived classes of StepperBase defining a particular integration algorithm.
  static const Int MAXSTP=50000;
  Doub EPS;
  Int nok;
  Int nbad;
  Int nvar;
  Doub x1,x2,hmin;
  bool dense;
  VecDoub y,dydx;
  VecDoub &ystart;
  Output &out;
  typename Stepper::Dtype &derivs;
  Stepper s;
  Int nstp;
  Doub x,h;
  Odeint(VecDoub,IO &ystartt,const Doub xx1,const Doub xx2,
   const Doub atol,const Doub rtol,const Doub h1,
   const Doub hminn,Output &outt,typename Stepper::Dtype &derivss); 
  Constructor sets everything up. The routine integrates starting values $ystart\{0..nvar-1\}$ from $xx1$ to $xx2$ with absolute tolerance $atol$ and relative tolerance $rtol$. The quantity $h1$ should be set as a guessed first stepsize, $hmin$ as the minimum allowed stepsize (can be zero). An Output object out should be input to control the saving of intermediate values.
```

On output, \( n_{\text{ok}} \) and \( n_{\text{bad}} \) are the number of good and bad (but retried and fixed) steps taken, and \( y_{\text{start}} \) is replaced by values at the end of the integration interval. \( \text{derivs} \) is the user-supplied routine (function or functor) for calculating the right-hand side derivative.

```cpp
void integrate(); // Does the actual integration.
```

```cpp
template<class Stepper>
Odeint<Stepper>::Odeint(VecDoub_I0 &ystartt, const Doub xx1, const Doub xx2, const Doub atol, const Doub rtol, const Doub h1, const Doub hminn,
Output &out, typename Stepper::Dtype &derivss):
ystart(ystartt),
integrate = void integrate0(derivs);

// Store initial values.
```

```cpp
for (Int i=0; i<nvar; i++)
y[i]=ystart[i];
```

```cpp
The Odeint object doesn’t know in advance which specific stepper object it will be instantiated with. It does, however, rely on the fact that the stepper object will be derived from, and thus have the methods in, this StepperBase object, which serves as the base class for all subsequent ODE algorithms in this chapter:

```cpp
struct StepperBase {
    Base class for all ODE algorithms.
    Doub &x;
    Doub xold;
    VecDoub &y,&dydx;
    Doub atol,rtol;
    bool dense;
    Doub hdid;
    Doub hnext;
}
```

```cpp
stepper.h
```

```cpp
Used for dense output.
```

```cpp
Actual stepsize accomplished by the step routine.
```

```cpp
Stepsize predicted by the controller for the next step.
```
Doub EPS;  
Int n,neqn;  
VecDoub yout,yerr;  
neqn = n except for StepperStoerm.  
StepperBase(VecDoub_I0 &yy, VecDoub_I0 &dydxx, Doub &xx, const Doub atol1,  
const Doub rtol1, bool dens): x(xx),y(yy),dydx(dydxx),atol(atol1),  
rtol(rtol1),dense(dens),n(y.size()),neqn(n),yout(n),yerr(n) {}  
Input to the constructor are the dependent variable vector 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.

17.0.3 The Output Object

Output is controlled by the various constructors in the Output structure. The  
default constructor, with no arguments, suppresses all output. The constructor with  
argument nsave provides dense output provided nsave > 0. This means output  
at values of x of your choosing, not necessarily the natural places that the stepper  
method would land. The output points are nsave + 1 uniformly spaced points  
including x1 and x2. If nsave ≤ 0, output is saved at every integration step, that  
is, only at the points where the stepper happens to land. While most of your needs  
should be met by these options, you should find it easy to modify Output for your  
particular application.

```c
odeint.h
struct Output {
    Structure for output from ODE solver such as Odeint.
    Int kmax;  
    Int nvar;  
    Int nsave;  
    bool dense;  
    Int count;  
    Doub x1,x2,xout,dxout;  
    VecDoub xsaved;  
    MatDoub ysave;  
    Output(): kmax(-1),dense(false),count(0) {}  
    Default constructor gives no output.
    Output(const Int nsavee): kmax(500),nsave(nsavee),count(0),xsave(kmax) {}  
    Constructor provides dense output at nsave equally spaced intervals. If nsave ≤ 0, output  
is saved only at the actual integration steps.
    dense = nsave > 0 ? true : false;
}  
void init(const Int neqn, const Doub xlo, const Doub xhi) {  
    Called by Odeint constructor, which passes neqn, the number of equations, xlo, the starting  
    point of the integration, and xhi, the ending point.
    nvar=neqn;  
    if (kmax == -1) return;
    ysave.resize(nvar,kmax);
    if (dense) {
        x1=xlo;
        x2=xhi;
        xout=x1;
        dxout=(x2-x1)/nsave;
    }
}  
void resize() {  
    Resize storage arrays by a factor of two, keeping saved data.
    Int kold=kmax;  
    kmax *= 2;
    VecDoub tempvec(xsave);
```
17.0.4 A Quick-Start Example

Before we dive deep into the pros and cons of the different stepper types (the meat of this chapter), let’s see how to code the solution of an actual problem. Suppose we want to solve Van der Pol’s equation, which when written in first-order form is

\[
\begin{align*}
y' &= y_1 \\
y_1' &= \frac{(1 - y_1^2)y_1 - y_0}{\epsilon}
\end{align*}
\]  

(17.0.4)

First encapsulate (17.0.4) in a functor (see §1.3.3). Using a functor instead of a bare function gives you the opportunity to pass other information to the function,
such as the values of fixed parameters. Every stepper class in this chapter is accord-
ingenly templated on the type of the functor defining the right-hand side derivatives. For our example, the right-hand side functor looks like:

```cpp
class rhs_van {
    Doub eps;
    rhs_van(Doub eps) : eps(epss) {}%
    void operator() (const Doub x, VecDoub_I &y, VecDoub_0 &dydx) {
        dydx[0]= y[1];
        dydx[1]=((1.0-y[0]*y[0])*y[1]-y[0])/eps;
    }
};
```

The key thing is the line beginning `void operator()`: It *always* should have this form, with the definition of `dydx` following. Here we have chosen to specify $\epsilon$ as a parameter in the constructor so that the main program can easily pass a specific value to the right-hand side. Alternatively, you could have omitted the constructor, relying on the compiler-supplied default constructor, and hard-coded a value of $\epsilon$ in the routine. Note, of course, that there is nothing special about the name `rhs_van`.

We will integrate from 0 to 2 with initial conditions $y_0 = 2$, $y_1 = 0$ and with $\epsilon = 10^{-3}$. Then your main program will have declarations like the following:

```cpp
const Int nvar=2;
const Doub atol=1.0e-3, rtol=atol, h1=0.01, hmin=0.0, x1=0.0, x2=2.0;
VecDoub ystart(nvar);
ystart[0]=2.0;
ystart[1]=0.0;
Output out (20); Dense output at 20 points plus x1.
rhs_van d(1.0e-3); Declare `d` as a `rhs_van` object.
Odeint<StepperDopr5<rhs_van> > ode(ystart,x1,x2,atol,rtol,h1,hmin,out,d);
od.e_integrate();
```

Note how the `Odeint` object is templated on the stepper, which in turn is templated on the derivative object, `rhs_van` in this case. The space between the two closing angle brackets is necessary; otherwise the compiler parses `>` as the right-shift operator!

The number of good steps taken is available in `ode.nok` and the number of bad steps in `ode.nbad`. The output, which is equally spaced, can be printed by statements like

```cpp
for (Int i=0;i<out.count;i++)
    cout << out.xsave[i] << " " << out.ysave[0][i] << " " <<
         out.ysave[1][i] << endl;
```

You can alternatively save output at the actual integration steps by the declaration

```cpp
Output out(-1);
```

or suppress all saving of output with

```cpp
Output out;
```

In this case, the solution values at the endpoint are available in `ystart[0]` and `ystart[1]`, overwriting the starting values.

CITED REFERENCES AND FURTHER READING:

Runge-Kutta Method

The formula for the Euler method is

\[ y_{n+1} = y_n + hf(x_n, y_n) \]  \hspace{1cm} (17.1.1)

which advances a solution from \( x_n \) to \( x_{n+1} = x_n + h \). The formula is unsymmetrical: It advances the solution through an interval \( h \), but uses derivative information only at the beginning of that interval (see Figure 17.1.1). That means (and you can verify by expansion in power series) that the step's error is only one power of \( h \) smaller than the correction, i.e., \( O(h^2) \) added to (17.1.1).

There are several reasons that Euler's method is not recommended for practical use, among them, (i) the method is not very accurate when compared to other, fancier, methods run at the equivalent stepsize, and (ii) neither is it very stable (see §17.5 below).

Consider, however, the use of a step like (17.1.1) to take a "trial" step to the midpoint of the interval. Then use the values of both \( x \) and \( y \) at that midpoint to compute the "real" step across the whole interval. Figure 17.1.2 illustrates the idea. In equations,

\[ k_1 = hf(x_n, y_n) \]
\[ k_2 = hf \left( x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1 \right) \]
\[ y_{n+1} = y_n + k_2 + O(h^3) \] \hspace{1cm} (17.1.2)

As indicated in the error term, this symmetrization cancels out the first-order error term, making the method second order. [A method is conventionally called \( n \)th order if its error term is \( O(h^{n+1}) \).] In fact, (17.1.2) is called the second-order Runge-Kutta or midpoint method.

We needn't stop there. There are many ways to evaluate the right-hand side \( f(x, y) \) that all agree to first order, but that have different coefficients of higher-order error terms. Adding up the right combination of these, we can eliminate the error terms order by order. That is the basic idea of the Runge-Kutta method. Abramowitz and Stegun [1] and Gear [2] give various specific formulas that derive from this basic idea. By far the most often used is the classical fourth-order Runge-Kutta formula,
Figure 17.1.1. Euler's method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

Figure 17.1.2. Midpoint method. Second-order accuracy is obtained by using the initial derivative at each step to find a point halfway across the interval, then using the midpoint derivative across the full width of the interval. In the figure, filled dots represent final function values, while open dots represent function values that are discarded once their derivatives have been calculated and used.

which has a certain sleekness of organization about it:

\[ k_1 = hf(x_n, y_n) \]
\[ k_2 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \]
\[ k_3 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \]
\[ k_4 = hf(x_n + h, y_n + k_3) \]
\[ y_{n+1} = y_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + O(h^5) \] (17.1.3)

The fourth-order Runge-Kutta method requires four evaluations of the right-hand side per step \( h \) (see Figure 17.1.3). This will be superior to the midpoint method (17.1.2) if at least twice as large a step is possible with (17.1.3) for the same accuracy. Is that so? The answer is: often, perhaps even usually, but surely not always! This takes us back to a central theme, namely that high order does not always mean high accuracy. The statement “fourth-order Runge-Kutta is generally superior to second-order” is a true one, but as much a statement about the kind of problems that people solve as a statement about strict mathematics.

For many scientific users, fourth-order Runge-Kutta is not just the first word
17.1 Runge-Kutta Method

Figure 17.1.3. Fourth-order Runge-Kutta method. In each step the derivative is evaluated four times: once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the final function value (shown as a filled dot) is calculated. (See text for details.)

on ODE integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm. Keep in mind, however, that the old workhorse’s last trip may well be to take you to the poorhouse: Newer Runge-Kutta methods are much more efficient, and Bulirsch-Stoer or predictor-corrector methods can be even more efficient for problems where very high accuracy is a requirement. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields. However, even the old workhorse is more nimble with new horseshoes. In §17.2 we will give a modern implementation of a Runge-Kutta method that is quite competitive as long as very high accuracy is not required. An excellent discussion of the pitfalls in constructing a good Runge-Kutta code is given in [3].

Here is the routine rk4 for carrying out one classical Runge-Kutta step on a set of n differential equations. This routine is completely separate from the various stepper routines introduced in the previous section and given in the rest of the chapter. It is meant for only the most trivial applications. You input the values of the independent variables, and you get out new values that are stepped by a stepsize h (which can be positive or negative). You will notice that the routine requires you to supply not only 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 will become clear only in the next section, but in brief is this: This call may not be your only one with these starting conditions. You may have taken a previous step with too large a stepsize, and this is your replacement. In that case, you do not want to call derivs unnecessarily at the start. Note that the routine that follows has, therefore, only three calls to derivs.

```c
void rk4(VecDoub_I &y, VecDoub_I &dydx, const Doub x, const Doub h,
        VecDoub_0 &yout, void derivs(const Doub, VecDoub_I &, VecDoub_0 &))
Given values for the variables y[0..n-1] and their derivatives dydx[0..n-1] known at x, use
the fourth-order Runge-Kutta method to advance the solution over an interval h and return
the incremented variables as yout[0..n-1]. The user supplies the routine derivs(x,y,dydx),
which returns derivatives dydx at x.
{    
    Int n=y.size();
    VecDoub dym(n),dyt(n),yt(n);
    Doub hh=h*0.5;
    Doub h6=h/6.0;
    Doub xh=x+hh;
```
Chapter 17. Integration of Ordinary Differential Equations

for (Int i=0; i<n; i++) yt[i]=y[i]+hh*dydx[i];
derivs(xh,yt,dyt);
for (Int i=0; i<n; i++) yt[i]=y[i]+hh*dyt[i];
derivs(xh,yt,dym);
for (Int i=0; i<n; i++) {
y[i]=y[i]+h*dym[i];
dym[i] += dyt[i];
}
derivs(x+h,yt,dyt);
for (Int i=0; i<n; i++)
} yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);

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:

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

\[
y(x + 2h) = y_1 + (2h)^5 \phi + O(h^6) + \ldots
\]
\[
y(x + 2h) = y_2 + 2(h^5)\phi + O(h^6) + \ldots
\]  

(17.2.1)

where, to order \( h^5 \), the value \( \phi \) remains constant over the step. [Taylor series expansion tells us the \( \phi \) 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,

\[
\Delta \equiv y_2 - y_1
\]  

(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,

\[
y(x + 2h) = y_2 + \frac{\Delta}{15} + O(h^6)
\]  

(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 \( \Delta \) 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
stepsizes. 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 ($\leq 10^{-3}$) 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

$$
k_1 = hf(x_n, y_n)
$$

$$
k_2 = hf(x_n + c_2h, y_n + a_{21}k_1)
$$

$$
k_6 = hf(x_n + c_6h, y_n + a_{61}k_1 + \cdots + a_{65}k_5)
$$

$$
y_{n+1} = y_n + b_1k_1 + b_2k_2 + b_3k_3 + b_4k_4 + b_5k_5 + b_6k_6 + O(h^6)
$$

The embedded fourth-order formula is

$$
y^*_{n+1} = y_n + b_1^*k_1 + b_2^*k_2 + b_3^*k_3 + b_4^*k_4 + b_5^*k_5 + b_6^*k_6 + O(h^5)
$$

and so the error estimate is

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

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

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.
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

\[
|\Delta| = |y_{n+1} - y_{n+1}^*| \leq \text{scale}
\]

(17.2.7)

where

\[
\text{scale} = \text{atol} + |y|\text{rtol}
\]

(17.2.8)

Here \(\text{atol}\) is the absolute error tolerance and \(\text{rtol}\) 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 \(\Delta\) is actually a vector of desired accuracies, \(\Delta_i\), one for each equation in the set of ODEs. In practice one takes some norm of the vector \(\Delta\). 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 \(\text{atol}\) and \(\text{rtol}\) could be different for each component of \(y\), we will take them as constant. So define

\[
\text{err} = \sqrt{\frac{1}{N} \sum_{i=0}^{N-1} \left( \frac{\Delta_i}{\text{scale}_i} \right)^2}
\]

(17.2.9)

and accept the step if \(\text{err} \leq 1\), otherwise reject it.

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

\[
h_0 = h_1 \left| \frac{\text{err}_0}{\text{err}_1} \right|^{1/5}
\]

(17.2.10)

Let \(\text{err}_0\) denote the desired error, which is 1 in an efficient integration. Then
equation (17.2.10) is used in two ways: If $err_1$ is larger than 1 in magnitude, the equation tells how much to decrease the stepsize when we retry the present (failed) step. If $err_1$ 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, $atol = 0, rtol = \epsilon$, where $\epsilon$ 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 $atol = rtol = \epsilon$. 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

$$scale = \epsilon h \times dydx[i]$$  \hspace{1cm} (17.2.11)

This enforces fractional accuracy $\epsilon$ 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 $err_0 = 1$ and the subscripts $1 \rightarrow n$ and $0 \rightarrow n + 1$) is thus replaced by

$$h_{n+1} = Sh_n \left( \frac{1}{err_n} \right)^{1/5}$$  \hspace{1cm} (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 $\text{err}$ 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

$$h_{n+1} = Sh_n \text{err}_n^\alpha \text{err}_{n-1}^\beta$$  \hspace{1cm} (17.2.13)

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

$$\beta \approx 0.4/k, \quad \alpha \approx 0.7/k = 1/k - 0.75\beta$$  \hspace{1cm} (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...