CHAPTER 8

ORDINARY DIFFERENTIAL EQUATIONS

PERHAPS the most common use of computers in physics is for the solution of differential equations. In this chapter we look at techniques for solving ordinary differential equations, such as the equations of motion of rigid bodies or the equations governing the behavior of electrical circuits. In the following chapter we look at techniques for partial differential equations, such as the wave equation and the diffusion equation.

8.1 FIRST-ORDER DIFFERENTIAL EQUATIONS WITH ONE VARIABLE

We begin our study of differential equations by looking at ordinary differential equations, meaning those for which there is only one independent variable, such as time, and all dependent variables are functions solely of that one independent variable. The simplest type of ordinary differential equation is a first-order equation with one dependent variable, such as

$$\frac{dx}{dt} = \frac{2x}{t}. \tag{8.1}$$

This equation, however, can be solved exactly by hand by separating the variables. There's no need to use a computer in this case. But suppose instead that you had

$$\frac{dx}{dt} = \frac{2x}{t} + \frac{3x^2}{t^3}. \tag{8.2}$$

Now the equation is no longer separable and moreover it's nonlinear, meaning that powers or other nonlinear functions of the dependent variable x appear in the equation. Nonlinear equations can rarely be solved analytically, but they can be solved numerically. Computers don't care whether a differential equation is linear or nonlinear—the techniques used to solve it are the same either way.

The general form of a first-order one-variable ordinary differential equation is

$$\frac{dx}{dt} = f(x,t), \tag{8.3}$$

where $f(x,t)$ is some function we specify. In Eq. (8.2) we had $f(x,t) = 2x/t + 3x^2/t^3$. The independent variable is denoted t in this example, because in physics the independent variable is often time. But of course there are other possibilities. We could just as well have written our equation as

$$\frac{dy}{dx} = f(x,y). \tag{8.4}$$

In this chapter we will stick with t for the independent variable, but it's worth bearing in mind that there are plenty of examples where the independent variable is not time.

To calculate a full solution to Eq. (8.3) we also require an initial condition or boundary condition—we have to specify the value of x at one particular value of t, for instance at $t = 0$. In all the problems we'll tackle in this chapter we will assume that we're given both the equation and its initial or boundary conditions.

8.1.1 EULER'S METHOD

Suppose we are given an equation of the form (8.3) and an initial condition that fixes the value of x for some t. Then we can write the value of x a short interval h later using a Taylor expansion thus:

$$\begin{aligned} x(t+h) &= x(t) + h\frac{dx}{dt} + \tfrac{1}{2}h^2\frac{d^2x}{dt^2} + \ldots \\ &= x(t) + hf(x,t) + O(h^2), \end{aligned} \tag{8.5}$$

where we have used Eq. (8.3) and $O(h^2)$ is a shorthand for terms that go as h^2 or higher. If h is small then h^2 is very small, so we can neglect the terms in h^2 and get

$$x(t+h) = x(t) + hf(x,t). \tag{8.6}$$

If we know the value of x at time t we can use this equation to calculate the value a short time later. Then we can just repeat the exercise to calculate x another interval h after that, and so forth, and thereby calculate x at a succession of evenly spaced points for as long as we want. We don't get $x(t)$ for all values of t from this calculation, only at a finite set of points, but if h is small enough

we can get a pretty good picture of what the solution to the equation looks like. As we saw in Section 3.1, we can make a convincing plot of a curve by approximating it with a set of closely spaced points.

Thus, for instance, we might be given a differential equation for x and an initial condition at $t = a$ and asked to make a graph of $x(t)$ for values of t from a to b. To do this, we would divide the interval from a to b into steps of size h and use (8.6) repeatedly to calculate $x(t)$, then plot the results. This method for solving differential equations is called *Euler's method*, after its inventor, Leonhard Euler.

EXAMPLE 8.1: EULER'S METHOD

Let us use Euler's method to solve the differential equation

$$\frac{dx}{dt} = -x^3 + \sin t \tag{8.7}$$

with the initial condition $x = 0$ at $t = 0$. Here is a program to do the calculation from $t = 0$ to $t = 10$ in 1000 steps and plot the result:

File: euler.py

```
from math import sin
from numpy import arange
from pylab import plot,xlabel,ylabel,show

def f(x,t):
    return -x**3 + sin(t)

a = 0.0           # Start of the interval
b = 10.0          # End of the interval
N = 1000          # Number of steps
h = (b-a)/N       # Size of a single step
x = 0.0           # Initial condition

tpoints = arange(a,b,h)
xpoints = []
for t in tpoints:
    xpoints.append(x)
    x += h*f(x,t)

plot(tpoints,xpoints)
xlabel("t")
ylabel("x(t)")
show()
```

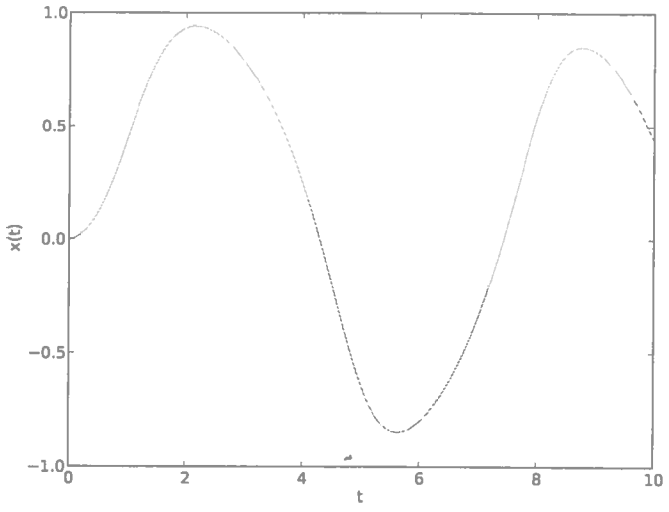


Figure 8.1: Numerical solution of an ordinary differential equation. A solution to Eq. (8.7) from $x = 0$ to $x = 10$, calculated using Euler's method.

If we run this program it produces the picture shown in Fig. 8.1, which, as we'll see, turns out to be a pretty good approximation to the shape of the true solution to the equation. In this case, Euler's method does a good job.

In general, Euler's method is not bad. It gives reasonable answers in many cases. In practice, however, we never actually use Euler's method. Why not? Because there is a better method that's very little extra work to program, much more accurate, and runs just as fast and often faster. This is the so-called Runge–Kutta method, which we'll look at in a moment. First, however, let's look a little more closely at Euler's method, to understand why it's not ideal.[1]

Euler's method only gives approximate solutions. The approximation arises because we neglected the h^2 term (and all higher-order terms) in Eq. (8.5). The

[1]It's not completely correct to say that we never use Euler's method. We never use it for solving *ordinary* differential equations, but in Section 9.3 we will see that Euler's method is useful for solving partial differential equations. It's true in that case also that Euler's method is not very accurate, but there are other bigger sources of inaccuracy when solving partial differential equations which mean that the inaccuracy of Euler's method is moot, and in such situations its simplicity makes it the method of choice.

size of the h^2 term is $\frac{1}{2}h^2\,\mathrm{d}^2x/\mathrm{d}t^2$, which tells us the error introduced on a single step of the method, to leading order, and this error gets smaller as h gets smaller so we can make the step more accurate by making h small.

But we don't just take a single step when we use Euler's method. We take many. If we want to calculate a solution from $t = a$ to $t = b$ using steps of size h, then the total number of steps we need is $N = (b-a)/h$. Let us denote the values of t at which the steps fall by $t_k = a + kh$ and the corresponding values of x (which we calculate as we go along) by x_k. Then the total, cumulative error incurred as we solve our differential equation all the way from a to b is given by the sum of the individual errors on each step thus:

$$\sum_{k=0}^{N-1} \tfrac{1}{2}h^2 \left(\frac{\mathrm{d}^2x}{\mathrm{d}t^2}\right)_{\substack{x=x_k\\t=t_k}} = \tfrac{1}{2}h \sum_{k=0}^{N-1} h \left(\frac{\mathrm{d}f}{\mathrm{d}t}\right)_{\substack{x=x_k\\t=t_k}} \simeq \tfrac{1}{2}h \int_a^b \frac{\mathrm{d}f}{\mathrm{d}t}\,\mathrm{d}t$$
$$= \tfrac{1}{2}h\big[f(x(b),b) - f(x(a),a)\big], \tag{8.8}$$

where we have approximated the sum by an integral, which is a good approximation if h is small.

Notice that the final expression for the total error is linear in h, even though the individual errors are of order h^2, meaning that the total error goes down by a factor of two when we make h half as large. In principle this allows us to make the error as small as we like, although when we make h smaller we also increase the number of steps $N = (b-a)/h$ and hence the calculation will take proportionately longer—a calculation that's twice as accurate will take twice as long.

Perhaps this doesn't sound too bad. If that's the way it had to be, we could live with it. But it doesn't have to be that way. The Runge–Kutta method does much better.

8.1.2 THE RUNGE–KUTTA METHOD

You might think that the way to improve on Euler's method would be to use the Taylor expansion of Eq. (8.5) again, but keep terms to higher order. For instance, in addition to the order h term we could keep the order h^2 term, which is equal to

$$\tfrac{1}{2}h^2\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = \tfrac{1}{2}h^2\frac{\mathrm{d}f}{\mathrm{d}t}. \tag{8.9}$$

This would give us a more accurate expression for $x(t+h)$, and in some cases this approach might work, but in a lot of cases it would not. It requires us to know the derivative $\mathrm{d}f/\mathrm{d}t$, which we can calculate only if we have an explicit

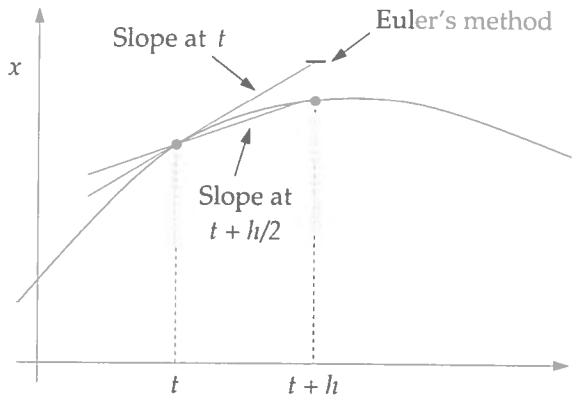


Figure 8.2: Euler's method and the second-order Runge–Kutta method. Euler's method is equivalent to taking the slope dx/dt at time t and extrapolating it into the future to time $t + h$. A better approximation is to perform the extrapolation using the slope at time $t + \frac{1}{2}h$.

expression for f. Often we have no such expression because, for instance, the function f is calculated as the output of another computer program or function and therefore doesn't have a mathematical formula. And even if f is known explicitly, a method that requires us to calculate its derivative is less convenient than the Runge–Kutta method, which gives higher accuracy and doesn't require any derivatives.

The Runge–Kutta method is really a set of methods—there are many of them of different orders, which give results of varying degrees of accuracy. In fact technically Euler's method is a Runge–Kutta method. It is the first-order Runge–Kutta method. Let us look at the next method in the series, the second-order method, also sometimes called the *midpoint method*, for reasons that will shortly become clear.

Euler's method can be represented in graphical fashion as shown in Fig. 8.2. The curve represents the true form of $x(t)$, which we are trying to calculate. The differential equation $dx/dt = f(x,t)$ tells us that the slope of the solution is equal to the function $f(x,t)$, so that, given the value of x at time t we can calculate the slope at that point, as shown in the figure. Then we extrapolate that slope to time $t + h$ and it gives us an estimate of the value of $x(t + h)$, which is labeled "Euler's method" in the figure. If the curve of $x(t)$ were in fact a straight line between t and $t + h$, then this method would give a perfect estimate of $x(t + h)$. But if it's curved, as in the picture, then the estimate is only

approximate, and the error introduced is the difference between the estimate and the true value of $x(t+h)$.

Now suppose we do the same calculation but instead use the slope at the midpoint $t + \frac{1}{2}h$ to do our extrapolation, as shown in the figure. If we extrapolate using this slope we get a different estimate of $x(t+h)$ which is usually significantly better than Euler's method. This is the basis for the second-order Runge–Kutta method.

In mathematical terms the method involves performing a Taylor expansion around $t + \frac{1}{2}h$ to get the value of $x(t+h)$ thus:

$$x(t+h) = x(t+\tfrac{1}{2}h) + \tfrac{1}{2}h\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)_{t+\frac{1}{2}h} + \tfrac{1}{8}h^2\left(\frac{\mathrm{d}^2x}{\mathrm{d}t^2}\right)_{t+\frac{1}{2}h} + \mathrm{O}(h^3). \tag{8.10}$$

Similarly we can derive an expression for $x(t)$:

$$x(t) = x(t+\tfrac{1}{2}h) - \tfrac{1}{2}h\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)_{t+\frac{1}{2}h} + \tfrac{1}{8}h^2\left(\frac{\mathrm{d}^2x}{\mathrm{d}t^2}\right)_{t+\frac{1}{2}h} + \mathrm{O}(h^3). \tag{8.11}$$

Subtracting the second expression from the first and rearranging then gives

$$\begin{aligned} x(t+h) &= x(t) + h\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)_{t+\frac{1}{2}h} + \mathrm{O}(h^3) \\ &= x(t) + hf\big(x(t+\tfrac{1}{2}h), t+\tfrac{1}{2}h\big) + \mathrm{O}(h^3). \end{aligned} \tag{8.12}$$

Notice that the term in h^2 has completely disappeared. The error term is now $\mathrm{O}(h^3)$, so our approximation is a whole factor of h more accurate than before. If h is small this could make a big difference to the accuracy of the calculation.

Though it looks promising, there is a problem with this approach: Eq. (8.12) requires a knowledge of $x(t+\frac{1}{2}h)$, which we don't have. We only know the value at $x(t)$. We get around this by approximating $x(t+\frac{1}{2}h)$ using Euler's method $x(t+\frac{1}{2}h) = x(t) + \frac{1}{2}hf(x,t)$ and then substituting into the equation above. The complete calculation for a single step can be written like this:

$$k_1 = hf(x,t), \tag{8.13a}$$

$$k_2 = hf(x + \tfrac{1}{2}k_1, t + \tfrac{1}{2}h), \tag{8.13b}$$

$$x(t+h) = x(t) + k_2. \tag{8.13c}$$

Notice how the first equation gives us a value for k_1 which, when inserted into the second equation, gives us our estimate of $x(t+\frac{1}{2}h)$. Then the resulting value of k_2, inserted into the third equation, gives us the final Runge–Kutta estimate for $x(t+h)$.

These are the equations for the second-order Runge–Kutta method. As with the methods for performing integrals that we studied a Chapter 5, a "second-order" method, in this context, is a method *accurate* to order h^2, meaning that the *error* is of order h^3. Euler's method, by contrast, is a first-order method with an error of order h^2. Note that these designations refer to just a single step of each method. As discussed in Section 8.1.1, real calculations involve doing many steps one after another, with errors that accumulate, so that the accuracy of the final calculation is poorer (typically one order in h poorer) than the individual steps.

The second-order Runge–Kutta method is only a little more complicated to program than Euler's method, but gives much more accurate results for any given value of h. Or, alternatively, we could make h bigger—and so take fewer steps—while still getting the same level of accuracy as Euler's method, thus creating a program that achieves the same result as Euler's method but runs faster.

We are not entirely done with our derivation yet, however. Since we don't have an exact value of $x(t+\frac{1}{2}h)$ and had to approximate it using Euler's method, there is an extra source of error in Eq. (8.12), coming from this second approximation, in addition to the $O(h^3)$ error we have already acknowledged. How do we know that this second error isn't larger than $O(h^3)$ and doesn't make the accuracy of our calculation worse?

We can show that in fact this is not a problem by expanding the quantity $f(x+\frac{1}{2}k_1, t+\frac{1}{2}h)$ in Eq. (8.13b) in its first argument only, around $x(t+\frac{1}{2}h)$:

$$f(x(t)+\tfrac{1}{2}k_1, t+\tfrac{1}{2}h) = f(x(t+\tfrac{1}{2}h), t+\tfrac{1}{2}h) + [x(t)+\tfrac{1}{2}k_1 - x(t+\tfrac{1}{2}h)]\left(\frac{\partial f}{\partial x}\right)_{x(t+h/2),t+h/2} + O([x(t)+\tfrac{1}{2}k_1 - x(t+\tfrac{1}{2}h)]^2). \tag{8.14}$$

But from Eq. (8.5) we have

$$x(t+\tfrac{1}{2}h) = x(t) + \tfrac{1}{2}hf(x,t) + O(h^2) = x(t) + \tfrac{1}{2}k_1 + O(h^2), \tag{8.15}$$

so $x(t)+\frac{1}{2}k_1 - x(t+\frac{1}{2}h) = O(h^2)$ and

$$f(x(t)+\tfrac{1}{2}k_1, t+\tfrac{1}{2}h) = f(x(t+\tfrac{1}{2}h), t+\tfrac{1}{2}h) + O(h^2). \tag{8.16}$$

This means that Eq. (8.13b) gives $k_2 = hf(x(t+\frac{1}{2}h), t+\frac{1}{2}h) + O(h^3)$, and hence there's no problem—our Euler's method approximation for $x(t+\frac{1}{2}h)$ does introduce an additional error into the calculation, but the error goes like h^3 and hence our second-order Runge–Kutta method is still accurate to $O(h^3)$ overall.

EXAMPLE 8.2: THE SECOND-ORDER RUNGE–KUTTA METHOD

Let us use the second-order Runge–Kutta method to solve the same differential equation as we solved in Example 8.1. The program is a minor modification of our program for Euler's method:

File: rk2.py

```
from math import sin
from numpy import arange
from pylab import plot,xlabel,ylabel,show

def f(x,t):
    return -x**3 + sin(t)

a = 0.0
b = 10.0
N = 10
h = (b-a)/N

tpoints = arange(a,b,h)
xpoints = []

x = 0.0
for t in tpoints:
    xpoints.append(x)
    k1 = h*f(x,t)
    k2 = h*f(x+0.5*k1,t+0.5*h)
    x += k2

plot(tpoints,xpoints)
xlabel("t")
ylabel("x(t)")
show()
```

If we run this program repeatedly with different values for the number of points N, starting with 10, then 20, then 50, then 100, and plot the results, we get the plot shown in Fig. 8.3. The figure reveals that the solution with 10 points is quite poor, as is the solution with 20. But the solutions for 50 and 100 points look very similar, indicating that the method has converged to a result close to the true solution, and indeed a comparison with Fig. 8.1 shows good agreement with our Euler's method solution, which used 1000 points.

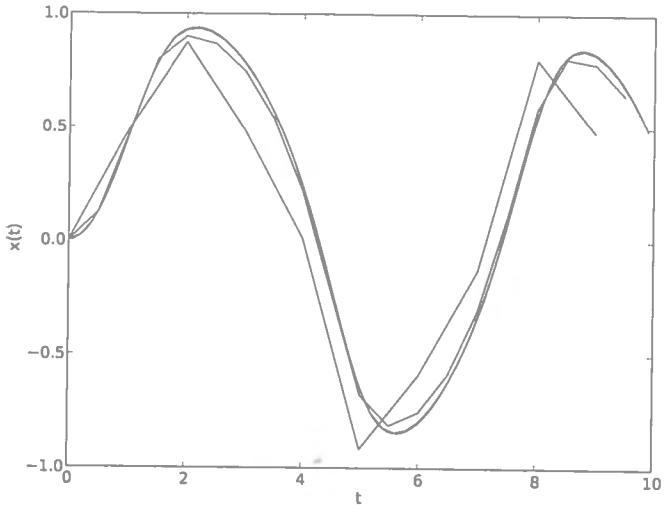


Figure 8.3: Solutions calculated with the second-order Runge–Kutta method. Solutions to Eq. (8.7) calculated using the second-order Runge–Kutta method with $N = 10$, 20, 50, and 100 steps.

8.1.3 THE FOURTH-ORDER RUNGE–KUTTA METHOD

We can take this approach further. By performing Taylor expansions around various points and then taking the right linear combinations of them, we can arrange for terms in h^3, h^4, and so on to cancel out of our expressions, and so get more and more accurate rules for solving differential equations. The downside is that the equations become more complicated as we go to higher order. Many people feel, however, that the sweet spot is the fourth-order rule, which offers a good balance of high accuracy and equations that are still relatively simple to program. The equations look like this:

$$k_1 = hf(x,t), \tag{8.17a}$$
$$k_2 = hf(x + \tfrac{1}{2}k_1, t + \tfrac{1}{2}h), \tag{8.17b}$$
$$k_3 = hf(x + \tfrac{1}{2}k_2, t + \tfrac{1}{2}h), \tag{8.17c}$$
$$k_4 = hf(x + k_3, t + h), \tag{8.17d}$$
$$x(t+h) = x(t) + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{8.17e}$$

This is the *fourth-order Runge–Kutta method*, and it is by far the most common method for the numerical solution of ordinary differential equations. It is accurate to terms of order h^4 and carries an error of order h^5. Although its derivation is quite complicated (we'll not go over the algebra—it's very tedious), the final equations are relatively simple. There are just five of them, and yet the result is a method that is three orders of h more accurate than Euler's method for steps of the same size. In practice this can make the fourth-order method as much as a million times more accurate than Euler's method. Indeed the fourth-order method is significantly better even than the second-order method of Section 8.1.2. Alternatively, we can use the fourth-order Runge–Kutta method with much larger h and many fewer steps and still get accuracy just as good as Euler's method, giving a method that runs far faster yet gives comparable results.

For many professional physicists, the fourth-order Runge–Kutta method is the first method they turn to when they want to solve an ordinary differential equation on the computer. It is simple to program and gives excellent results. It is the workhorse of differential equation solvers and one of the best known computer algorithms of any kind anywhere.

EXAMPLE 8.3: THE FOURTH-ORDER RUNGE–KUTTA METHOD

Let us once more solve the differential equation from Eq. (8.7), this time using the fourth-order Runge–Kutta method. The program is again only a minor modification of our previous ones:

File: `rk4.py`

```
from math import sin
from numpy import arange
from pylab import plot,xlabel,ylabel,show

def f(x,t):
    return -x**3 + sin(t)

a = 0.0
b = 10.0
N = 10
h = (b-a)/N

tpoints = arange(a,b,h)
xpoints = []
x = 0.0
```

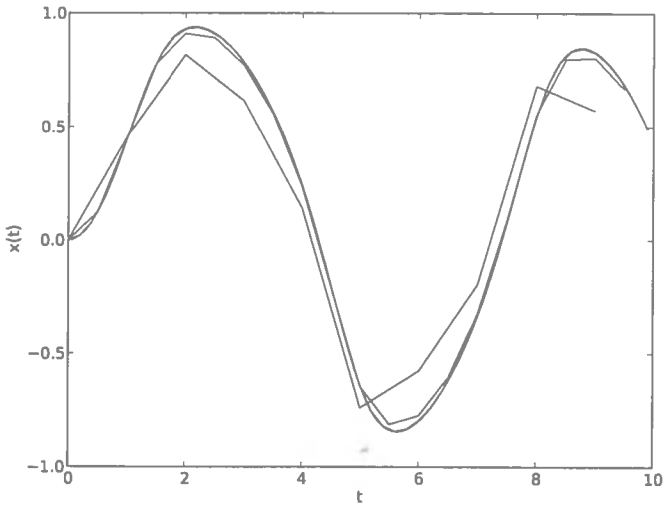


Figure 8.4: Solutions calculated with the fourth-order Runge–Kutta method. Solutions to Eq. (8.7) calculated using the fourth-order Runge–Kutta method with $N = 10$, 20, 50, and 100 steps.

```
for t in tpoints:
    xpoints.append(x)
    k1 = h*f(x,t)
    k2 = h*f(x+0.5*k1,t+0.5*h)
    k3 = h*f(x+0.5*k2,t+0.5*h)
    k4 = h*f(x+k3,t+h)
    x += (k1+2*k2+2*k3+k4)/6

plot(tpoints,xpoints)
xlabel("t")
ylabel("x(t)")
show()
```

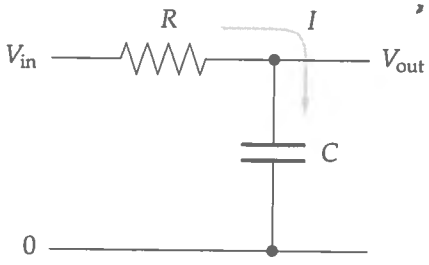
Again we run the program repeatedly with $N = 10$, 20, 50, and 100. Figure 8.4 shows the results. Now we see that, remarkably, even the solution with 20 points is close to the final converged solution for the equation. With only 20 points we get quite a jagged curve—20 points is not enough to make the curve appear smooth in the plot—but the points nonetheless lie close to the final solution of the equation. With only 20 points the fourth-order method has

calculated a solution almost as accurate as Euler's method with a thousand points.

One minor downside of the fourth-order Runge–Kutta method, and indeed of all Runge–Kutta methods, is that if you get the equations wrong, it may not be obvious in the solution they produce. If, for example, you miss one of the factors of $\frac{1}{2}$ or 2, or have a minus sign when you should have a plus, then the method will probably still produce a solution that looks approximately right. The solution will be *much* less accurate than the correct fourth-order method—if you don't use the equations exactly as in Eq. (8.17) you will probably only get a solution about as accurate as Euler's method, which, as we have seen, is much worse. This means that you must be careful when writing programs that use the Runge–Kutta method. Check your code in detail to make sure all the equations are exactly correct. If you make a mistake you may never realize it because your program will appear to give reasonable answers, but in fact there will be large errors. This contrasts with most other types of calculation in computational physics, where if you make even a small error in the program it is likely to produce ridiculous results that are so obviously wrong that the error is relatively easy to spot.

Exercise 8.1: A low-pass filter

Here is a simple electronic circuit with one resistor and one capacitor:



This circuit acts as a low-pass filter: you send a signal in on the left and it comes out filtered on the right.

Using Ohm's law and the capacitor law and assuming that the output load has very high impedance, so that a negligible amount of current flows through it, we can write down the equations governing this circuit as follows. Let I be the current that flows through R and into the capacitor, and let Q be the charge on the capacitor. Then:

$$IR = V_{\text{in}} - V_{\text{out}}, \qquad Q = CV_{\text{out}}, \qquad I = \frac{dQ}{dt}.$$

Substituting the second equation into the third, then substituting the result into the first equation, we find that $V_{\text{in}} - V_{\text{out}} = RC\,(\mathrm{d}V_{\text{out}}/\mathrm{d}t)$, or equivalently

$$\frac{\mathrm{d}V_{\text{out}}}{\mathrm{d}t} = \frac{1}{RC}(V_{\text{in}} - V_{\text{out}}).$$

a) Write a program (or modify a previous one) to solve this equation for $V_{\text{out}}(t)$ using the fourth-order Runge–Kutta method when the input signal is a square-wave with frequency 1 and amplitude 1:

$$V_{\text{in}}(t) = \begin{cases} 1 & \text{if } \lfloor 2t \rfloor \text{ is even,} \\ -1 & \text{if } \lfloor 2t \rfloor \text{ is odd,} \end{cases} \tag{8.18}$$

where $\lfloor x \rfloor$ means x rounded down to the next lowest integer. Use the program to make plots of the output of the filter circuit from $t = 0$ to $t = 10$ when $RC = 0.01$, 0.1, and 1, with initial condition $V_{\text{out}}(0) = 0$. You will have to make a decision about what value of h to use in your calculation. Small values give more accurate results, but the program will take longer to run. Try a variety of different values and choose one for your final calculations that seems sensible to you.

b) Based on the graphs produced by your program, describe what you see and explain what the circuit is doing.

A program similar to the one you wrote is running inside most stereos and music players, to create the effect of the "bass" control. In the old days, the bass control on a stereo would have been connected to a real electronic low-pass filter in the amplifier circuitry, but these days there is just a computer processor that simulates the behavior of the filter in a manner similar to your program.

8.1.4 SOLUTIONS OVER INFINITE RANGES

We have seen how to find the solution of a differential equation starting from a given initial condition and going a finite distance in t, but in some cases we want to find the solution all the way out to $t = \infty$. In that case we cannot use the method above directly, since we'd need an infinite number of steps to reach $t = \infty$, but we can play a trick similar to the one we played when we were doing integrals in Section 5.8, and change variables. We define

$$u = \frac{t}{1+t} \qquad \text{or equivalently} \qquad t = \frac{u}{1-u}, \tag{8.19}$$

so that as $t \to \infty$ we have $u \to 1$. Then, using the chain rule, we can rewrite our differential equation $\mathrm{d}x/\mathrm{d}t = f(x,t)$ as

$$\frac{\mathrm{d}x}{\mathrm{d}u}\frac{\mathrm{d}u}{\mathrm{d}t} = f(x,t), \tag{8.20}$$

or

$$\frac{dx}{du} = \frac{dt}{du} f\left(x, \frac{u}{1-u}\right). \tag{8.21}$$

But

$$\frac{dt}{du} = \frac{1}{(1-u)^2}, \tag{8.22}$$

so

$$\frac{dx}{du} = (1-u)^{-2} f\left(x, \frac{u}{1-u}\right). \tag{8.23}$$

If we define a new function $g(x,u)$ by

$$g(x,u) = (1-u)^{-2} f\left(x, \frac{u}{1-u}\right) \tag{8.24}$$

then we have

$$\frac{dx}{du} = g(x,u), \tag{8.25}$$

which is a normal first-order differential equation again, as before. Solving this equation for values of u up to 1 is equivalent to solving the original equation for values of t up to infinity. The solution will give us $x(u)$ and we then map u back onto t using Eq. (8.19) to get $x(t)$.

EXAMPLE 8.4: SOLUTION OVER AN INFINITE RANGE

Suppose we want to solve the equation

$$\frac{dx}{dt} = \frac{1}{x^2+t^2},$$

from $t = 0$ to $t = \infty$ with $x = 1$ at $t = 0$. What would be the equivalent differential equation in x and u that we would solve?

Applying Eq. (8.24), we have

$$g(x,u) = (1-u)^{-2} \frac{1}{x^2 + u^2/(1-u)^2} = \frac{1}{x^2(1-u)^2 + u^2}. \tag{8.26}$$

So we would solve the equation

$$\frac{dx}{du} = \frac{1}{x^2(1-u)^2 + u^2} \tag{8.27}$$

from $u = 0$ to $u = 1$, with an initial condition $x = 1$ at $u = 0$. We can calculate the solution with only a small modification of the program we used in Example 8.3:

File: odeinf.py

```
from numpy import arange
from pylab import plot,xlabel,ylabel,xlim,show

def g(x,u):
    return 1/(x**2*(1-u)**2+u**2)

a = 0.0
b = 1.0
N = 100
h = (b-a)/N

upoints = arange(a,b,h)
tpoints = []
xpoints = []

x = 1.0
for u in upoints:
    tpoints.append(u/(1-u))
    xpoints.append(x)
    k1 = h*g(x,u)
    k2 = h*g(x+0.5*k1,u+0.5*h)
    k3 = h*g(x+0.5*k2,u+0.5*h)
    k4 = h*g(x+k3,u+h)
    x += (k1+2*k2+2*k3+k4)/6

plot(tpoints,xpoints)
xlim(0,80)
xlabel("t")
ylabel("x(t)")
show()
```

Note how we made a list `tpoints` of the value of t at each step of the Runge–Kutta method, as we went along. Although we don't need these values for the solution itself, we use them at the end to make a plot of the final solution in terms of t rather than u. The resulting plot is shown in Fig. 8.5. (It only goes up to $t = 80$. Obviously it cannot go all the way out to infinity—one cannot draw an infinitely wide plot—but the solution itself does go out to infinity.)

As with the integrals of Section 5.8, there are other changes of variables that can be used in calculations like this, including transformations based on trigonometric functions, hyperbolic functions, and others. The transformation of Eq. (8.19) is often a good first guess—it works well in many cases—but other

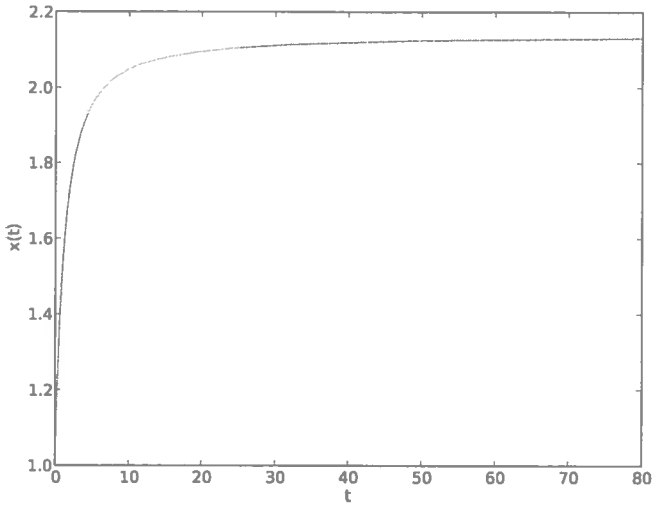


Figure 8.5: Solution of a differential equation to $t = \infty$. The solution of the differential equation in Eq. (8.4), calculated by solving all the way out to $t = \infty$ as described in the text, although only the part of the solution up to $t = 80$ is shown here.

choices can be appropriate too. A shrewd choice of variables can make the algebra easier, simplify the form of the function $g(x, u)$, or give the solution more accuracy in a region of particular interest.

8.2 DIFFERENTIAL EQUATIONS WITH MORE THAN ONE VARIABLE

So far we have considered ordinary differential equations with only one dependent variable x, but in many physics problems we have more than one variable. That is, we have *simultaneous differential equations*, where the derivative of each variable can depend on any or all of the variables, as well as the independent variable t. For example:

$$\frac{dx}{dt} = xy - x, \qquad \frac{dy}{dt} = y - xy + \sin^2 \omega t. \tag{8.28}$$

Note that there is still only one *independent* variable t. These are still ordinary differential equations, not partial differential equations.

A general form for two first-order simultaneous differential equations is[2]

$$\frac{dx}{dt} = f_x(x, y, t), \qquad \frac{dy}{dt} = f_y(x, y, t), \tag{8.29}$$

where f_x and f_y are general, possibly nonlinear, functions of x, y, and t. For an arbitrary number of variables the equations can be written using vector notation as

$$\frac{d\mathbf{r}}{dt} = \mathbf{f}(\mathbf{r}, t), \tag{8.30}$$

where $\mathbf{r} = (x, y, \ldots)$ and $\mathbf{f}$ is a vector of functions $\mathbf{f}(\mathbf{r}, t) = (f_x(\mathbf{r}, t), f_y(\mathbf{r}, t), \ldots)$.

Although simultaneous differential equations are often a lot harder to solve analytically than single equations, when solving computationally they are actually not much more difficult than the one-variable case. For instance, we can Taylor expand the vector $\mathbf{r}$ thus:

$$\mathbf{r}(t+h) = \mathbf{r}(t) + h\frac{d\mathbf{r}}{dt} + O(h^2) = \mathbf{r}(t) + h\mathbf{f}(\mathbf{r}, t) + O(h^2). \tag{8.31}$$

Dropping the terms of order h^2 and higher we get Euler's method for the multi-variable case:

$$\mathbf{r}(t+h) = \mathbf{r}(t) + h\mathbf{f}(\mathbf{r}, t). \tag{8.32}$$

The Taylor expansions used to derive the Runge–Kutta rules also generalize straightforwardly to the multi-variable case, and in particular the multi-variable version of the fourth-order Runge–Kutta method is an obvious vector generalization of the one-variable version:

$$\mathbf{k}_1 = h\mathbf{f}(\mathbf{r}, t), \tag{8.33a}$$

$$\mathbf{k}_2 = h\mathbf{f}(\mathbf{r} + \tfrac{1}{2}\mathbf{k}_1, t + \tfrac{1}{2}h), \tag{8.33b}$$

$$\mathbf{k}_3 = h\mathbf{f}(\mathbf{r} + \tfrac{1}{2}\mathbf{k}_2, t + \tfrac{1}{2}h), \tag{8.33c}$$

$$\mathbf{k}_4 = h\mathbf{f}(\mathbf{r} + \mathbf{k}_3, t + h), \tag{8.33d}$$

$$\mathbf{r}(t+h) = \mathbf{r}(t) + \tfrac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4). \tag{8.33e}$$

These equations can be conveniently translated into Python using arrays to represent the vectors. Since Python allows us to do arithmetic with vectors di-

[2] Although it covers most cases of interest, this is not the most general possible form. In principle, dx/dt could also depend on dy/dt, possibly in nonlinear fashion. We'll assume that the equations have already been separated in the derivatives to remove such dependencies. It's worth noting, however, that such separation is not always possible—for instance, when the equations involve transcendental functions. In such cases, other methods, such as the relaxation methods discussed in Section 8.6.2, may be needed to find a solution.

rectly, and allows vectors to be both the arguments and the results of functions, the code is only slightly more complicated than for the one-variable case.

EXAMPLE 8.5: SIMULTANEOUS ORDINARY DIFFERENTIAL EQUATIONS

Let us calculate a solution to the equations given in Eq. (8.28) from $t = 0$ to $t = 10$, for the case $\omega = 1$ with initial condition $x = y = 1$ at $t = 0$. Here is a suitable program, again based on a slight modification of our earlier programs.

File: odesim.py

```
from math import sin
from numpy import array,arange
from pylab import plot,xlabel,show

def f(r,t):
    x = r[0]
    y = r[1]
    fx = x*y - x
    fy = y - x*y + sin(t)**2
    return array([fx,fy],float)

a = 0.0
b = 10.0
N = 1000
h = (b-a)/N

tpoints = arange(a,b,h)
xpoints = []
ypoints = []

r = array([1.0,1.0],float)
for t in tpoints:
    xpoints.append(r[0])
    ypoints.append(r[1])
    k1 = h*f(r,t)
    k2 = h*f(r+0.5*k1,t+0.5*h)
    k3 = h*f(r+0.5*k2,t+0.5*h)
    k4 = h*f(r+k3,t+h)
    r += (k1+2*k2+2*k3+k4)/6
plot(tpoints,xpoints)
plot(tpoints,ypoints)
xlabel("t")
show()
```

Note in particular the definition of the function `f(r,t)`, which takes a vector argument `r`, breaks it apart into its components x and y, forms the values of f_x and f_y from them, then puts those values together into an array and returns that array as the final output of the function. In fact, the construction of this function is really the only complicated part of the program; in other respects the program is almost identical to the program we used for the one-variable case in Example 8.3. The lines representing the Runge–Kutta method itself are unchanged except for the replacement of the scalar variable `x` by the new vector variable `r`.

This program will form the basis for the solution of many other problems in this chapter.

Exercise 8.2: The Lotka–Volterra equations

The Lotka–Volterra equations are a mathematical model of predator–prey interactions between biological species. Let two variables x and y be proportional to the size of the populations of two species, traditionally called "rabbits" (the prey) and "foxes" (the predators). You could think of x and y as being the population in thousands, say, so that $x = 2$ means there are 2000 rabbits. Strictly the only allowed values of x and y would then be multiples of 0.001, since you can only have whole numbers of rabbits or foxes. But 0.001 is a pretty close spacing of values, so it's a decent approximation to treat x and y as continuous real numbers so long as neither gets very close to zero.

In the Lotka–Volterra model the rabbits reproduce at a rate proportional to their population, but are eaten by the foxes at a rate proportional to both their own population and the population of foxes:

$$\frac{dx}{dt} = \alpha x - \beta xy,$$

where α and β are constants. At the same time the foxes reproduce at a rate proportional to the rate at which they eat rabbits—because they need food to grow and reproduce—but also die of old age at a rate proportional to their own population:

$$\frac{dy}{dt} = \gamma xy - \delta y,$$

where γ and δ are also constants.

a) Write a program to solve these equations using the fourth-order Runge–Kutta method for the case $\alpha = 1$, $\beta = \gamma = 0.5$, and $\delta = 2$, starting from the initial condition $x = y = 2$. Have the program make a graph showing both x and y as a function of time on the same axes from $t = 0$ to $t = 30$. (Hint: Notice that the differential equations in this case do not depend explicitly on time t—in

vector notation, the right-hand side of each equation is a function $f(\mathbf{r})$ with no t dependence. You may nonetheless find it convenient to define a Python function `f(r,t)` including the time variable, so that your program takes the same form as programs given earlier in this chapter. You don't have to do it that way, but it can avoid some confusion. Several of the following exercises have a similar lack of explicit time-dependence.)

b) Describe in words what is going on in the system, in terms of rabbits and foxes.

Exercise 8.3: The Lorenz equations

One of the most celebrated sets of differential equations in physics is the Lorenz equations:

$$\frac{dx}{dt} = \sigma(y-x), \qquad \frac{dy}{dt} = rx - y - xz, \qquad \frac{dz}{dt} = xy - bz,$$

where σ, r, and b are constants. (The names σ, r, and b are odd, but traditional—they are always used in these equations for historical reasons.)

These equations were first studied by Edward Lorenz in 1963, who derived them from a simplified model of weather patterns. The reason for their fame is that they were one of the first incontrovertible examples of *deterministic chaos*, the occurrence of apparently random motion even though there is no randomness built into the equations. We encountered a different example of chaos in the logistic map of Exercise 3.6.

a) Write a program to solve the Lorenz equations for the case $\sigma = 10$, $r = 28$, and $b = \frac{8}{3}$ in the range from $t = 0$ to $t = 50$ with initial conditions $(x,y,z) = (0,1,0)$. Have your program make a plot of y as a function of time. Note the unpredictable nature of the motion. (Hint: If you base your program on previous ones, be careful. This problem has parameters r and b with the same names as variables in previous programs—make sure to give your variables new names, or use different names for the parameters, to avoid introducing errors into your code.)

b) Modify your program to produce a plot of z against x. You should see a picture of the famous "strange attractor" of the Lorenz equations, a lop-sided butterfly-shaped plot that never repeats itself.

8.3 SECOND-ORDER DIFFERENTIAL EQUATIONS

So far we have looked at first-order differential equations, but first-order equations are in fact quite rare in physics. Many, perhaps most, of the equations encountered in physics are second-order or higher. Luckily, now that we know how to solve first-order equations, solving second-order ones is pretty easy, because of the following trick.

Consider first the simple case where there is only one dependent variable x. The general form for a second-order differential equation with one dependent variable is

$$\frac{d^2x}{dt^2} = f\left(x, \frac{dx}{dt}, t\right). \tag{8.34}$$

That is, the second derivative can be any arbitrary function, including possibly a nonlinear function, of x, t, and the derivative dx/dt. So we could have, for instance,

$$\frac{d^2x}{dt^2} = \frac{1}{x}\left(\frac{dx}{dt}\right)^2 + 2\frac{dx}{dt} - x^3e^{-4t}. \tag{8.35}$$

Now here's the trick. We define a new quantity y by

$$\frac{dx}{dt} = y, \tag{8.36}$$

in terms of which Eq. (8.34) can be written

$$\frac{dy}{dt} = f(x, y, t). \tag{8.37}$$

Between them, Eqs. (8.36) and (8.37) are equivalent to the one second-order equation we started with, as we can prove by substituting (8.36) into (8.37) to recover (8.34) again. But (8.36) and (8.37) are both first order. So this process reduces our second-order equation to two simultaneous first-order equations. And we already know how to solve simultaneous first-order equations, so we can now use the techniques we have learned to solve our second-order equation as well.

We can do a similar trick for higher-order equations. For instance, the general form of a third-order equation is

$$\frac{d^3x}{dt^3} = f\left(x, \frac{dx}{dt}, \frac{d^2x}{dt^2}, t\right). \tag{8.38}$$

We define two additional variables y and z by

$$\frac{dx}{dt} = y, \qquad \frac{dy}{dt} = z, \tag{8.39}$$

so that Eq. (8.38) becomes

$$\frac{dz}{dt} = f(x, y, z, t). \tag{8.40}$$

Between them Eqs. (8.39) and (8.40) give us three first-order equations that are equivalent to our one third-order equation, so again we can solve using the methods we already know about for simultaneous first-order equations.

This approach can be generalized to equations of any order, although equations of order higher than three are rare in physics, so you probably won't need to solve them often.

The method can also be generalized in a straightforward manner to equations with more than one dependent variable—the variables become vectors but the basic equations are the same as above. Thus a set of simultaneous second-order equations can be written in vector form as

$$\frac{d^2\mathbf{r}}{dt^2} = \mathbf{f}\left(\mathbf{r}, \frac{d\mathbf{r}}{dt}, t\right), \tag{8.41}$$

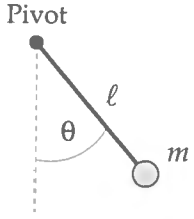
which is equivalent to the first-order equations

$$\frac{d\mathbf{r}}{dt} = \mathbf{s}, \qquad \frac{d\mathbf{s}}{dt} = \mathbf{f}(\mathbf{r}, \mathbf{s}, t). \tag{8.42}$$

If we started off with two simultaneous second-order equations, for instance, then we would end up with *four* simultaneous first-order equations after applying the transformation above. More generally, an initial system of n equations of mth order becomes a system of $m \times n$ simultaneous first-order equations, which we can solve by the standard methods.

EXAMPLE 8.6: THE NONLINEAR PENDULUM

A standard problem in physics is the linear pendulum, where you approximate the behavior of a pendulum by a linear differential equation than can be solved exactly. But a real pendulum is nonlinear. Consider a pendulum with an arm of length ℓ holding a bob of mass m:



In terms of the angle θ of displacement of the arm from the vertical, the acceleration of the mass is $\ell\, d^2\theta/dt^2$ in the tangential direction. Meanwhile the force on the mass is vertically downward with magnitude mg, where $g = 9.81\ \mathrm{m\,s^{-2}}$ is the acceleration due to gravity and, for the sake of simplicity, we are ignoring friction and assuming the arm to be massless. The component of this force in

the tangential direction is $mg\sin\theta$, always toward the rest point at $\theta = 0$, and hence Newton's second law gives us an equation of motion for the pendulum of the form

$$m\ell\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -mg\sin\theta, \tag{8.43}$$

or equivalently

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{\ell}\sin\theta. \tag{8.44}$$

Because it is nonlinear it is not easy to solve this equation analytically, and no exact solution is known. But a solution on the computer is straightforward. We first use the trick described in the previous section to turn the second-order equation, Eq. (8.44), into two first-order equations. We define a new variable ω by

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = \omega. \tag{8.45}$$

Then Eq. (8.44) becomes

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = -\frac{g}{\ell}\sin\theta. \tag{8.46}$$

Between them, these two first-order equations are equivalent to the one second-order equation we started with. Now we combine the two variables θ and ω into a single vector $\mathbf{r} = (\theta, \omega)$ and apply the fourth-order Runge–Kutta method in vector form to solve the two equations simultaneously. We are only really interested in the solution for one of the variables, the variable θ. The method gives us the solution for both, but we can simply ignore the value of ω if we don't need it. The program will be similar to that of Example 8.5, except that the function `f(r,t)` must be redefined appropriately. If the arm of the pendulum were 10 cm long, for example, we would have

```
g = 9.81
l = 0.1

def f(r,t):
    theta = r[0]
    omega = r[1]
    ftheta = omega
    fomega = -(g/l)*sin(theta)
    return array([ftheta,fomega],float)
```

The rest of the program is left to you—see Exercise 8.4.

Exercise 8.4: Building on the results from Example 8.6 above, calculate the motion of a nonlinear pendulum as follows.

a) Write a program to solve the two first-order equations, Eqs. (8.45) and (8.46), using the fourth-order Runge–Kutta method for a pendulum with a 10 cm arm. Use your program to calculate the angle θ of displacement for several periods of the pendulum when it is released from a standstill at $\theta = 179°$ from the vertical. Make a graph of θ as a function of time.

b) Extend your program to create an animation of the motion of the pendulum. Your animation should, at a minimum, include a representation of the moving pendulum bob and the pendulum arm. (Hint: You will probably find the function `rate` discussed in Section 3.5 useful for making your animation run at a sensible speed. Also, you may want to make the step size for your Runge–Kutta calculation smaller than the frame-rate of your animation, i.e., do several Runge–Kutta steps per frame on screen. This is certainly allowed and may help to make your calculation more accurate.)

For a bigger challenge, take a look at Exercise 8.15 on page 398, which invites you to write a program to calculate the chaotic motion of the double pendulum.

Exercise 8.5: The driven pendulum

A pendulum like the one in Exercise 8.4 can be driven by, for example, exerting a small oscillating force horizontally on the mass. Then the equation of motion for the pendulum becomes

$$\frac{d^2\theta}{dt^2} = -\frac{g}{\ell}\sin\theta + C\cos\theta\sin\Omega t,$$

where C and Ω are constants.

a) Write a program to solve this equation for θ as a function of time with $\ell = 10\,\text{cm}$, $C = 2\,\text{s}^{-2}$ and $\Omega = 5\,\text{s}^{-1}$ and make a plot of θ as a function of time from $t = 0$ to $t = 100\,\text{s}$. Start the pendulum at rest with $\theta = 0$ and $d\theta/dt = 0$.

b) Now change the value of Ω, while keeping C the same, to find a value for which the pendulum resonates with the driving force and swings widely from side to side. Make a plot for this case also.

Exercise 8.6: Harmonic and anharmonic oscillators

The simple harmonic oscillator arises in many physical problems, in mechanics, electricity and magnetism, and condensed matter physics, among other areas. Consider the standard oscillator equation

$$\frac{d^2x}{dt^2} = -\omega^2 x.$$

a) Using the methods described in the preceding section, turn this second-order equation into two coupled first-order equations. Then write a program to solve them for the case $\omega = 1$ in the range from $t = 0$ to $t = 50$. A second-order equation requires two initial conditions, one on x and one on its derivative. For this problem use $x = 1$ and $dx/dt = 0$ as initial conditions. Have your program make a graph showing the value of x as a function of time.

b) Now increase the amplitude of the oscillations by making the initial value of x bigger—say $x = 2$—and confirm that the period of the oscillations stays roughly the same.

c) Modify your program to solve for the motion of the anharmonic oscillator described by the equation

$$\frac{d^2x}{dt^2} = -\omega^2 x^3.$$

Again take $\omega = 1$ and initial conditions $x = 1$ and $dx/dt = 0$ and make a plot of the motion of the oscillator. Again increase the amplitude. You should observe that the oscillator oscillates faster at higher amplitudes. (You can try lower amplitudes too if you like, which should be slower.) The variation of frequency with amplitude in an anharmonic oscillator was studied previously in Exercise 5.10.

d) Modify your program so that instead of plotting x against t, it plots dx/dt against x, i.e., the "velocity" of the oscillator against its "position." Such a plot is called a *phase space* plot.

e) The *van der Pol oscillator*, which appears in electronic circuits and in laser physics, is described by the equation

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + \omega^2 x = 0.$$

Modify your program to solve this equation from $t = 0$ to $t = 20$ and hence make a phase space plot for the van der Pol oscillator with $\omega = 1$, $\mu = 1$, and initial conditions $x = 1$ and $dx/dt = 0$. Try it also for $\mu = 2$ and $\mu = 4$ (still with $\omega = 1$). Make sure you use a small enough value of the time interval h to get a smooth, accurate phase space plot.

Exercise 8.7: Trajectory with air resistance

Many elementary mechanics problems deal with the physics of objects moving or flying through the air, but they almost always ignore friction and air resistance to make the equations solvable. If we're using a computer, however, we don't need solvable equations.

Consider, for instance, a spherical cannonball shot from a cannon standing on level ground. The air resistance on a moving sphere is a force in the opposite direction to the motion with magnitude

$$F = \tfrac{1}{2}\pi R^2 \rho C v^2,$$

where R is the sphere's radius, ρ is the density of air, v is the velocity, and C is the so-called *coefficient of drag* (a property of the shape of the moving object, in this case a sphere).

a) Starting from Newton's second law, $F = ma$, show that the equations of motion for the position (x, y) of the cannonball are

$$\ddot{x} = -\frac{\pi R^2 \rho C}{2m}\,\dot{x}\sqrt{\dot{x}^2 + \dot{y}^2}, \qquad \ddot{y} = -g - \frac{\pi R^2 \rho C}{2m}\,\dot{y}\sqrt{\dot{x}^2 + \dot{y}^2},$$

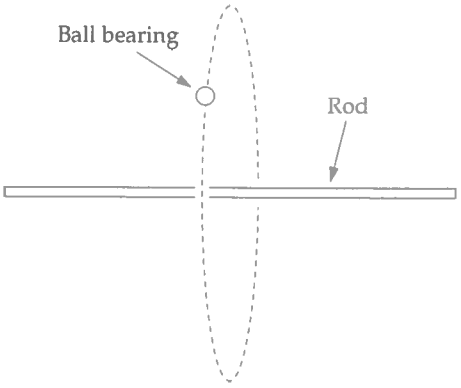
where m is the mass of the cannonball, g is the acceleration due to gravity, and $\dot{x}$ and $\ddot{x}$ are the first and second derivatives of x with respect to time.

b) Change these two second-order equations into four first-order equations using the methods you have learned, then write a program that solves the equations for a cannonball of mass 1 kg and radius 8 cm, shot at 30° to the horizontal with initial velocity 100 m s^{-1}. The density of air is $\rho = 1.22$ kg m^{-3} and the coefficient of drag for a sphere is $C = 0.47$. Make a plot of the trajectory of the cannonball (i.e., a graph of y as a function of x).

c) When one ignores air resistance, the distance traveled by a projectile does not depend on the mass of the projectile. In real life, however, mass certainly does make a difference. Use your program to estimate the total distance traveled (over horizontal ground) by the cannonball above, and then experiment with the program to determine whether the cannonball travels further if it is heavier or lighter. You could, for instance, plot a series of trajectories for cannonballs of different masses, or you could make a graph of distance traveled as a function of mass. Describe briefly what you discover.

Exercise 8.8: Space garbage

A heavy steel rod and a spherical ball-bearing, discarded by a passing spaceship, are floating in zero gravity and the ball bearing is orbiting around the rod under the effect of its gravitational pull:



For simplicity we'll assume that the rod is of negligible cross-section and heavy enough that it doesn't move significantly, and that the ball bearing is orbiting around the rod's mid-point in a plane perpendicular to the rod.

a) Treating the rod as a line of mass M and length L and the ball bearing as a point mass m, show that the attractive force F felt by the ball bearing in the direction toward the center of the rod is given by

$$F = \frac{GMm}{L}\sqrt{x^2+y^2}\int_{-L/2}^{L/2}\frac{dz}{(x^2+y^2+z^2)^{3/2}},$$

where G is Newton's gravitational constant and x and y are the coordinates of the ball bearing in the plane perpendicular to the rod. The integral can be done in closed form and gives

$$F = \frac{GMm}{\sqrt{(x^2+y^2)(x^2+y^2+L^2/4)}}.$$

Hence show that the equations of motion for the position x, y of the ball bearing in the xy-plane are

$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^2\sqrt{r^2+L^2/4}}, \qquad \frac{d^2y}{dt^2} = -GM\frac{y}{r^2\sqrt{r^2+L^2/4}},$$

where $r = \sqrt{x^2+y^2}$.

b) Convert these two second-order equations into four first-order ones using the techniques of Section 8.3. Then, working in units where $G = 1$, write a program to solve them for $M = 10$, $L = 2$, and initial conditions $(x,y) = (1,0)$ with velocity of $+1$ in the y direction. Calculate the orbit from $t = 0$ to $t = 10$ and make a plot of it, meaning a plot of y against x. You should find that the ball bearing does not orbit in a circle or ellipse as a planet does, but has a precessing orbit, which arises because the attractive force is not a simple $1/r^2$ force as it is for a planet orbiting the Sun.

Exercise 8.9: Vibration in a one-dimensional system

In Example 6.2 on page 235 we studied the motion of a system of N identical masses (in zero gravity) joined by identical linear springs like this:

As we showed, the horizontal displacements ξ_i of masses $i = 1 \ldots N$ satisfy equations of motion

$$m\frac{d^2\xi_1}{dt^2} = k(\xi_2 - \xi_1) + F_1,$$
$$m\frac{d^2\xi_i}{dt^2} = k(\xi_{i+1} - \xi_i) + k(\xi_{i-1} - \xi_i) + F_i,$$
$$m\frac{d^2\xi_N}{dt^2} = k(\xi_{N-1} - \xi_N) + F_N.$$

where m is the mass, k is the spring constant, and F_i is the external force on mass i. In Example 6.2 we showed how these equations could be solved by guessing a form for the solution and using a matrix method. Here we'll solve them more directly.

a) Write a program to solve for the motion of the masses using the fourth-order Runge–Kutta method for the case we studied previously where $m = 1$ and $k = 6$, and the driving forces are all zero except for $F_1 = \cos \omega t$ with $\omega = 2$. Plot your solutions for the displacements ξ_i of all the masses as a function of time from $t = 0$ to $t = 20$ on the same plot. Write your program to work with general N, but test it out for small values—$N = 5$ is a reasonable choice.

You will need first of all to convert the N second-order equations of motion into $2N$ first-order equations. Then combine all of the dependent variables in those equations into a single large vector $\mathbf{r}$ to which you can apply the Runge–Kutta method in the standard fashion.

b) Modify your program to create an animation of the movement of the masses, represented as spheres on the computer screen. You will probably find the `rate` function discussed in Section 3.5 useful for making your animation run at a sensible speed.

8.4 VARYING THE STEP SIZE

The methods we have seen so far in this chapter all use repeated steps of a fixed size h, the size being chosen by you, the programmer. In most situations, however, we can get better results if we allow the step size to vary during the running of the program, with the program choosing the best value at each step.

Suppose we are solving a first-order differential equation of the general form $dx/dt = f(x,t)$ and suppose as a function of time the solution looks something like Fig. 8.6. In some regions the function is slowly varying, in which case we can accurately capture its shape with only a few, widely spaced points. But in the central region of the figure the function varies rapidly and in this region we need points that are more closely spaced. If we are allowed to vary the size h of our steps, making them large in the regions where the solution varies little and small when we need more detail, then we can calculate the whole solution faster (because we need fewer points overall) but still very accurately (because we use small step sizes in the regions where they are needed). This type of scheme is called an *adaptive step size* method, and some version of it is used in most large-scale numerical solutions of differential equations.

The basic idea behind an adaptive step size scheme is to vary the step sizes h so that the error introduced per unit interval in t is roughly constant.

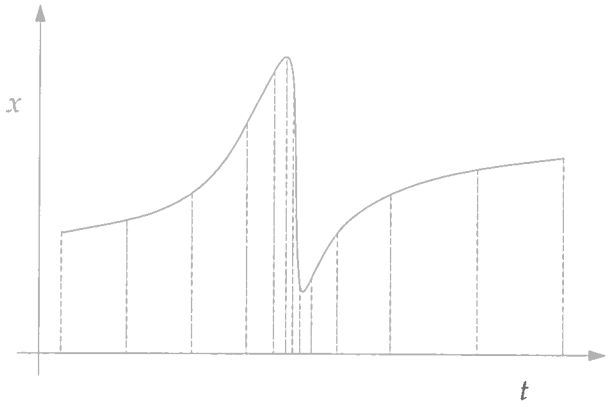


Figure 8.6: Adaptive step sizes. When solving a differential equation whose solution $x(t)$ varies slowly with t in some regions but more rapidly in others, it makes sense to use a varying step size. When the solution is slowly varying a large step size will give good results with less computational effort. When the solution is rapidly varying we must use smaller steps to get good accuracy.

For instance, we might specify that we want an error of 0.001 per unit time, or less, so that if we calculate a solution from say $t = 0$ to $t = 10$ we will get a total error of 0.01 or less. We achieve this by making the step size smaller in regions where the solution is tricky, but we must be careful because if we use smaller steps we will also need to take more steps and the errors pile up, so each individual step will have to be more accurate overall.

In practice the adaptive step size method has two parts. First we have to estimate the error on our steps, then we compare that error to our required accuracy and either increase or decrease the step size to achieve the accuracy we want. Here's how the approach works when applied to the fourth-order Runge–Kutta method.

We choose some initial value of h—typically very small, to be on the safe side—and, using our ordinary Runge–Kutta method, we first do *two* steps of the solution, each of size h, one after another—see Fig. 8.7. So if we start at time t, we will after two steps get to time $t + 2h$ and get an estimate of $x(t + 2h)$. Now here's the clever part: we go back to the start again, to time t, and we do one more Runge–Kutta step, but this time of twice the size, i.e., of size $2h$. This third larger step also takes us to time $t + 2h$ and gives us another estimate of $x(t + 2h)$, which will usually be close to but slightly different from the first estimate, since it was calculated in a different way. It turns out that by comparing

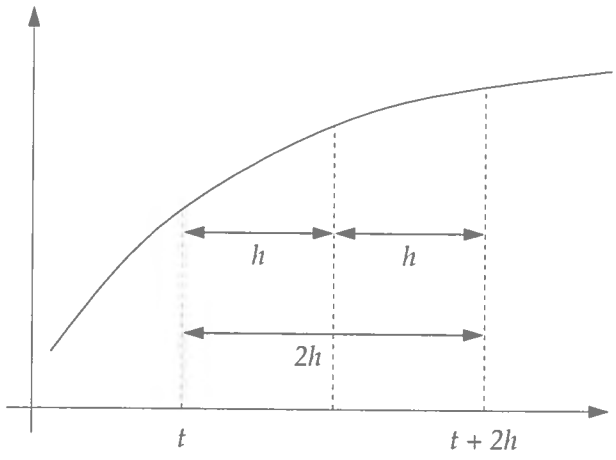


Figure 8.7: The adaptive step size method. Starting from some time t, the method involves first taking two steps of size h each, then going back to t again and taking a single step of size $2h$. Both give us estimates of the solution at time $t + 2h$ and by comparing these we can estimate the error.

the two estimates we can tell how accurate our calculation is.

The fourth-order Runge–Kutta method is *accurate* to fourth order but the *error* on the method is fifth order. That is, the size of the error on a single step is ch^5 to leading order for some constant c. So if we start at time t and do two steps of size h then the error will be roughly $2ch^5$. That is, the true value of $x(t + 2h)$ is related to our estimated value, call it x_1, by

$$x(t + 2h) = x_1 + 2ch^5. \tag{8.47}$$

On the other hand, when we do a single large step of size $2h$ the error is $c(2h)^5 = 32ch^5$, and so

$$x(t + 2h) = x_2 + 32ch^5, \tag{8.48}$$

where x_2 is our second estimate of $x(t + 2h)$. Equating these two expressions we get $x_1 = x_2 + 30ch^5$, which implies that the per-step error $\epsilon = ch^5$ on steps of size h is

$$\epsilon = ch^5 = \tfrac{1}{30}(x_1 - x_2). \tag{8.49}$$

Our goal is to make the size of this error exactly equal to some target accuracy that we choose. In general, unless we are very lucky, the two will not be exactly equal. Either Eq. (8.49) will be better than the target, which means

we are performing steps that are smaller than they need to be and hence wasting time, or it will be worse than the target, which is unacceptable—the whole point here is to perform a calculation that meets the specified target accuracy.

So let us ask the following question: what size would our steps have to be to make the size of the error in Eq. (8.49) exactly equal to the target, to make our calculation exactly as accurate as we need it to be but not more? Let us denote this perfect step size h'. If we were to take steps of size h' then the error on a single step would be

$$\epsilon' = ch'^5 = ch^5\left(\frac{h'}{h}\right)^5 = \tfrac{1}{30}(x_1 - x_2)\left(\frac{h'}{h}\right)^5, \tag{8.50}$$

where we have used Eq. (8.49). At the same time suppose that the target accuracy per unit time for our calculation is δ, which means that the target accuracy for a single step of size h' would be $h'\delta$. We want to find the value of h' such that the actual accuracy (8.50) is equal to this target accuracy. We are only interested in the absolute magnitude of the error, not its sign, so we want the h' that satisfies

$$\tfrac{1}{30}|x_1 - x_2|\left(\frac{h'}{h}\right)^5 = h'\delta. \tag{8.51}$$

Rearranging for h' we then find that

$$h' = h\left(\frac{30h\delta}{|x_1 - x_2|}\right)^{1/4} = h\rho^{1/4}, \tag{8.52}$$

where

$$\rho = \frac{30h\delta}{|x_1 - x_2|} \tag{8.53}$$

which is precisely the ratio of the target accuracy $h\delta$ and the actual accuracy $\frac{1}{30}|x_1 - x_2|$ for steps of size h.

The complete method is now as follows. We perform two steps of size h and then, starting from the same starting point, one step of size $2h$. This gives us our two estimates x_1 and x_2 of $x(t + 2h)$. We use these to calculate the ratio ρ in Eq. (8.53). If $\rho > 1$ then we know that the actual accuracy of our Runge–Kutta steps is better than the target accuracy, so our calculation is fine, in the sense that it meets the target, but it is wasteful because it is using steps that are smaller than they need to be. So we keep the results and move on to time $t + 2h$ to continue our solution, but we make our steps bigger the next time around to avoid this waste. Plugging our value of ρ into Eq. (8.52) tells us exactly what the new larger value h' of the step size should be to achieve this.

Conversely, if $\rho < 1$ then the actual accuracy of our calculation is poorer than the target accuracy—we have missed our target and the current step of the calculation has failed. In this case we need to repeat the current step again, but with a smaller step size, and again Eq. (8.52) tells us what that step size should be.

Thus, after each step of the process, depending on the value of ρ, we either increase the value of h and move on to the next step or decrease the value of h and repeat the current step. Note that for the actual solution of our differential equation we always use the estimate x_1 for the value of x, not the estimate x_2, since x_1 was made using smaller steps and is thus, in general, more accurate. The estimate x_2 made with the larger step is used only for calculating the error and updating the step size, never for calculating the final solution.

The adaptive step size method involves more work for the computer than methods that use a fixed step size—we have to do at least three Runge–Kutta steps for every two that we actually use in calculating the solution, and sometimes more than three in cases where we have to repeat a step because we missed our target accuracy. However, the extra effort usually pays off because the method gets you an answer with the accuracy you require with very little waste. In the end the program almost always takes less time to run, and usually much less.

It is possible, by chance, for the two estimates x_1 and x_2 to coincidentally agree with one another—errors are inherently unpredictable and the two can occasionally be the same or roughly the same just by luck. If this happens, h' in Eq. (8.52) can erroneously become very large or diverge, causing the calculation to break down. To prevent this, one commonly places an upper limit on how much the value of h can increase from one step to another. For instance, a common rule of thumb is that it should not increase by more than a factor of two on any given pair of steps (pairs of successive steps being the fundamental unit in the method described here).

The adaptive step size method can be used to solve simultaneous differential equations as well as single equations. In such cases we need to decide how to generalize the formula (8.49) for the error, or equivalently the formula (8.53) for the ratio ρ, to the case of more than one dependent variable. The derivation leading to Eq. (8.49) can be duplicated for each variable to show that variables x, y, etc. have separate errors

$$\epsilon_x = \tfrac{1}{30}(x_1 - x_2), \qquad \epsilon_y = \tfrac{1}{30}(y_1 - y_2), \tag{8.54}$$

and so forth. There is, however, more than one way that these separate errors

can be combined into a single overall error for use in Eq. (8.53), depending on the particular needs of the calculation. For instance, if we have variables x and y that represent coordinates of a point in a two-dimensional space, we might wish to perform a calculation that ensures that the Euclidean error in the position of the point meets a certain target, where by Euclidean error we mean $\sqrt{\epsilon_x^2 + \epsilon_y^2}$. In that case it is straightforward to see that we would use the same formulas for the adaptive method as before, except that $\frac{1}{30}|x_1 - x_2|$ in Eq. (8.53) should be replaced with $\sqrt{\epsilon_x^2 + \epsilon_y^2}$. On the other hand, suppose we are performing a calculation like that of Example 8.6 for the nonlinear pendulum, where we are solving a single second-order equation for θ but we introduce an additional variable ω to turn the problem into two first-order equations. In that case we don't really care about ω—it is introduced only for convenience—and its accuracy doesn't matter so long as θ is calculated accurately. In this situation we would use Eq. (8.53) directly, with x replaced by θ, and ignore ω in the calculation of the step sizes. (An example of such a calculation for the nonlinear pendulum is given below.) Thus it may take a little thought to determine, for any particular calculation, what the appropriate generalization of the adaptive method is to simultaneous equations, but the answer usually becomes clear once one determines the correct definition for the error on the calculation.

One further point is worth making about the adaptive step size method. It may seem unnecessarily strict to insist that we repeat the current step of the calculation if we miss our target accuracy. One might imagine that one could get reasonable answers if we always moved on to the next step, even when we miss our target: certainly there will be some steps where the error is a little bigger than the target value, but there will be others where it is a little smaller, and with luck it might all just wash out in the end—the total error at the end of the calculation would be roughly, if not exactly, where we want it to be. Unfortunately, however, this usually doesn't work. If one takes this approach, then one often ends up with a calculation that significantly misses the required accuracy target because there are a few steps that have unusually large errors. The problem is that the errors are cumulative—a large error on even one step makes all subsequent steps inaccurate too. If errors fluctuate from step to step then at some point you are going to get an undesirably large error which can doom the entire calculation. Thus it really is important to repeat steps that miss the target accuracy, rather than just letting them slip past, so that you can be certain no step has a very large error.

As an example of the adaptive step size method let us return once more to the nonlinear pendulum of Example 8.6. Figure 8.8 shows the results of a

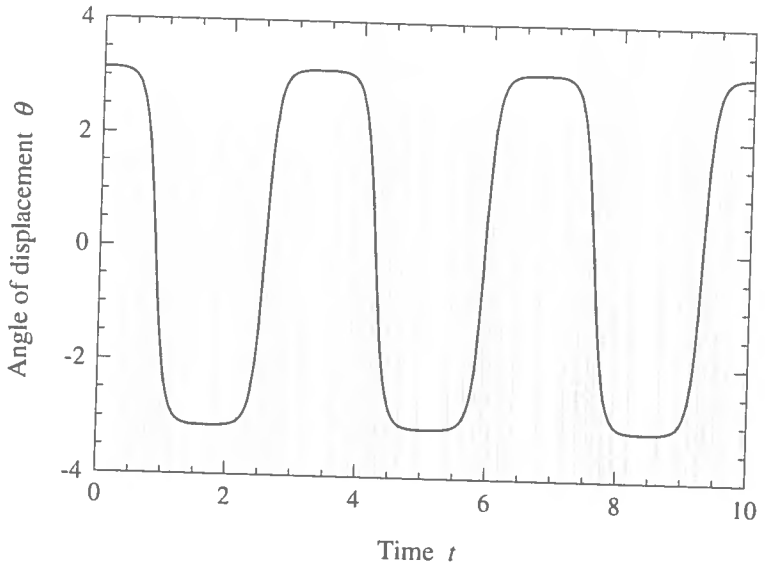
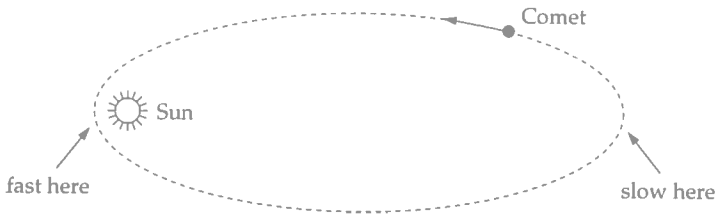


Figure 8.8: Motion of a nonlinear pendulum. This figure shows the angle θ of displacement of a nonlinear pendulum from the vertical as a function of time, calculated using the adaptive step size approach described in this section. The vertical dotted lines indicate the position of every twentieth Runge–Kutta step.

calculation of the motion of such a pendulum using adaptive step sizes. The solid curve shows the angle of displacement of the pendulum as a function of time—the wavelike form indicates that it's swinging back and forth. The vertical lines in the plot show the position of every twentieth Runge–Kutta step in the calculation (i.e., every tenth iteration of the adaptive method, since we always take two Runge–Kutta steps at once). As you can see from the figure, the method makes the step sizes longer in the flat portions of the curve at the top and bottom of each swing where little is happening, but in the steep portions where the pendulum is moving rapidly the step sizes are much smaller, which ensures accurate calculations of the motion.

Exercise 8.10: Cometary orbits

Many comets travel in highly elongated orbits around the Sun. For much of their lives they are far out in the solar system, moving very slowly, but on rare occasions their orbit brings them close to the Sun for a fly-by and for a brief period of time they move very fast indeed:



This is a classic example of a system for which an adaptive step size method is useful, because for the large periods of time when the comet is moving slowly we can use long time-steps, so that the program runs quickly, but short time-steps are crucial in the brief but fast-moving period close to the Sun.

The differential equation obeyed by a comet is straightforward to derive. The force between the Sun, with mass M at the origin, and a comet of mass m with position vector $\mathbf{r}$ is GMm/r^2 in direction $-\mathbf{r}/r$ (i.e., the direction towards the Sun), and hence Newton's second law tells us that

$$m\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -\left(\frac{GMm}{r^2}\right)\frac{\mathbf{r}}{r}.$$

Canceling the m and taking the x component we have

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -GM\frac{x}{r^3},$$

and similarly for the other two coordinates. We can, however, throw out one of the coordinates because the comet stays in a single plane as it orbits. If we orient our axes so that this plane is perpendicular to the z-axis, we can forget about the z coordinate and we are left with just two second-order equations to solve:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -GM\frac{x}{r^3}, \qquad \frac{\mathrm{d}^2y}{\mathrm{d}t^2} = -GM\frac{y}{r^3},$$

where $r = \sqrt{x^2+y^2}$.

a) Turn these two second-order equations into four first-order equations, using the methods you have learned.

b) Write a program to solve your equations using the fourth-order Runge–Kutta method with a *fixed* step size. You will need to look up the mass of the Sun and Newton's gravitational constant G. As an initial condition, take a comet at coordinates $x = 4$ billion kilometers and $y = 0$ (which is somewhere out around the orbit of Neptune) with initial velocity $v_x = 0$ and $v_y = 500\,\mathrm{m\,s^{-1}}$. Make a graph showing the trajectory of the comet (i.e., a plot of y against x).

Choose a fixed step size h that allows you to accurately calculate at least two full orbits of the comet. Since orbits are periodic, a good indicator of an accurate calculation is that successive orbits of the comet lie on top of one another on your plot. If they do not then you need a smaller value of h. Give a short description

of your findings. What value of h did you use? What did you observe in your simulation? How long did the calculation take?

c) Make a copy of your program and modify the copy to do the calculation using an adaptive step size. Set a target accuracy of $\delta = 1$ kilometer per year in the position of the comet and again plot the trajectory. What do you see? How do the speed, accuracy, and step size of the calculation compare with those in part (b)?

d) Modify your program to place dots on your graph showing the position of the comet at each Runge–Kutta step around a single orbit. You should see the steps getting closer together when the comet is close to the Sun and further apart when it is far out in the solar system.

Calculations like this can be extended to cases where we have more than one orbiting body—see Exercise 8.16 for an example. We can include planets, moons, asteroids, and others. Analytic calculations are impossible for such complex systems, but with careful numerical solution of differential equations we can calculate the motions of objects throughout the entire solar system.

Here's one further interesting wrinkle to the adaptive method. Recall Eq. (8.47), which relates the results of a "double step" of the method to the solution of the differential equation:

$$x(t+2h) = x_1 + 2ch^5 + \mathrm{O}(h^6). \tag{8.55}$$

(We have added the $\mathrm{O}(h^6)$ here to remind us of the next term in the series.)

We also know from Eq. (8.49) that

$$ch^5 = \tfrac{1}{30}(x_1 - x_2), \tag{8.56}$$

where x_1 and x_2 are the two estimates of $x(t+2h)$ calculated in the adaptive method. Substituting (8.56) into (8.55), we find that

$$x(t+2h) = x_1 + \tfrac{1}{15}(x_1 - x_2) + \mathrm{O}(h^6), \tag{8.57}$$

which is now accurate to order h^5 and has a error of order h^6—one order better than the standard fourth-order Runge–Kutta method. Equation (8.57) involves only quantities we have already computed in the course of the adaptive method, so it's essentially no extra work to calculate this more accurate estimate of the solution.

This trick is called *local extrapolation*. It is a kind of free bonus prize that comes along with the adaptive method, giving us a more accurate answer for no extra work. The only catch with it is that we don't know the size of the

error on Eq. (8.57). It is, presumably, better than the error on the old fourth-order result (which is $2ch^5$, with ch^5 given by Eq. (8.56)), but we don't know by how much.

It is an easy extra step to incorporate local extrapolation into adaptive calculations. We could have used it in our solution of the motion of the pendulum in Fig. 8.8, for example. You could use it if you do Exercise 8.10 on calculating cometary orbits. It typically offers at least a modest improvement in the accuracy of your results.

The real interest in extrapolation, however, arises when we take the method further. It is possible to use methods similar to this not only to eliminate the leading-order error (the $O(h^5)$ term in Eq. (8.55)), but also any number of higher-order terms as well, resulting in impressively accurate solutions to differential equations even when using quite large values of h. The technique for doing this is called Richardson extrapolation and it's the basis of one of the most powerful methods for solving differential equations. Richardson extrapolation, however, is not usually used with the Runge–Kutta method, but rather with another method, called the "modified midpoint method," which we will examine in Section 8.5.4.

8.5 OTHER METHODS FOR DIFFERENTIAL EQUATIONS

So far in this chapter we have concentrated our attention on the Runge–Kutta method for solving differential equations. The Runge–Kutta method is a robust, accurate method that's easy to program and gives good results in most cases. It is, however, not the only method available. There are a number of other methods for solving differential equations that, while less widely used than the Runge–Kutta method, are nonetheless useful in certain situations. In this section we look at several additional methods, including the leapfrog and Verlet methods, and the Bulirsch–Stoer method, which combines a modified version of the leapfrog method with Richardson extrapolation to create one of the most accurate methods for solving differential equations (although it is also quite complex to program).

8.5.1 THE LEAPFROG METHOD

Consider a first-order differential equation in a single variable:

$$\frac{dx}{dt} = f(x,t). \tag{8.58}$$

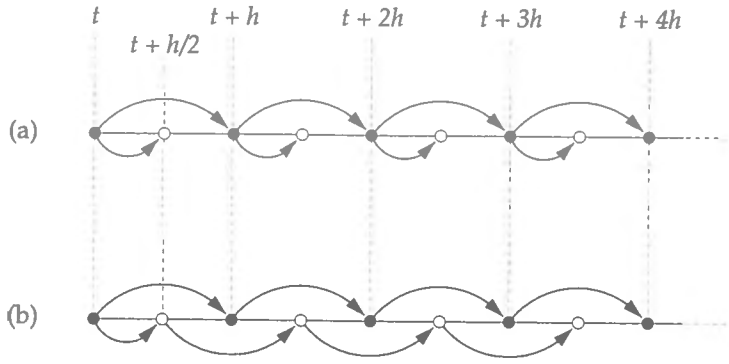


Figure 8.9: Second-order Runge–Kutta and the leapfrog method. (a) A diagrammatic representation of the calculations involved in the second-order Runge–Kutta method. On every step we use the starting position to calculate a value at the midpoint (open circle), then use that value to calculate the value at the end of the interval (filled circle). (b) The leapfrog method starts out the same, with a half step to the first midpoint and a full step to the end of the first interval. But thereafter each midpoint is calculated from the previous midpoint.

In Section 8.1.2 we introduced the second-order Runge–Kutta method (also sometimes called the midpoint method) in which, given the value of the dependent variable x at time t, one estimates its value at $t + h$ by using the slope at the midpoint $f(x(t + \frac{1}{2}h), t + \frac{1}{2}h)$. Because one doesn't normally know the value $x(t + \frac{1}{2}h)$, one first estimates it using Euler's method. The equations for the method can be written thus:

$$x(t + \tfrac{1}{2}h) = x(t) + \tfrac{1}{2}hf(x,t), \tag{8.59a}$$

$$x(t + h) = x(t) + hf(x(t + \tfrac{1}{2}h), t + \tfrac{1}{2}h). \tag{8.59b}$$

This is a slightly different way of writing the equations from the one we used previously (see Eq. (8.13)) but it is equivalent and it will be convenient for what follows.

The second-order Runge–Kutta method involves using these equations repeatedly to calculate the value of x at intervals of h as far as we wish to go. Each step is accurate to order h^2 and has an error of order h^3. When we combine many steps, one after another, the total error is one order of h worse (see Section 8.1.1), meaning it is of order h^2 in this case.

Figure 8.9a shows a simple graphical representation of what the Runge–Kutta method is doing. At each step we calculate the solution at the midpoint,

and then use that solution as a stepping stone to calculate the value at $t + h$. The *leapfrog method* is a variant on this idea, as depicted in Fig. 8.9b. This method starts out the same way as second-order Runge–Kutta, with a half-step to the midpoint, follow by a full step to calculate $x(t+h)$. But then, for the next step, rather than calculating the midpoint value from $x(t+h)$ as we would in the Runge–Kutta method, we instead calculate it from the previous midpoint value $x(t+\frac{1}{2}h)$. In mathematical language we have

$$x\bigl(t+\tfrac{3}{2}h\bigr) = x\bigl(t+\tfrac{1}{2}h\bigr) + hf(x(t+h), t+h). \tag{8.60}$$

In this calculation $f(x(t+h), t+h)$ plays the role of the gradient at the midpoint between $t+\frac{1}{2}h$ and $t+\frac{3}{2}h$, so the calculation has second-order accuracy again and a third-order error. Moreover, once we have $x\bigl(t+\frac{3}{2}h\bigr)$ we can use it to do the next full step thus:

$$x(t+2h) = x(t+h) + hf\bigl(x\bigl(t+\tfrac{3}{2}h\bigr), t+\tfrac{3}{2}h\bigr). \tag{8.61}$$

And we can go on repeating this process as long as we like. Given values of $x(t)$ and $x\bigl(t+\frac{1}{2}h\bigr)$, we repeatedly apply the equations

$$x(t+h) = x(t) + hf\bigl(x\bigl(t+\tfrac{1}{2}h\bigr), t+\tfrac{1}{2}h\bigr), \tag{8.62a}$$

$$x\bigl(t+\tfrac{3}{2}h\bigr) = x\bigl(t+\tfrac{1}{2}h\bigr) + hf(x(t+h), t+h). \tag{8.62b}$$

This is the leapfrog method, so called because each step "leaps over" the position of the previously calculated value. Like the second-order Runge–Kutta method, each step of the method is accurate to order h^2 and carries an error of order h^3. If we compound many steps of size h then the final result is accurate to order h and carries an h^2 error. The method can be extended to the solution of simultaneous differential equations just as the Runge–Kutta method can, by replacing the single variable x with a vector $\mathbf{r}$ and the function $f(x,t)$ with a vector function $\mathbf{f}(\mathbf{r},t)$:

$$\mathbf{r}(t+h) = \mathbf{r}(t) + h\mathbf{f}\bigl(\mathbf{r}\bigl(t+\tfrac{1}{2}h\bigr), t+\tfrac{1}{2}h\bigr), \tag{8.63a}$$

$$\mathbf{r}\bigl(t+\tfrac{3}{2}h\bigr) = \mathbf{r}\bigl(t+\tfrac{1}{2}h\bigr) + h\mathbf{f}(\mathbf{r}(t+h), t+h). \tag{8.63b}$$

It can also be extended to the solution of second- or higher-order equations by converting the equations into simultaneous first-order equations, as shown in Section 8.3.

On the face of it, however, it's not immediately clear why we would want to use this method. It's true it is quite simple, but the fourth-order Runge–Kutta

method is not much more complicated and significantly more accurate for almost all calculations. But the leapfrog method has two significant virtues that make it worth considering. First, it is time-reversal symmetric, which makes it useful for physics problems where energy conservation is important. And second, its error is even in the step size h, which makes it ideal as a starting point for the Richardson extrapolation method mentioned at the end of Section 8.4. In the following sections we look at these issues in more detail.

8.5.2 TIME REVERSAL AND ENERGY CONSERVATION

The leapfrog method is time-reversal symmetric. When we use the method to solve a differential equation, the state of the calculation at any time t_1 is completely specified by giving the two values $x(t_1)$ and $x(t_1 + \frac{1}{2}h)$. Given only these values the rest of the solution going forward in time can be calculated by repeated application of Eq. (8.62). Suppose we continue the solution to a later time $t = t_2$, calculating values up to and including $x(t_2)$ and $x(t_2 + \frac{1}{2}h)$. Time-reversal symmetry means that if we take these values and use the leapfrog method backwards, with time interval $-h$ equal to minus the interval we used in the forward calculation, then we will retrace our steps and recover the exact values $x(t_1)$ and $x(t_1 + \frac{1}{2}h)$ at time t_1 (apart from any rounding error).

To see this let us set $h \to -h$ in Eq. (8.62):

$$x(t-h) = x(t) - hf\bigl(x(t-\tfrac{1}{2}h), t-\tfrac{1}{2}h\bigr), \tag{8.64a}$$

$$x\bigl(t-\tfrac{3}{2}h\bigr) = x\bigl(t-\tfrac{1}{2}h\bigr) - hf\bigl(x(t-h), t-h\bigr). \tag{8.64b}$$

Now put $t \to t + \frac{3}{2}h$ and we get

$$x\bigl(t+\tfrac{1}{2}h\bigr) = x\bigl(t+\tfrac{3}{2}h\bigr) - hf\bigl(x(t+h), t+h\bigr), \tag{8.65a}$$

$$x(t) = x(t+h) - hf\bigl(x\bigl(t+\tfrac{1}{2}h\bigr), t+\tfrac{1}{2}h\bigr). \tag{8.65b}$$

These equations give us the values of $x(t)$ and $x\bigl(t+\frac{1}{2}h\bigr)$ in terms of $x(t+h)$ and $x\bigl(t+\frac{3}{2}h\bigr)$. But if you compare these equations to Eq. (8.62), you'll see that they are simply performing the same mathematical operations as the forward calculation, only in reverse—everywhere we previously added a term $hf(x,t)$ we now subtract it again. Thus when we use the leapfrog method with step size $-h$ to solve a differential equation backwards, we get the exact same values $x(t)$ at every time-step that we get when we solve the equation forwards.

The same is not true of, for example, the second-order Runge–Kutta method. If you put $h \to -h$ in Eq. (8.59), the resulting equations do not give you the same mathematical operations as the forward Runge–Kutta method. The

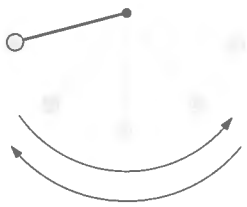
method will give you a solution in either the forward or backward direction, but the solutions will not agree exactly, in general, even after you allow for rounding error.

Why is time-reversal symmetry important? It turns out that it has a couple of useful implications. One concerns the conservation of energy.

Consider as an illustration the frictionless nonlinear pendulum, which we studied in Example 8.6. The motion of the pendulum is given by Eqs. (8.45) and (8.46), which read

$$\frac{d\theta}{dt} = \omega, \qquad \frac{d\omega}{dt} = -\frac{g}{\ell}\sin\theta. \tag{8.66}$$

If we solve these equations using a Runge–Kutta method we can get a pretty good solution, as shown in Fig. 8.8 on page 361, but it is nonetheless only approximate, as nearly all computer calculations are. Among other things, this means that the total energy of the system, kinetic plus potential, is only approximately constant during the calculation. A frictionless pendulum should have constant energy, but the Runge–Kutta method isn't perfect and energies calculated using it tend to fluctuate and drift slightly over time. The top panel of Fig. 8.10 shows results from a solution of the equations above using the second-order Runge–Kutta method and the drift of the total energy with time is clearly visible. (We have deliberately used the less accurate second-order method in this case to make the drift larger and easier to see. With the fourth-order Runge–Kutta method, which is more accurate, the drift would be significantly smaller, though it would still be there.)



We consider one full swing of the pendulum, starting on one side and swinging across then back.

Now suppose we solve the same differential equations using the leapfrog method. Imagine doing so for one full swing of the pendulum. The pendulum starts at the furthest limit of its swing, swings all the way across, then all the way back again. In real life, the total energy of the system must remain constant throughout the motion, and in particular it must be the same when the pendulum returns to its initial point as it was when it started out. Our solution using the leapfrog method, on the other hand, is only approximate, so it's possible the energy might drift. Let us suppose for the sake of argument that it drifts upward, as it did for the Runge–Kutta method in the top panel of Fig. 8.10, so that its value at the end of the swing is slightly higher than at the beginning.

Now let us calculate the pendulum's motion once again, still using the leapfrog method but this time in reverse, starting at the end of the swing and solving backwards, with minus the step size that we used in our forward calculation. As we have shown, when we run the leapfrog method backwards in

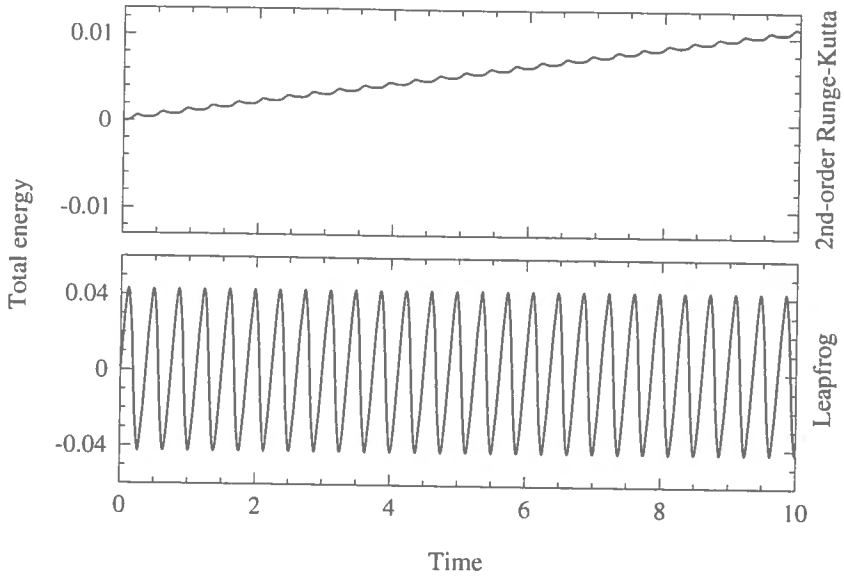


Figure 8.10: Total energy of the nonlinear pendulum. Top: The total energy, potential plus kinetic, of a nonlinear pendulum as a function of time, calculated using the second-order Runge–Kutta method. Bottom: The same energy calculated using the leapfrog method. Neither is constant, but the leapfrog method returns to the same value at the end of each swing of the pendulum and so conserves energy in the long run, while the energy calculated with Runge–Kutta drifts steadily away from the true value as time passes.

this fashion it will retrace its steps and end up exactly at the starting point of the motion again (apart from rounding error). Thus, if the energy increased during the forward calculation it must decrease when we do things in reverse.

But here's the thing. The physics of the pendulum is itself time-reversal symmetric. The motion of swinging across and back, the motion that the pendulum makes in a single period, is exactly the same backwards as it is forwards. Hence, when we perform the backward solution we are solving for the exact same motion and moreover doing it using the exact same method (since we are using the leapfrog method in both directions). This means that the values of the variables θ and ω will be exactly the same at each successive step of the solution in the reverse direction as they were going forward. Hence, if the energy increased during the forward solution it must also increase during the backward one.

Now we have a contradiction. We have shown that if the energy increases during the forward calculation then it must both decrease and increase during the backward one. Clearly this is impossible—it cannot do both—and hence we conclude that it cannot have increased during the forward calculation. An analogous argument shows it cannot decrease either, so the only remaining possibility is that it stays the same. In other words, the leapfrog method conserves energy. The total energy of the system will stay constant over time when we solve the equations using the leapfrog method, except for any small changes introduced by rounding error.

There are a couple of caveats. First, even though the energy is conserved we should not make the mistake of assuming this means our solution for the motion is exact. It isn't. The leapfrog method only gives approximate solutions for differential equations—as discussed in Section 8.5.1 the method is only accurate to second order on each step and has a third-order error. So the values we get for the angle θ for our pendulum, for example, will not be exactly correct, even though the energy is constant.

Second, the argument we have given applies to a full swing of the pendulum. It tells us that the energy at the end of a full swing will be the same as it was at the beginning. It does not tell us that the energy will be conserved throughout the swing, and indeed, as we will see, it is not. The energy may fluctuate during the course of the pendulum swing, but it will always come back to the correct value at the end of the swing. More generally, if the leapfrog method is used to solve equations of motion for any periodic system, such as a pendulum or a planet orbiting a star, then energy will be conserved over any full period of the system (or many full periods), but it will not, in general, be conserved over fractions of a period.

If we can live with these limitations, however, the leapfrog method can be useful for solving the equations of motion of energy conserving physical systems over long periods of time. If we wait long enough, a solution using a Runge–Kutta method will drift in energy—the pendulum might run down and stop swinging, or the planet might fall out of orbit and into its star. But a solution using the leapfrog method will run forever.

As an example look again at Fig. 8.10. The bottom panel shows the total energy of the nonlinear pendulum calculated using the leapfrog method and we can see that indeed it is constant on average over long periods of time—many swings of the pendulum—even though it oscillates over the course of individual swings. As the figure shows, the accuracy of the energy in the short term is actually poorer than the second-order Runge–Kutta method (notice that

the vertical scales are different in the two panels), but in the long term the leapfrog method will be far better than the Runge–Kutta method, as the latter drifts further and further from the true value of the energy.

8.5.3 THE VERLET METHOD

Suppose, as in the previous section, that we are using the leapfrog method to solve the classical equations of motion for a physical system. Such equations, derived from Newton's second law $F = ma$, take the form of second-order differential equations

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = f(x,t), \tag{8.67}$$

or the vector equivalent when there is more than one dependent variable. Examples include the motion of projectiles, the pendulum of the previous section, and the cometary orbit of Exercise 8.10. As we have seen, we can convert such equations of motion into coupled first-order equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v, \qquad \frac{\mathrm{d}v}{\mathrm{d}t} = f(x,t), \tag{8.68}$$

where we use the variable name v here as a reminder that, when we are talking about equations of motion, the quantity it represents is a velocity (or sometimes an angular velocity, as in the case of the pendulum).

If we want to apply the leapfrog method to these equations the normal strategy would be to define a vector $\mathbf{r} = (x,v)$, combine the two equations (8.68) into a single vector equation

$$\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} = \mathbf{f}(\mathbf{r},t), \tag{8.69}$$

and then solve this equation for $\mathbf{r}$ using the leapfrog method.

Rather than going this route, however, let us instead write out the leapfrog method in full, as applied to (8.68). If we are given the value of x at some time t and the value of v at time $t + \frac{1}{2}h$ then, applying the method, the value of x a time interval h later is

$$x(t+h) = x(t) + hv\bigl(t+\tfrac{1}{2}h\bigr). \tag{8.70}$$

And the value of v an interval h later is

$$v\bigl(t+\tfrac{3}{2}h\bigr) = v\bigl(t+\tfrac{1}{2}h\bigr) + hf\bigl(x(t+h),t+h\bigr). \tag{8.71}$$

We can derive a full solution to the problem by using just these two equations repeatedly, as many times as we wish. Notice that the equations involve the

value of x only at time t plus integer multiples of h and the value of v only at half-integer multiples. We never need to calculate v at any of the integer points or x at the half integers. This is an improvement over the normal leapfrog method applied to the vector $\mathbf{r} = (x, v)$, which would involve solving for both x and v at all points, integer and half-integer. Equations (8.70) and (8.71) require only half as much work to evaluate as the full leapfrog method.

This simplification works only for equations of motion or other differential equations that have the special structure of Eq. (8.68), where the right-hand side of the first equation depends on v but not x and the right-hand side of the second equation depends on x but not v. Many physics problems, however, boil down to solving equations of motion, so the method is widely applicable.

A minor problem with the method arises if we want to calculate some quantity that depends on both x and v, such as a the total energy of the system. Potential energy depends on position x while kinetic energy depends on velocity v, so calculating the total energy, potential plus kinetic, at any time t, requires us to know the values of both variables at that time. Unfortunately we know x only at the integer points and v only at the half-integer points, so we never know both at the same time.

But there's an easy solution to this problem. We can calculate the velocity at the integer points by doing an additional half step as follows. If we did know $v(t+h)$ then we could calculate $v(t+\frac{1}{2}h)$ from it by doing a half step backwards using Euler's method. That is, we would do Euler's method with a step size of $-\frac{1}{2}h$:

$$v(t+\tfrac{1}{2}h) = v(t+h) - \tfrac{1}{2}hf(x(t+h), t+h). \tag{8.72}$$

Alternatively, by rearranging this equation we can calculate $v(t+h)$ from $v(t+\frac{1}{2}h)$ like this:

$$v(t+h) = v(t+\tfrac{1}{2}h) + \tfrac{1}{2}hf(x(t+h), t+h). \tag{8.73}$$

This equation gives us v at integer steps in terms of quantities we already know from our leapfrog calculation, allowing us to calculate total energy (or any other quantity) at each step.

A complete calculation combines Eqs. (8.70) and (8.71) with Eq. (8.73), plus an initial half step at the very beginning to get everything started. Putting it all together, here's what we have.

We are given the initial values of x and v at some time t. Then

$$v(t+\tfrac{1}{2}h) = v(t) + \tfrac{1}{2}hf(x(t), t). \tag{8.74}$$

Then subsequent values of x and v are derived by repeatedly applying

$$x(t+h) = x(t) + hv\left(t+\tfrac{1}{2}h\right), \tag{8.75a}$$
$$k = hf(x(t+h), t+h), \tag{8.75b}$$
$$v(t+h) = v\left(t+\tfrac{1}{2}h\right) + \tfrac{1}{2}k, \tag{8.75c}$$
$$v\left(t+\tfrac{3}{2}h\right) = v\left(t+\tfrac{1}{2}h\right) + k. \tag{8.75d}$$

This variant of the leapfrog method is called the *Verlet method* after physicist Loup Verlet, who discovered it in the 1960s (although it was known to others long before that, as far back as the eighteenth century).

The method can be easily extended to equations of motion in more than one dimension. If we wish to solve an equation of motion of the form

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = \mathbf{f}(\mathbf{r}, t), \tag{8.76}$$

where $\mathbf{r} = (x, y, \ldots)$ is a d-dimensional vector, then, given initial conditions on $\mathbf{r}$ and the velocity $\mathbf{v} = \mathrm{d}\mathbf{r}/\mathrm{d}t$, the appropriate generalization of the Verlet method involves first performing a half step to calculate $\mathbf{v}\left(t+\frac{1}{2}h\right)$:

$$\mathbf{v}\left(t+\tfrac{1}{2}h\right) = \mathbf{v}(t) + \tfrac{1}{2}h\mathbf{f}(\mathbf{r}(t), t), \tag{8.77}$$

then repeatedly applying the equations

$$\mathbf{r}(t+h) = \mathbf{r}(t) + h\mathbf{v}\left(t+\tfrac{1}{2}h\right), \tag{8.78a}$$
$$\mathbf{k} = h\mathbf{f}(\mathbf{r}(t+h), t+h), \tag{8.78b}$$
$$\mathbf{v}(t+h) = \mathbf{v}\left(t+\tfrac{1}{2}h\right) + \tfrac{1}{2}\mathbf{k}, \tag{8.78c}$$
$$\mathbf{v}\left(t+\tfrac{3}{2}h\right) = \mathbf{v}\left(t+\tfrac{1}{2}h\right) + \mathbf{k}. \tag{8.78d}$$

Exercise 8.11: Write a program to solve the differential equation

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = \left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2 + x + 5 = 0$$

using the leapfrog method. Solve from $t = 0$ to $t = 50$ in steps of $h = 0.001$ with initial condition $x = 1$ and $\mathrm{d}x/\mathrm{d}t = 0$. Make a plot of your solution showing x as a function of t.

Exercise 8.12: Orbit of the Earth

Use the Verlet method to calculate the orbit of the Earth around the Sun. The equations of motion for the position $\mathbf{r} = (x, y)$ of the planet in its orbital plane are the same as those for any orbiting body and are derived in Exercise 8.10 on page 361. In vector form, they are

$$\frac{d^2\mathbf{r}}{dt^2} = -GM\frac{\mathbf{r}}{r^3},$$

where $G = 6.6738 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$ is Newton's gravitational constant and $M = 1.9891 \times 10^{30}\,\mathrm{kg}$ is the mass of the Sun.

The orbit of the Earth is not perfectly circular, the planet being sometimes closer to and sometimes further from the Sun. When it is at its closest point, or *perihelion*, it is moving precisely tangentially (i.e., perpendicular to the line between itself and the Sun) and it has distance $1.4710 \times 10^{11}\,\mathrm{m}$ from the Sun and linear velocity $3.0287 \times 10^4\,\mathrm{m\,s^{-1}}$.

a) Write a program to calculate the orbit of the Earth using the Verlet method, Eqs. (8.77) and (8.78), with a time-step of $h = 1$ hour. Make a plot of the orbit, showing several complete revolutions about the Sun. The orbit should be very slightly, but visibly, non-circular.

b) The gravitational potential energy of the Earth is $-GMm/r$, where $m = 5.9722 \times 10^{24}\,\mathrm{kg}$ is the mass of the planet, and its kinetic energy is $\frac{1}{2}mv^2$ as usual. Modify your program to calculate both of these quantities at each step, along with their sum (which is the total energy), and make a plot showing all three as a function of time on the same axes. You should find that the potential and kinetic energies vary visibly during the course of an orbit, but the total energy remains constant.

c) Now plot the total energy alone without the others and you should be able to see a slight variation over the course of an orbit. Because you're using the Verlet method, however, which conserves energy in the long term, the energy should always return to its starting value at the end of each complete orbit.

8.5.4 THE MODIFIED MIDPOINT METHOD

The leapfrog method offers another, more subtle, advantage over the Runge–Kutta method, namely that the total error on the method, after many steps, is an even function of the step size h. To put that another way, the expansion of the error in powers of h contains only even terms and no odd ones. This result will be crucial in the next section, where it forms the basis for a powerful solution method for differential equations called the Bulirsch–Stoer method.

The argument that the error on the leapfrog method is even in h relies once more on the fact that the method is time-reversal symmetric, as discussed in Section 8.5.2. Recall that a single step of the leapfrog method is accurate up to terms in h^2 and has an h^3 error to leading order. More generally, we can write

the error on a single step as some function $\epsilon(h)$ of the step size, with the first term in that function being proportional to h^3. The question is what the other terms look like.

Imagine taking a small step using the leapfrog method, which gives the solution to our differential equation a short time later plus error $\epsilon(h)$. Now imagine taking the same step backwards, i.e., with step size $-h$. Given that the leapfrog method is time-reversal symmetric—the change in the solution going backwards is exactly the reverse of the change forwards—it follows that the backward error $\epsilon(-h)$ must be minus the forward one:

$$\epsilon(-h) = -\epsilon(h). \tag{8.79}$$

This equation tells us that $\epsilon(h)$ is an odd function. It is antisymmetric about the origin and its Taylor expansion in powers of h can contain only odd powers. We know the first term in the series is proportional to h^3, so in general $\epsilon(h)$ must take the form

$$\epsilon(h) = c_3h^3 + c_5h^5 + c_7h^7 + \ldots \tag{8.80}$$

and so on, for some set of constants $c_3, c_5, \ldots$

But, as we've also seen, first for the Euler method in Section 8.1.1 and later for the Runge–Kutta and leapfrog methods as well, if you perform many steps of size h then the total, cumulative error over all of them is one order worse in h than it is for each single step. Roughly speaking, if the error on a single step is $\epsilon(h)$ and it takes Δ/h steps to cover an interval of time Δ, then the total error is of order $\epsilon(h) \times \Delta/h$, which is one order lower in h than $\epsilon(h)$ itself. Hence, given that in the present case $\epsilon(h)$ contains only odd powers of h starting with h^3, the total error on the leapfrog method, after many steps, must contain only even powers, starting with h^2.

This result is correct as far as it goes, but there is a catch. Recall that to get the leapfrog method started, we initially take one half-step using Euler's method, as in Eq. (8.59a) on page 365. This additional step introduces an error of its own. That error is of size h^2 to leading order (as always with Euler's method), which is the same as the overall error on the leapfrog method and hence makes the final error no worse in terms of the order of h. Unfortunately, however, the higher-order terms in Euler's method are not restricted to even powers of h—all powers from h^2 onward are present, meaning that this one extra step at the start of the calculation spoils our result about even powers above. The total error at the end of the whole calculation will now contain both even and odd powers.

There is, however, a solution for this problem. Suppose we wish to solve our differential equation from some initial time t forward to a later time $t + H$ (where H is not necessarily small) using n leapfrog steps of size $h = H/n$ each. Let us write the leapfrog method in a slightly different form from before. We define

$$x_0 = x(t), \tag{8.81a}$$
$$y_1 = x_0 + \tfrac{1}{2}hf(x_0, t). \tag{8.81b}$$

Then

$$x_1 = x_0 + hf(y_1, t + \tfrac{1}{2}h), \tag{8.82a}$$
$$y_2 = y_1 + hf(x_1, t + h), \tag{8.82b}$$
$$x_2 = x_1 + hf(y_2, t + \tfrac{3}{2}h), \tag{8.82c}$$

and so forth. The variables x_m here represent the solution at integer multiples of h and the variables y_m at half-integer multiples. In general, we have

$$y_{m+1} = y_m + hf(x_m, t + mh), \tag{8.83a}$$
$$x_{m+1} = x_m + hf(y_{m+1}, t + (m + \tfrac{1}{2})h). \tag{8.83b}$$

The last two points in the solution are $y_n = x(t + H - \frac{1}{2}h)$ and $x_n = x(t + H)$. Normally, we would take the value of x_n as our final solution for $x(t + H)$, but there is another possibility: we can also calculate a final value from y_n. Using the same trick that we used to derive Eq. (8.73), we can write

$$x(t + H) = y_n + \tfrac{1}{2}hf(x_n, t + H). \tag{8.84}$$

Thus we have two different ways to calculate a value for $x(t + H)$. Or we can combine the two, Eq. (8.84) and the estimate $x(t + H) = x_n$, taking their average thus:

$$x(t + H) = \tfrac{1}{2}\left[x_n + y_n + \tfrac{1}{2}hf(x_n, t + H)\right]. \tag{8.85}$$

Miraculously, it turns out that if we calculate $x(t + H)$ from this equation then the odd-order error terms that arise from the Euler's method step at the start of the leapfrog calculation cancel out, giving a total error on Eq. (8.85) that once again contains only even powers of h. This result was originally proved by mathematician William Gragg in 1965 and the resulting method is sometimes called *Gragg's method* in his honor, although it is more commonly referred to as the *modified midpoint method*. The modified midpoint method combines the leapfrog method in the form of Eqs. (8.81) and (8.83) with Eq. (8.85) to make an

estimate of $x(t+H)$ that carries a leading-order error of order h^2 and higher-order terms containing even powers of h only.

The modified midpoint method is rarely used alone, since it offers little advantage over either the ordinary leapfrog method (if you want an energy conserving solution) or the fourth-order Runge–Kutta method (which is significantly more accurate). It plays an important role, nonetheless, as the basis for the powerful Bulirsch–Stoer method, which we study in the next section.

8.5.5 THE BULIRSCH–STOER METHOD

The Bulirsch–Stoer method for solving differential equations combines two ideas we have seen already: the modified midpoint method and Richardson extrapolation. It's reminiscent in some ways of the Romberg method for evaluating integrals that we studied in Section 5.4, and the equations are similar. Here's how it works.

We are given a differential equation, and for now let's again assume the simplest case of a first-order, single-variable equation:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x,t). \tag{8.86}$$

We are also, as usual, given an initial condition at some time t, and, as in the previous section, let us solve the equation over an interval of time from t to some later time $t+H$.

We start by calculating a solution using the modified midpoint method and, in the first instance, we will use just a single step for the whole solution, from t to $t+H$ (or you can think of it as two half-steps—see Eq. (8.81)). In other words our step size for the modified midpoint method, which we'll call h_1, will just be equal to H. This gives an estimate of the value of $x(t+H)$, which we will denote $R_{1,1}$. (The R is for "Richardson extrapolation.") If H is a large interval of time then $R_{1,1}$ will be a rather crude estimate, but that need not worry us, as we'll see.

Once we have calculated $R_{1,1}$, we go back to the start at time t again and repeat our calculation, again using the modified midpoint method, but this time with two steps of size $h_2 = \frac{1}{2}H$. This gives us a second estimate of $x(t+H)$, which we'll call $R_{2,1}$.

We showed in the previous section that the total error on the modified midpoint method is an even function of the step size, so it follows that

$$x(t+H) = R_{2,1} + c_1 h_2^2 + \mathrm{O}(h_2^4), \tag{8.87}$$

where c_1 is an unknown constant. Similarly

$$x(t+H) = R_{1,1} + c_1 h_1^2 + O(h_1^4) = R_{1,1} + 4c_1 h_2^2 + O(h_2^4), \tag{8.88}$$

where we have used the fact that $h_1 = 2h_2$. Since Eqs. (8.87) and (8.88) are both expressions for the same quantity we can equate them and, after rearranging, we find that

$$c_1 h_2^2 = \tfrac{1}{3}(R_{2,1} - R_{1,1}). \tag{8.89}$$

Substituting this back into Eq. (8.87), we get

$$x(t+H) = R_{2,1} + \tfrac{1}{3}(R_{2,1} - R_{1,1}) + O(h_2^4). \tag{8.90}$$

In other words, we have found a new estimate of $x(t+H)$ which is more accurate than either of the estimates that went into it—it has an error of order h^4, two orders in h better than the basic leapfrog method and as good as the fourth-order Runge–Kutta method (which, when you add up errors over more than one step, is accurate to order h^3 and carries an h^4 error). Let us call this new estimate $R_{2,2}$:

$$R_{2,2} = R_{2,1} + \tfrac{1}{3}(R_{2,1} - R_{1,1}). \tag{8.91}$$

We can take this approach further. If we increase the number of steps to three, with step size $h_3 = \frac{1}{3}H$, and solve from t to $t+H$ again we get a new estimate $R_{3,1}$. Then, following the same line of argument as above, we can calculate a further estimate

$$R_{3,2} = R_{3,1} + \tfrac{4}{5}(R_{3,1} - R_{2,1}), \tag{8.92}$$

which has an error of order h_3^4. This allows us to write

$$x(t+H) = R_{3,2} + c_2 h_3^4 + O(h_3^6), \tag{8.93}$$

where c_2 is another constant. Combining Eqs. (8.90) and (8.91), we also have

$$x(t+H) = R_{2,2} + c_2 h_2^4 + O(h_2^6) = R_{2,2} + \tfrac{81}{16} c_2 h_3^4 + O(h_3^6), \tag{8.94}$$

where we have made use of the fact that $h_2 = \frac{3}{2}h_3$. Equating this result with (8.93) and rearranging gives

$$c_2 h_3^4 = \tfrac{16}{65}(R_{3,2} - R_{2,2}), \tag{8.95}$$

and substituting this into Eq. (8.93) gives

$$x(t+H) = R_{3,3} + O(h_3^6), \tag{8.96}$$

where

$$R_{3,3} = R_{3,2} + \tfrac{16}{65}(R_{3,2} - R_{2,2}). \tag{8.97}$$

Now our error is of order h^6, and we've taken only three modified midpoint steps!

The power of this method lies in the way it cancels out the error terms to higher and higher orders on successive steps, along with the fact that the modified midpoint method has only even-order error terms, which means that every time we cancel out another term we gain two extra orders of accuracy in h.

We can take this process as far as we like. Each time around, we solve our differential equation again from t to $t + H$ using the modified midpoint method, but with one more step than last time. Suppose we denote the current number of steps by n and our modified midpoint estimate estimate of $x(t + H)$ by $R_{n,1}$. Then we can use the method above to cancel error terms and arrive at a series of further estimates $R_{n,2}$, $R_{n,3}$, and so on, where $R_{n,m}$ carries an error of order h^{2m}:

$$x(t + H) = R_{n,m} + c_m h_n^{2m} + O(h_n^{2m+2}), \tag{8.98}$$

where c_m is an unknown constant. The corresponding estimate $R_{n-1,m}$ made with one less step satisfies

$$x(t + H) = R_{n-1,m} + c_m h_{n-1}^{2m} + O(h_{n-1}^{2m+2}). \tag{8.99}$$

But $h_n = H/n$ and $h_{n-1} = H/(n-1)$, so

$$h_{n-1} = \frac{n}{n-1} h_n. \tag{8.100}$$

Substituting this into (8.99), equating with (8.98), and rearranging, we then find that

$$c_m h_n^{2m} = \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)]^{2m} - 1}. \tag{8.101}$$

And putting this in Eq. (8.98) gives us a new estimate of $x(t + H)$ two orders of h more accurate:

$$x(t + H) = R_{n,m+1} + O(h_n^{2m+2}), \tag{8.102}$$

where

$$R_{n,m+1} = R_{n,m} + \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)]^{2m} - 1}. \tag{8.103}$$

Equation (8.103) is the fundamental equation of Richardson extrapolation, and the heart of the Bulirsch–Stoer solution method. It allows us to calculate

remarkably accurate estimates of $x(t+H)$ while only using a very few steps of the modified midpoint method.

A diagram may help to make the structure of the method clearer:

$$
\begin{array}{lccccc}
n=1: & R_{1,1} & & & & \\
 & & \searrow & & & \\
n=2: & R_{2,1} & \to & R_{2,2} & & \\
 & & \searrow & & \searrow & \\
n=3: & R_{3,1} & \to & R_{3,2} & \to & R_{3,3} \\
 & & \searrow & & \searrow & \searrow \\
n=4: & R_{4,1} & \to & R_{4,2} & \to & R_{4,3} \to R_{4,4}
\end{array}
$$

Modified midpoint — Richardson extrapolation

For each value of n we calculate a basic modified midpoint estimate $R_{n,1}$ with n steps, and then a series of further extrapolation estimates, working along a row of the diagram. Each extrapolation estimate depends on two previous estimates, as indicated by the arrows, and the last estimate in each row is the highest-order estimate for that value of n.

The method also gives us estimates of the error each time around. The quantity $c_m h_n^{2m}$ in Eq. (8.101) is precisely the (leading-order) error on the current estimate of $x(t+H)$. The Bulirsch–Stoer method involves increasing the number of steps n until the error on our best estimate of $x(t+H)$ is as small as we want it to be. As with the adaptive Runge–Kutta method of Section 8.4, we typically specify the accuracy we want in terms of the error per unit time δ, in which case the required accuracy for a solution over the interval H is $H\delta$. When the error falls below this value, the calculation is finished. Thus the Bulirsch–Stoer method is actually an adaptive method—it performs only as many steps as are needed to give us the accuracy we require.[3]

If you previously read Section 5.4, on the technique for calculating integrals known as Romberg integration, the diagram above may look familiar—it is similar to the one on page 161 in that section. This is no coincidence. Romberg integration and the Bulirsch–Stoer method are both applications of the same idea of Richardson extrapolation, to two different problems, and Eq. (8.103)

[3]Actually it goes one step further than we really need, since the error calculated in Eq. (8.101) is the error on the *previous* estimate $R_{n,m}$ and not the error on the new estimate $R_{n,m+1}$. So, just as with the local extrapolation method discussed at the end of Section 8.4, the results will usually be a little more accurate than our target accuracy.

for the Bulirsch–Stoer method embodies essentially the same idea as Eq. (5.51) for Romberg integration, though there are some differences. In particular, in Romberg integration we doubled the number of steps each time around, instead of just increasing it by one. This is a convenient choice when doing integrals because it gives us "nested" sample points that improve the speed of the calculations by allowing us to reuse previous results, as discussed in Section 5.3. There is no equivalent speed improvement to be had when solving differential equations, which is a shame in a sense, but does leave us free to choose the number of steps however we like. Various choices have been investigated and the results seem to indicate that the simple choice used here, of increasing n by one each time around, is a good one—better in most cases than doubling the value of n.

There are some limitations to the Bulirsch–Stoer method. One is that it only calculates a really accurate answer for the final value $x(t+H)$. At all the intermediate points we only get the raw midpoint-method estimates, which are not particularly accurate. (They carry an error of order h^2.)

Furthermore, we are, in effect, calculating the terms in a series expansion of $x(t+H)$ in powers of h and the method is only worthwhile if the series converges reasonably quickly. If you need hundreds or thousands of terms to get a good answer, then the Bulirsch–Stoer method is not a good choice. This means in practice that the interval H over which we are solving has to be kept reasonably small. Practical experience suggests that the method works best if the number of modified midpoint steps is never greater than about eight or ten, which limits the size of the time interval H to relatively modest values.

Both of these problems can be overcome with the same simple technique: if we want a solution from say $t = a$ to $t = b$, we divide that time into some number N of smaller intervals of size $H = (b-a)/N$ and apply the Bulirsch–Stoer method separately to each one in turn. We should choose N large enough that we get a complete picture of the solution without having to rely on the modified midpoint estimates in the interior of the intervals. And H should be small enough that in any one interval the number of modified midpoint steps needed to reach the target accuracy is never too large.

The complete Bulirsch–Stoer method is then as follows. Let δ be the desired accuracy of your solution per unit time. Divide the entire solution into N equal intervals of length H each and apply the following steps to solve your differential equation in each one in turn:

1. Set $n = 1$ and use the modified midpoint method of Section 8.5.4 to calculate an estimate $R_{1,1}$ of the solution from t to $t + H$ using just one step.

2. Increase n by one and calculate a new modified midpoint estimate $R_{n,1}$ with that many steps.
3. Use Eq. (8.103) to calculate further estimates $R_{n,2} \ldots R_{n,n}$—a complete row in the diagram on page 380.
4. After calculating the whole row, compare the error given by Eq. (8.101) with the target accuracy $H\delta$. If the error is larger than the target accuracy, go back to step 2 again. Otherwise, move on to the next time interval H.

When the entire solution has been covered in this way, the calculation ends.

The Bulirsch–Stoer method can easily be extended to the solution of simultaneous differential equations by replacing the single dependent variable x with a vector $\mathbf{r}$ of two or more variables, as in Section 8.2. Then the estimates $R_{n,m}$ also become vectors $\mathbf{R}_{n,m}$ but the equations for the method remain otherwise the same. We can also apply the Bulirsch–Stoer method to second- or higher-order differential equations by first transforming those equations into first-order ones, as described in Section 8.3.

Although it is somewhat more complicated to program than the Runge–Kutta method, the Bulirsch–Stoer method can work significantly better even than the adaptive version of Runge–Kutta, giving more accurate solutions with less work. Because, as we have said, it relies on a series expansion, the method is mainly useful for equations whose solutions are relatively smooth, so that expansions work well. Differential equations with pathological behaviors—large fluctuations, divergences, and so forth—are not suitable candidates. If you have a differential equation that displays such behaviors then the adaptive Runge–Kutta method of Section 8.4 is a better choice. But in cases where it is applicable, the Bulirsch–Stoer method is considered by many to be the best method available for solving ordinary differential equations, the king of differential equation solvers.

EXAMPLE 8.7: BULIRSCH–STOER METHOD FOR THE NONLINEAR PENDULUM

Let us return to the nonlinear pendulum, which we examined previously in Example 8.6 and Section 8.4. The equations of motion were given in Eqs. (8.45) and (8.46), which we repeat here:

$$\frac{d\theta}{dt} = \omega, \qquad \frac{d\omega}{dt} = -\frac{g}{\ell}\sin\theta. \tag{8.104}$$

Let us solve these equations for the case of a pendulum with an arm $\ell = 10\,\text{cm}$ long, initially at rest with $\theta = 179°$, i.e., pointing almost, but not quite, vertically upward. (These are the same conditions as in Exercise 8.4.) Here is a

complete program to solve for the motion of the pendulum using the Bulirsch–Stoer method:

File: bulirsch.py

```
from math import sin,pi
from numpy import empty,array,arange
from pylab import plot,show

g = 9.81
l = 0.1
theta0 = 179*pi/180
a = 0.0
b = 10.0
N = 100          # Number of "big steps"
H = (b-a)/N      # Size of "big steps"
delta = 1e-8     # Required position accuracy per unit time

def f(r):
    theta = r[0]
    omega = r[1]
    ftheta = omega
    fomega = -(g/l)*sin(theta)
    return array([ftheta,fomega],float)

tpoints = arange(a,b,H)
thetapoints = []
r = array([theta0,0.0],float)

# Do the "big steps" of size H
for t in tpoints:

    thetapoints.append(r[0])

    # Do one modified midpoint step of size H
    # to get things started
    n = 1
    r1 = r + 0.5*H*f(r)
    r2 = r + H*f(r1)

    # The array R1 stores the first row of the
    # extrapolation table, which contains only the single
    # modified midpoint estimate of the solution at the
    # end of the interval
    R1 = empty([1,2],float)
    R1[0] = 0.5*(r1 + r2 + 0.5*H*f(r2))
```

```
        # Now increase n until the required accuracy is reached
        error = 2*H*delta
        while error>H*delta:

            n += 1
            h = H/n

            # Modified midpoint method
            r1 = r + 0.5*h*f(r)
            r2 = r + h*f(r1)
            for i in range(n-1):
                r1 += h*f(r2)
                r2 += h*f(r1)

            # Calculate extrapolation estimates.  Arrays R1 and R2
            # hold the two most recent lines of the table
            R2 = R1
            R1 = empty([n,2],float)
            R1[0] = 0.5*(r1 + r2 + 0.5*h*f(r2))
            for m in range(1,n):
                epsilon = (R1[m-1]-R2[m-1])/((n/(n-1))**(2*m)-1)
                R1[m] = R1[m-1] + epsilon
            error = abs(epsilon[0])

        # Set r equal to the most accurate estimate we have,
        # before moving on to the next big step
        r = R1[n-1]

    # Plot the results
    plot(tpoints,thetapoints)
    plot(tpoints,thetapoints,"b.")
    show()
```

There are a couple of points worth noting about this program. Notice, for instance, how we gave the variable `error` an initial value of $2H\delta$, thus ensuring that we go around the while loop at least once. Notice also how we have used the two arrays `R1` and `R2` to store the most recent two rows of extrapolation estimates $R_{n,m}$. Since the calculation of each row requires only the values in the current and previous rows, and since, ultimately, we are only interested in the final value of the final row, there is no need to retain more than two rows of estimates at any time. The rest can be safely discarded.

If we run the program it produces a solution essentially identical to our previous solution of the same problem using the adaptive Runge–Kutta method, Fig. 8.8. The real difference between the two methods lies in the time it takes them to reach a solution. The Bulirsch–Stoer method in this case takes about 3800 modified midpoint steps in total (including the steps performed for each individual value of n). The Runge–Kutta method takes about 4200 steps to calculate a solution to the same accuracy, which at first glance doesn't seem very different from 3800. But a Runge–Kutta step takes more computer time than a modified midpoint step, requiring four evaluations of the function f where the modified midpoint method requires only two. This means that the total number of operations for the Runge–Kutta solution is around 16 800, compared with only about 7600 for the Bulirsch–Stoer solution.

The Bulirsch–Stoer method does require us to do some additional work to perform the Richardson extrapolation, but the computational effort involved is typically small by comparison with the rest of the calculation. So to a good approximation we expect to arrive at a solution about twice as fast with the Bulirsch–Stoer method as with adaptive Runge–Kutta. The running time of neither calculation was very great in this case—they both finished in seconds—but for a larger calculation, something more taxing than this modest example, an improvement of a factor of two could make a great deal of difference. The difference between a program that runs in a week and one that runs in two weeks is significant. Moreover, the advantages of the Bulirsch–Stoer method become more pronounced if we demand a more accurate solution by reducing the value of the accuracy parameter δ. This makes the method particularly attractive for cases where solutions of very high precision are required.

Before ending this section, we should mention that the method described here is not exactly the original Bulirsch–Stoer method as invented by Bulirsch and Stoer. There are two ways in which it differs from the original. First, the original method used a number n of modified midpoint steps that increased exponentially, doubling on successive steps rather than increasing by one.[4] As mentioned above, however, the current belief (based primarily on accumulated experience rather than any rigorous result) is that the method is more efficient when n just goes up by one each time. Second, the original Bulirsch–Stoer method used a different extrapolation scheme, based on rational approximants

[4]Specifically, it used a sequence of values $n = 2, 3, 4, 6, 8, 12, 16\ldots$ so that each value was twice the next-to-last one.

rather than the Richardson extrapolation described in this section, which uses polynomial approximants. It was originally thought that rational approximants gave more accurate results, but again experience has shown this not to be the case, and current thought favors the polynomial scheme.

Exercise 8.13: Planetary orbits

This exercise asks you to calculate the orbits of two of the planets using the Bulirsch–Stoer method. The method gives results significantly more accurate than the Verlet method used to calculate the Earth's orbit in Exercise 8.12.

The equations of motion for the position x, y of a planet in its orbital plane are the same as those for any orbiting body and are derived in Exercise 8.10 on page 361:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -GM\frac{x}{r^3}, \qquad \frac{\mathrm{d}^2y}{\mathrm{d}t^2} = -GM\frac{y}{r^3},$$

where $G = 6.6738 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$ is Newton's gravitational constant, $M = 1.9891 \times 10^{30}$ kg is the mass of the Sun, and $r = \sqrt{x^2 + y^2}$.

Let us first solve these equations for the orbit of the Earth, duplicating the results of Exercise 8.12, though with greater accuracy. The Earth's orbit is not perfectly circular, but rather slightly elliptical. When it is at its closest approach to the Sun, its perihelion, it is moving precisely tangentially (i.e., perpendicular to the line between itself and the Sun) and it has distance 1.4710×10^{11} m from the Sun and linear velocity $3.0287 \times 10^4\,\mathrm{m\,s^{-1}}$.

a) Write a program, or modify the one from Example 8.7, to calculate the orbit of the Earth using the Bulirsch–Stoer method to a positional accuracy of 1 km per year. Divide the orbit into intervals of length $H = 1$ week and then calculate the solution for each interval using the combined modified midpoint/Richardson extrapolation method described in this section. Make a plot of the orbit, showing at least one complete revolution about the Sun.

b) Modify your program to calculate the orbit of the dwarf planet Pluto. The distance between the Sun and Pluto at perihelion is 4.4368×10^{12} m and the linear velocity is $6.1218 \times 10^3\,\mathrm{m\,s^{-1}}$. Choose a suitable value for H to make your calculation run in reasonable time, while once again giving a solution accurate to 1 km per year.

You should find that the orbit of Pluto is significantly elliptical—much more so than the orbit of the Earth. Pluto is a Kuiper belt object, similar to a comet, and (unlike true planets) it's typical for such objects to have quite elliptical orbits.

8.5.6 INTERVAL SIZE FOR THE BULIRSCH–STOER METHOD

As we have said, the Bulirsch–Stoer method works best if we keep the number n of modified midpoint steps small, rising to at most eight or ten on any round of the calculation. To ensure this we must choose a suitable value of H, small enough that the extrapolation process converges to the required accuracy quickly. In Example 8.7 we set H manually and this works fine for simple problems. One can just use trial and error to find a suitable value.

For large-scale calculations, however, it's better to have the value of H chosen automatically by the computer. It saves time and ensures that the value used is always a good one. There are various adaptive schemes one can use to calculate a good H, but here's one that is robust and relatively simple.

Suppose, as previously, that we are solving our differential equation from time $t = a$ to $t = b$ and let us choose some initial number of intervals N into which we divide this time, so that the length of each interval is $H = (b - a)/N$. Normally the initial value of N will be a small number, like four, or two, or even just one.

Now we carry out the operations of the Bulirsch–Stoer method on each of the intervals in turn as normal, subdividing each one into n modified midpoint steps and then extrapolating the results as we increase the value of n. For each value of n we calculate the error, Eq. (8.101), on our results and if this error meets our required accuracy level then the calculation for the current interval is finished.

However, if n reaches a predetermined maximum value—typically around eight—and we have not yet met our accuracy goal, then we abandon the calculation and instead subdivide the current time interval of size H into two smaller intervals of size $\frac{1}{2}H$ each. Then we apply the Bulirsch–Stoer method to each of these smaller intervals in turn.

We continue this process as long as necessary, repeatedly subdividing intervals until we reach the required accuracy. Any interval that fails to meet our accuracy goal before the number of steps n reaches the allowed maximum, is subdivided into two parts and the method applied to the parts separately.

Note that different intervals may be subdivided different numbers of times. This means that the ultimate size of the intervals used for different parts of the solution may not be the same. The division of the complete solution into intervals might end up looking something like this:

In portions of the solution where a good result can be obtained with a larger value of H the method will take advantage of that fact. In other portions, where necessary, it will use a smaller value of H (as in the central portions above). Exercises 8.17 and 8.18 give you an opportunity to try out this scheme.

8.6 BOUNDARY VALUE PROBLEMS

All the examples we have considered so far in this chapter have been *initial value problems*, meaning that we are solving differential equations given the initial values of the variables. This is the most common form of differential equation problem encountered in physics, but it is not the only one. There are also *boundary value problems*.

Consider, for instance, the differential equation governing the height above the ground of a ball thrown in the air:

$$\frac{d^2x}{dt^2} = -g, \tag{8.105}$$

where g is the acceleration due to gravity and we're ignoring friction. To fix the solution of this equation we could specify initial conditions, two initial conditions being required for a second-order equation. We could, for instance, specify the initial height of the ball and its initial upward velocity. However there is another possibility: we could specify our two conditions by giving one initial condition and one *ending* condition. We could, for instance, specify that the ball has height $x = 0$ at $t = 0$ and that $x = 0$ again at some later time $t = t_1$. In other words, we are specifying the time at which the ball is thrown and the time at which it lands. Then our goal would be to find the solution that satisfies these conditions. This problem might arise for instance if we wished to calculate the trajectory of a projectile necessary to make it land at a specific point, which is a classic problem in artillery fire.

Problems of this kind are called boundary value problems. They are somewhat harder to solve computationally than the initial value problems we have looked at previously, but a solution can be achieved by combining two techniques that we have already seen, as follows.

8.6.1 THE SHOOTING METHOD

A fundamental technique for solving boundary value problems is the *shooting method*. The shooting method is a trial-and-error method that searches for the correct values of the initial conditions that match a given set of boundary

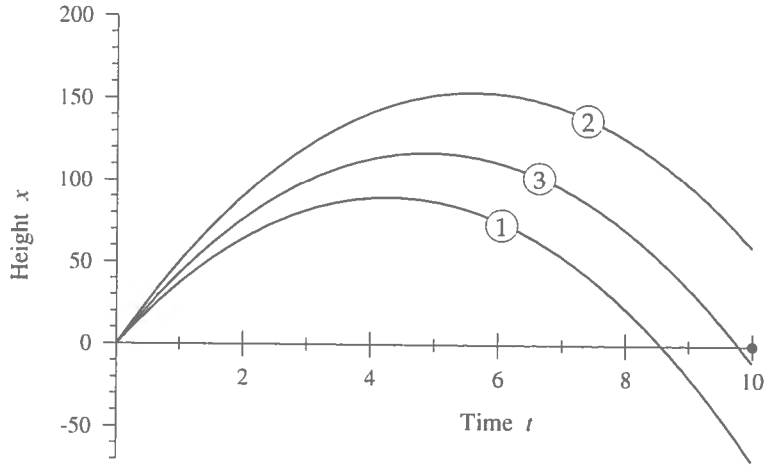


Figure 8.11: The shooting method. The shooting method allows us to match boundary conditions at the beginning and end of a solution. In this example we are solving for the trajectory of a thrown ball and require that its height x be zero—i.e., that it lands—at time $t = 10$. The shooting method involves making a guess as to the initial conditions that will achieve this. In this example, we undershoot our target (represented by the dot on the right) on our first guess (trajectory 1). On the second guess we overshoot. On the third we are closer, but still not perfect.

conditions, in effect turning the calculation back into an initial value problem. Consider the problem of the thrown ball above. In this problem we are given the initial position but not the velocity of the ball. All we know is that the ball lands a certain time later; the velocity is whatever it has to be to make this happen.

So we start by guessing a value for the initial upward velocity. Using this value we solve the differential equation and follow the ball until the time t_1 at which it is supposed to land and we ask whether it had height zero at that time—see Fig. 8.11. Probably it did not. Probably we overshot or undershot. In that case we change our guess for the initial velocity and try again.

The question is how exactly we should modify our guesses to converge to the correct value for the initial velocity. To understand how to do this, let us look at the problem in a slightly different way. In principle there is some function $x = f(v)$ which gives the height x of the ball at time t_1 as a function of the initial vertical velocity v. We don't know what this function is, but we can calculate it for any given value of v by solving the differential equation

with that initial velocity. Our goal in solving the boundary value problem is to find the value of v that makes the function zero. That is, we want to solve the equation $f(v) = 0$. But this is simply a matter of finding the root of the function $f(v)$ and we already know how to do that. We saw a number of methods for finding roots in Section 6.3, including binary search and the secant method. Either of these methods would work for the current problem.

So the shooting method involves using one of the standard methods for solving differential equations, such as the fourth-order Runge–Kutta method, to calculate the value of the function $f(v)$, which relates the unknown initial conditions to the final boundary condition(s). Then we use a root finding method such as binary search to find the value of this function that matches the given value of the boundary condition(s).

EXAMPLE 8.8: VERTICAL POSITION OF A THROWN BALL

Let us solve the problem above with the thrown ball for the case where the ball lands back at $x = 0$ after $t = 10$ seconds. The first step, as is normal for second-order equations, is to convert Eq. (8.105) into two first-order equations:

$$\frac{dx}{dt} = y, \qquad \frac{dy}{dt} = -g. \tag{8.106}$$

We will solve these using fourth-order Runge–Kutta, then perform a binary search to find the value of the initial velocity that matches the boundary conditions. Here is a program to accomplish the calculation:

File: `throw.py`

```
from numpy import array,arange

g = 9.81          # Acceleration due to gravity
a = 0.0           # Initial time
b = 10.0          # Final time
N = 1000          # Number of Runge-Kutta steps
h = (b-a)/N       # Size of Runge-Kutta steps
target = 1e-10    # Target accuracy for binary search

def f(r):
    x = r[0]
    y = r[1]
    fx = y
    fy = -g
    return array([fx,fy],float)
```

```
# Function to solve the equation and calculate the final height
def height(v):
    r = array([0.0,v],float)
    for t in arange(a,b,h):
        k1 = h*f(r)
        k2 = h*f(r+0.5*k1)
        k3 = h*f(r+0.5*k2)
        k4 = h*f(r+k3)
        r += (k1+2*k2+2*k3+k4)/6
    return r[0]

# Main program performs a binary search
v1 = 0.01
v2 = 1000.0
h1 = height(v1)
h2 = height(v2)

while abs(h2-h1)>target:
    vp = (v1+v2)/2
    hp = height(vp)
    if h1*hp>0:
        v1 = vp
        h1 = hp
    else:
        v2 = vp
        h2 = hp

v = (v1+v2)/2
print("The required initial velocity is",v,"m/s")
```

One point to notice about this program is that the condition for the binary search to stop is a condition on the accuracy of the height of the ball at the final time $t = 10$, not a condition on the initial velocity. In most cases we care about matching the boundary conditions accurately, not calculating the initial conditions accurately.

If we run the program it prints the following:

```
The required initial velocity is 49.05 m/s
```

In principle, we could now take this value and use it to solve the differential equation once again, to compute the actual trajectory that the ball follows, verifying in the process that indeed it lands back on the ground at the allotted time $t = 10$.

8.6.2 THE RELAXATION METHOD

There is another method for solving boundary value problems that finds some use in physics, the *relaxation method*.[5] This method involves defining a shape for the entire solution, one that matches the boundary conditions but may not be a correct solution of the differential equation in the region between the boundaries. Then one successively modifies this shape to bring it closer and closer to a solution of the differential equation, while making sure that it continues to satisfy the boundary conditions.

In a way, the relaxation method is the opposite of the shooting method. The shooting method starts with a correct solution to the differential equation that may not match the boundary conditions and modifies it until it does. The relaxation method starts with a solution that matches the boundary conditions, but may not be a correct solution to the equation.

In fact the relaxation method is most often used not for solving boundary value problems for ordinary differential equations, but for partial differential equations, and so we will delay our discussion of it until the next chapter, which focuses on partial differential equations. Exercise 9.7 at the end of that chapter gives you an opportunity to apply the relaxation method to an ordinary differential equation problem. If you do that exercise you will see that the method is essentially the same for ordinary differential equations as it is for partial ones.

8.6.3 EIGENVALUE PROBLEMS

A special type of boundary value problem arises when the equation (or equations) being solved are linear and homogeneous, meaning that every term in the equation is linear in the dependent variable. A good example is the Schrödinger equation. For a single particle of mass m in one dimension, the time-independent Schrödinger equation is

$$-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2\psi}{\mathrm{d}x^2} + V(x)\psi(x) = E\psi(x), \tag{8.107}$$

where $\psi(x)$ is the wavefunction, $V(x)$ is the potential energy at position x, and E is the total energy of the particle, potential plus kinetic. Note how every term

[5]The relaxation method has the same name as the method for solving nonlinear equations introduced in Section 6.3.1, which is no coincidence. The two are in fact the same method. The only difference is that in Section 6.3.1 we applied the relaxation method to solutions for single variables, whereas we are solving for entire functions in the case of differential equations.

in the equation is linear in ψ.

Consider the problem of a particle in a square potential well with infinitely high walls. That is,

$$V(x) = \begin{cases} 0 & \text{for } 0 < x < L, \\ \infty & \text{elsewhere,} \end{cases} \tag{8.108}$$

where L is the width of the well. This problem is solvable analytically, but it is instructive to see how we would solve it numerically as well.

The probability of finding the particle in the region with $V(x) = \infty$ is zero, so the wavefunction has to go to zero at $x = 0$ and $x = L$. Thus this appears to be a standard boundary-value problem which we could solve, for instance, using the shooting method. Since the differential equation is second-order, we would start by turning it into two first-order ones, thus:

$$\frac{\mathrm{d}\psi}{\mathrm{d}x} = \phi, \qquad \frac{\mathrm{d}\phi}{\mathrm{d}x} = \frac{2m}{\hbar^2}\bigl[V(x) - E\bigr]\psi. \tag{8.109}$$

To calculate a solution we need two initial conditions, one for each of the variables ψ and ϕ. We know the value of ψ is zero at $x = 0$ but we don't know the value of $\phi = \mathrm{d}\psi/\mathrm{d}x$, so we guess an initial value then calculate the solution from $x = 0$ to $x = L$. Figure 8.12 shows an example calculated using fourth-order Runge–Kutta (the solid curve in the figure).

In principle the solution should equal zero again at $x = L$, but in this case it doesn't. Based on our experience with the shooting method in Section 8.6.1, we might guess that we can fix this problem by changing the initial condition on the derivative $\phi = \mathrm{d}\psi/\mathrm{d}x$. Using a root-finding method such as binary search we should be able to find the value of the derivative that makes the wavefunction exactly zero at $x = L$. Unfortunately in the present case this will not work.

To see why, consider what happens when we change the initial condition for our solution. If, for example, we double the initial value of $\mathrm{d}\psi/\mathrm{d}x$, that doubles the value of the wavefunction close to $x = 0$, as shown by the dashed line in Fig. 8.12. But the Schrödinger equation is a linear equation, meaning that if $\psi(x)$ is a solution then so also is $2\psi(x)$. Thus if we double the initial values of $\psi(x)$ the corresponding solution to the Schrödinger equation is just the same solution that we had before, but times two. This means that if the solution previously failed to pass through zero at $x = L$, it will still fail to pass through zero. And indeed no amount of adjustment of the initial condition will ever make the solution pass through zero. The initial condition only affects the overall magnitude of the solution but does not change its essential shape.

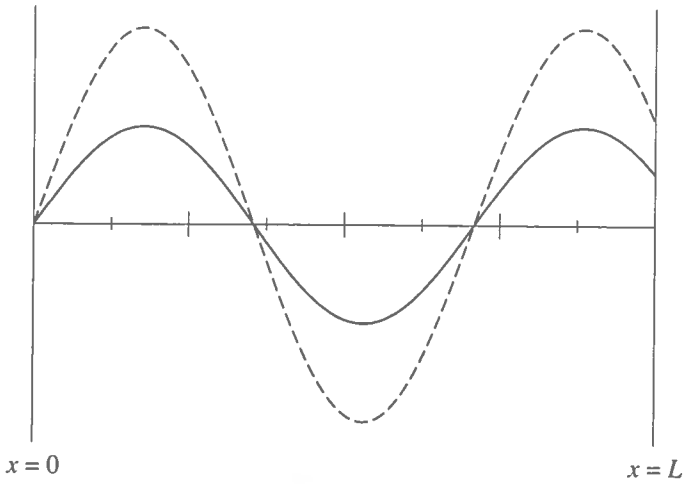


Figure 8.12: Solution of the Schrödinger equation in a square well. In attempting to solve the Schrödinger equation in a square well using the shooting method the initial value of the wavefunction at position $x = 0$ is known to be zero but we must guess a value for the initial gradient. The result will almost certainly be a wavefunction that fails to correctly return to zero at the far wall of the well, at $x = L$ (solid curve). Changing our guess for the initial slope does not help because the equation is linear, meaning the solution merely gets rescaled by a constant (dashed curve). If the solution did not previously pass through zero at $x = L$ then it still won't, and indeed will not for any choice of initial conditions.

The fundamental issue underlying our problem in this case is that there simply is no solution to the equation that is zero at both $x = 0$ and $x = L$. There is no solution that satisfies the boundary conditions. In fact, there are in this case solutions to the Schrödinger equation only for certain special values of the energy parameter E, the so-called allowed values or eigenvalues; for other values no solutions exist. It is precisely because of this phenomenon that the energies of quantum particles are quantized, meaning that there are discrete allowed energy levels and all other energies are forbidden.

So how do we find the allowed energies and solve the Schrödinger equation? One way to do it is to use a method akin to the shooting method, but instead of varying the initial conditions we vary the energy E. For a particular set of initial conditions, we vary E to find the value for which $\psi = 0$ at $x = L$. That value is an energy eigenstate of the system, an allowed quantum energy, and the corresponding solution to the Schrödinger equation is the wavefunction in that eigenstate. We can think of the solution to the Schrödinger equation

as giving us a function $f(E)$ equal to the value of the wavefunction at $x = L$ and we want to find the value of E that makes this function zero. That is, we want to find a root of the function. As with the shooting method, we can find that root using any of the methods we learned about in Section 6.3, such as binary search or the secant method.

And what about the unknown boundary condition on $\phi = \mathrm{d}\psi/\mathrm{d}x$? How is that to be fixed in this scheme? The answer is that it doesn't matter. Since, as we have said, the only effect of changing this boundary condition is to multiply the whole wavefunction by a constant, we can give it any value we like. The end result will be a solution for the wavefunction that is correct except for a possible overall multiplying factor. Traditionally this factor is fixed by requiring that the wavefunction be normalized so that $\int |\psi(x)|^2 \,\mathrm{d}x = 1$. If we need a wavefunction normalized in this fashion, then we can normalize it after the rest of the solution has been computed by calculating the value of the integral $\int |\psi(x)|^2 \,\mathrm{d}x$ using any integration method we like, and then dividing the wavefunction throughout by the square root of that value.

EXAMPLE 8.9: GROUND STATE ENERGY IN A SQUARE WELL

Let us calculate the ground state energy of an electron in a square potential well with infinitely high walls separated by a distance L equal to the Bohr radius $a_0 = 5.292 \times 10^{-11}$ m. Here's a program to do the calculation using the secant method:

File: squarewell.py

```
from numpy import array,arange

# Constants
m = 9.1094e-31      # Mass of electron
hbar = 1.0546e-34   # Planck's constant over 2*pi
e = 1.6022e-19      # Electron charge
L = 5.2918e-11      # Bohr radius
N = 1000
h = L/N

# Potential function
def V(x):
    return 0.0

def f(r,x,E):
    psi = r[0]
    phi = r[1]
```

```
    fpsi = phi
    fphi = (2*m/hbar**2)*(V(x)-E)*psi
    return array([fpsi,fphi],float)

# Calculate the wavefunction for a particular energy
def solve(E):
    psi = 0.0
    phi = 1.0
    r = array([psi,phi],float)

    for x in arange(0,L,h):
        k1 = h*f(r,x,E)
        k2 = h*f(r+0.5*k1,x+0.5*h,E)
        k3 = h*f(r+0.5*k2,x+0.5*h,E)
        k4 = h*f(r+k3,x+h,E)
        r += (k1+2*k2+2*k3+k4)/6

    return r[0]

# Main program to find the energy using the secant method
E1 = 0.0
E2 = e
psi2 = solve(E1)

target = e/1000
while abs(E1-E2)>target:
    psi1,psi2 = psi2,solve(E2)
    E1,E2 = E2,E2-psi2*(E2-E1)/(psi2-psi1)

print("E =",E2/e,"eV")
```

If we run the program, it prints the energy of the ground state thus:

```
E = 134.286371694 eV
```

which is indeed the correct answer. Note that the function `V(x)` does nothing in this program—since the potential everywhere in the well is zero, it plays no role in the calculation. However, by including the function in the program we make it easier to solve other, less trivial potential well problems. For instance, suppose the potential inside the well is not zero but varies as

$$V(x) = V_0 \frac{x}{L}\left(\frac{x}{L} - 1\right), \tag{8.110}$$

where $V_0 = 100\,\text{eV}$. It's a simple matter to solve for the ground-state energy of this problem also. We have only to change the function `V(x)`, thus:

```
V0 = 100*e
def V(x):
    return V0*(x/L)*(x/L-1)
```

Then we run the program again and this time find that the ground state energy is

```
E = 112.540107208 eV
```

Though the original square well problem is relatively easy to solve analytically, this second version of the problem with a varying potential would be much harder, and yet a solution is achieved easily using the computer.

Exercise 8.14: Quantum oscillators

Consider the one-dimensional, time-independent Schrödinger equation in a harmonic (i.e., quadratic) potential $V(x) = V_0x^2/a^2$, where V_0 and a are constants.

a) Write down the Schrödinger equation for this problem and convert it from a second-order equation to two first-order ones, as in Example 8.9. Write a program, or modify the one from Example 8.9, to find the energies of the ground state and the first two excited states for these equations when m is the electron mass, $V_0 = 50\,\text{eV}$, and $a = 10^{-11}\,\text{m}$. Note that in theory the wavefunction goes all the way out to $x = \pm\infty$, but you can get good answers by using a large but finite interval. Try using $x = -10a$ to $+10a$, with the wavefunction $\psi = 0$ at both boundaries. (In effect, you are putting the harmonic oscillator in a box with impenetrable walls.) The wavefunction is real everywhere, so you don't need to use complex variables, and you can use evenly spaced points for the solution—there is no need to use an adaptive method for this problem.

   The quantum harmonic oscillator is known to have energy states that are equally spaced. Check that this is true, to the precision of your calculation, for your answers. (Hint: The ground state has energy in the range 100 to 200 eV.)

b) Now modify your program to calculate the same three energies for the anharmonic oscillator with $V(x) = V_0x^4/a^4$, with the same parameter values.

c) Modify your program further to calculate the properly normalized wavefunctions of the anharmonic oscillator for the three states and make a plot of them, all on the same axes, as a function of x over a modest range near the origin—say $x = -5a$ to $x = 5a$.

   To normalize the wavefunctions you will have to calculate the value of the integral $\int_{-\infty}^{\infty} |\psi(x)|^2\,dx$ and then rescale ψ appropriately to ensure that the area

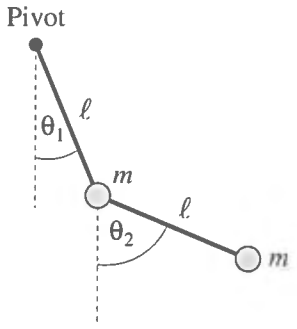
under the square of each of the wavefunctions is 1. Either the trapezoidal rule or Simpson's rule will give you a reasonable value for the integral. Note, however, that you may find a few very large values at the end of the array holding the wavefunction. Where do these large values come from? Are they real, or spurious?

One simple way to deal with the large values is to make use of the fact that the system is symmetric about its midpoint and calculate the integral of the wavefunction over only the left-hand half of the system, then double the result. This neatly misses out the large values.

The methods described in this section allow us to calculate solutions of the Schrödinger equation in one dimension, but the real world, of course, is three-dimensional. In three dimensions the Schrödinger equation becomes a partial differential equation, whose solution requires a different set of techniques. Partial differential equations are the topic of the next chapter.

FURTHER EXERCISES

8.15 The double pendulum: If you did Exercise 8.4 you will have created a program to calculate the movement of a nonlinear pendulum. Although it is nonlinear, the nonlinear pendulum's movement is nonetheless perfectly regular and periodic—there are no surprises. A *double pendulum*, on the other hand, is completely the opposite—chaotic and unpredictable. A double pendulum consists of a normal pendulum with another pendulum hanging from its end. For simplicity let us ignore friction, and assume that both pendulums have bobs of the same mass m and massless arms of the same length ℓ. Thus the setup looks like this:



The position of the arms at any moment in time is uniquely specified by the two angles θ_1 and θ_2. The equations of motion for the angles are most easily derived using the Lagrangian formalism, as follows.

The heights of the two bobs, measured from the level of the pivot are

$$h_1 = -\ell\cos\theta_1, \qquad h_2 = -\ell(\cos\theta_1 + \cos\theta_2),$$

so the potential energy of the system is

$$V = mgh_1 + mgh_2 = -mg\ell(2\cos\theta_1 + \cos\theta_2),$$

where g is the acceleration due to gravity. (The potential energy is negative because we have chosen to measure it downwards from the level of the pivot.)

The velocities of the two bobs are given by

$$v_1 = \ell\dot\theta_1, \qquad v_2^2 = \ell^2\bigl[\dot\theta_1^2 + \dot\theta_2^2 + 2\dot\theta_1\dot\theta_2\cos(\theta_1 - \theta_2)\bigr],$$

where $\dot\theta$ means the derivative of θ with respect to time t. (If you don't see where the second velocity equation comes from, it's a good exercise to derive it for yourself from the geometry of the pendulum.) Then the total kinetic energy is

$$T = \tfrac12 mv_1^2 + \tfrac12 mv_2^2 = m\ell^2\bigl[\dot\theta_1^2 + \tfrac12\dot\theta_2^2 + \dot\theta_1\dot\theta_2\cos(\theta_1 - \theta_2)\bigr],$$

and the Lagrangian of the system is

$$\mathscr{L} = T - V = m\ell^2\bigl[\dot\theta_1^2 + \tfrac12\dot\theta_2^2 + \dot\theta_1\dot\theta_2\cos(\theta_1 - \theta_2)\bigr] + mg\ell(2\cos\theta_1 + \cos\theta_2).$$

Then the equations of motion are given by the Euler–Lagrange equations

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial\mathscr{L}}{\partial\dot\theta_1}\right) = \frac{\partial\mathscr{L}}{\partial\theta_1}, \qquad \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial\mathscr{L}}{\partial\dot\theta_2}\right) = \frac{\partial\mathscr{L}}{\partial\theta_2},$$

which in this case give

$$2\ddot\theta_1 + \ddot\theta_2\cos(\theta_1 - \theta_2) + \dot\theta_2^2\sin(\theta_1 - \theta_2) + 2\frac{g}{\ell}\sin\theta_1 = 0,$$
$$\ddot\theta_2 + \ddot\theta_1\cos(\theta_1 - \theta_2) - \dot\theta_1^2\sin(\theta_1 - \theta_2) + \frac{g}{\ell}\sin\theta_2 = 0,$$

where the mass m has canceled out.

These are second-order equations, but we can convert them into first-order ones by the usual method, defining two new variables, ω_1 and ω_2, thus:

$$\dot\theta_1 = \omega_1, \qquad \dot\theta_2 = \omega_2.$$

In terms of these variables our equations of motion become

$$2\dot\omega_1 + \dot\omega_2\cos(\theta_1 - \theta_2) + \omega_2^2\sin(\theta_1 - \theta_2) + 2\frac{g}{\ell}\sin\theta_1 = 0,$$
$$\dot\omega_2 + \dot\omega_1\cos(\theta_1 - \theta_2) - \omega_1^2\sin(\theta_1 - \theta_2) + \frac{g}{\ell}\sin\theta_2 = 0.$$

Finally we have to rearrange these into the standard form of Eq. (8.29) with a single derivative on the left-hand side of each one, which gives

$$\dot{\omega}_1 = -\frac{\omega_1^2 \sin(2\theta_1 - 2\theta_2) + 2\omega_2^2 \sin(\theta_1 - \theta_2) + (g/\ell)[\sin(\theta_1 - 2\theta_2) + 3\sin\theta_1]}{3 - \cos(2\theta_1 - 2\theta_2)},$$

$$\dot{\omega}_2 = \frac{4\omega_1^2 \sin(\theta_1 - \theta_2) + \omega_2^2 \sin(2\theta_1 - 2\theta_2) + 2(g/\ell)[\sin(2\theta_1 - \theta_2) - \sin\theta_2]}{3 - \cos(2\theta_1 - 2\theta_2)}.$$

(This last step is quite tricky and involves some trigonometric identities. If you have not seen the derivation before, you may find it useful to go through it for yourself.)

These two equations, along with the equations $\dot{\theta}_1 = \omega_1$ and $\dot{\theta}_2 = \omega_2$, give us four first-order equations which between them define the motion of the double pendulum.

a) Derive an expression for the total energy $E = T + V$ of the system in terms of the variables θ_1, θ_2, ω_1, and ω_2, plus the constants g, ℓ, and m.

b) Write a program using the fourth-order Runge–Kutta method to solve the equations of motion for the case where $\ell = 40\,\text{cm}$, with the initial conditions $\theta_1 = \theta_2 = 90°$ and $\omega_1 = \omega_2 = 0$. Use your program to calculate the total energy of the system assuming that the mass of the bobs is 1 kg each, and make a graph of energy as a function of time from $t = 0$ to $t = 100$ seconds.

Because of energy conservation, the total energy should be constant over time (actually it should be zero for these particular initial conditions), but you will find that it is not perfectly constant because of the approximate nature of the solution of the differential equations. Choose a suitable value of the step size h to ensure that the variation in energy is less than 10^{-5} joules over the course of the calculation.

c) Make a copy of your program and modify the copy to create a second program that does not produce a graph, but instead makes an animation of the motion of the double pendulum over time. At a minimum, the animation should show the two arms and the two bobs.

Hint: As in Exercise 8.4 you will probably find the function `rate` useful in order to make your program run at a steady speed. You will probably also find that the value of h needed to get the required accuracy in your solution gives a frame-rate much faster than any that can reasonably be displayed in your animation, so you won't be able to display every time-step of the calculation in the animation. Instead you will have to arrange the program so that it updates the animation only once every several Runge–Kutta steps.

8.16 The three-body problem: If you mastered Exercise 8.10 on cometary orbits, here's a more challenging problem in celestial mechanics—and a classic in the field—the *three-body problem*.

Three stars, in otherwise empty space, are initially at rest, with the following masses and positions, in arbitrary units:

	Mass	x	y
Star 1	150	3	1
Star 2	200	−1	−2
Star 3	250	−1	1

(All the z coordinates are zero, so the three stars lie in the xy plane.)

a) Show that the equation of motion governing the position $\mathbf{r}_1$ of the first star is

$$\frac{d^2\mathbf{r}_1}{dt^2} = Gm_2\frac{\mathbf{r}_2 - \mathbf{r}_1}{|\mathbf{r}_2 - \mathbf{r}_1|^3} + Gm_3\frac{\mathbf{r}_3 - \mathbf{r}_1}{|\mathbf{r}_3 - \mathbf{r}_1|^3}$$

and derive two similar equations for the positions $\mathbf{r}_2$ and $\mathbf{r}_3$ of the other two stars. Then convert the three second-order equations into six equivalent first-order equations, using the techniques you have learned.

b) Working in units where $G = 1$, write a program to solve your equations and hence calculate the motion of the stars from $t = 0$ to $t = 2$. Make a plot showing the trails of all three stars (i.e., a graph of y against x for each star).

c) Modify your program to make an animation of the motion on the screen from $t = 0$ to $t = 10$. You may wish to make the three stars different sizes or colors (or both) so that you can tell which is which.

To do this calculation properly you will need to use an adaptive step size method, for the same reasons as in Exercise 8.10—the stars move very rapidly when they are close together and very slowly when they are far apart. An adaptive method is the only way to get the accuracy you need in the fast-moving parts of the motion without wasting hours uselessly calculating the slow parts with a tiny step size. Construct your program so that it introduces an error of no more than 10^{-3} in the position of any star per unit time.

Creating an animation with an adaptive step size can be challenging, since the steps do not all correspond to the same amount of real time. The simplest thing to do is just to ignore the varying step sizes and make an animation as if they were all equal, updating the positions of the stars on the screen at every step or every several steps. This will give you a reasonable visualization of the motion, but it will look a little odd because the stars will slow down, rather than speed up, as they come close together, because the adaptive calculation will automatically take more steps in this region.

A better solution is to vary the frame-rate of your animation so that the frames run proportionally faster when h is smaller, meaning that the frame-rate needs to be equal to C/h for some constant C. You can achieve this by using the `rate` function from the `visual` package to set a different frame-rate on each step, equal to C/h. If you do this, it's a good idea to not let the value of h grow too large, or the animation will make some large jumps that look uneven on the screen. Insert extra program lines to ensure that h never exceeds a value h_{max} that you choose. Values for the constants of around $C = 0.1$ and $h_{\text{max}} = 10^{-3}$ seem to give reasonable results.

8.17 Cometary orbits and the Bulirsch–Stoer method: Repeat the calculation of the cometary orbit in Exercise 8.10 (page 361) using the adaptive Bulirsch–Stoer method of

Section 8.5.6 to calculate a solution accurate to $\delta = 1$ kilometer per year in the position of the comet. Calculate the solution from $t = 0$ to $t = 2 \times 10^9$ s, initially using just a single time interval of size $H = 2 \times 10^9$ s and allowing a maximum of $n = 8$ modified midpoint steps before dividing the interval in half and trying again. Then these intervals may be subdivided again, as described in Section 8.5.6, as many times as necessary until the method converges in eight steps or less in each interval.

Make a plot of the orbit (i.e., a plot of y against x) and have your program add dots to the trajectory to show where the ends of the time intervals lie. You should see the time intervals getting shorter in the part of the trajectory close to the Sun, where the comet is moving rapidly.

Hint: The simplest way to do this calculation is to make use of recursion, the ability of a Python function to call itself. (If you're not familiar with the idea of recursion you might like to look at Exercise 2.13 on page 83 before doing this exercise.) Write a user-defined function called, say, `step(r,t,H)` that takes as arguments the position vector $\mathbf{r} = (x,y)$ at a starting time t and an interval length H, and returns the new value of $\mathbf{r}$ at time $t + H$. This function should perform the modified midpoint/Richardson extrapolation calculation described in Section 8.5.5 until either the calculation converges to the required accuracy or you reach the maximum number $n = 8$ of modified midpoint steps. If it fails to converge in eight steps, have your function call itself, twice, to calculate separately the solution for the first then the second half of the interval from t to $t + H$, something like this:

```
r1 = step(r,t,H/2)
r2 = step(r1,t+H/2,H/2)
```

(Then *these* functions can call themselves, and so forth, subdividing the interval as many times as necessary to reach the required accuracy.)

8.18 Oscillating chemical reactions: The *Belousov–Zhabotinsky reaction* is a chemical oscillator, a cocktail of chemicals which, when heated, undergoes a series of reactions that cause the chemical concentrations in the mixture to oscillate between two extremes. You can add an indicator dye to the reaction which changes color depending on the concentrations and watch the mixture switch back and forth between two different colors for as long as you go on heating the mixture.

Physicist Ilya Prigogine formulated a mathematical model of this type of chemical oscillator, which he called the "Brusselator" after his home town of Brussels. The equations for the Brusselator are

$$\frac{dx}{dt} = 1 - (b+1)x + ax^2y, \qquad \frac{dy}{dt} = bx - ax^2y.$$

Here x and y represent concentrations of chemicals and a and b are positive constants.

Write a program to solve these equations for the case $a = 1$, $b = 3$ with initial conditions $x = y = 0$, to an accuracy of at least $\delta = 10^{-10}$ per unit time in both x and y, using the adaptive Bulirsch–Stoer method described in Section 8.5.6. Calculate

a solution from $t = 0$ to $t = 20$, initially using a single time interval of size $H = 20$. Allow a maximum of $n = 8$ modified midpoint steps in an interval before you divide in half and try again.

Make a plot of your solutions for x and y as a function of time, both on the same graph, and have your program add dots to the curves to show where the boundaries of the time intervals lie. You should find that the points are significantly closer together in parts of the solution where the variables are changing rapidly.

Hint: The simplest way to perform the calculation is to make use of recursion, as described in Exercise 8.17.