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

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{2x}{t}. \tag{8.1}$$

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

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{2x}{t} + \frac{3x^2}{t^3}. \tag{8.2}$$

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

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x,t), \tag{8.3}$$

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

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x,y). \tag{8.4}$$

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:

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

where we have used Eq. (8.3) and $\mathrm{O}(h^2)$ 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()
```

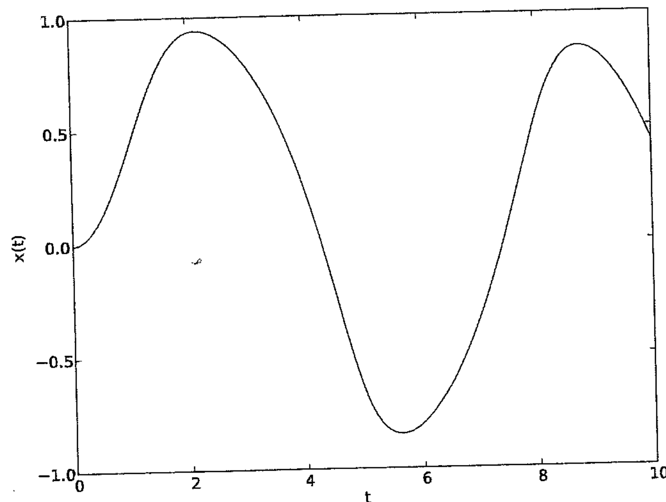


Figure 8.1: Numerical solution of an ordinary differential equation. A solution to Eq. (8.7) from $x = 0$ to $x = 10$, calculated using Euler's method.

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

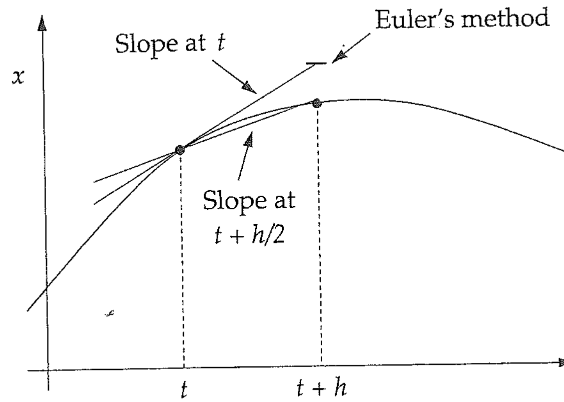


Figure 8.2: Euler's method and the second-order Runge–Kutta method. Euler's method is equivalent to taking the slope $\mathrm{d}x/\mathrm{d}t$ at time t and extrapolating it into the future to time $t + h$. A better approximation is to perform the extrapolation using the slope at time $t + \frac{1}{2}h$.

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 $\mathrm{d}x/\mathrm{d}t = f(x,t)$ 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 $x(t+\frac{1}{2}h)$:

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

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.

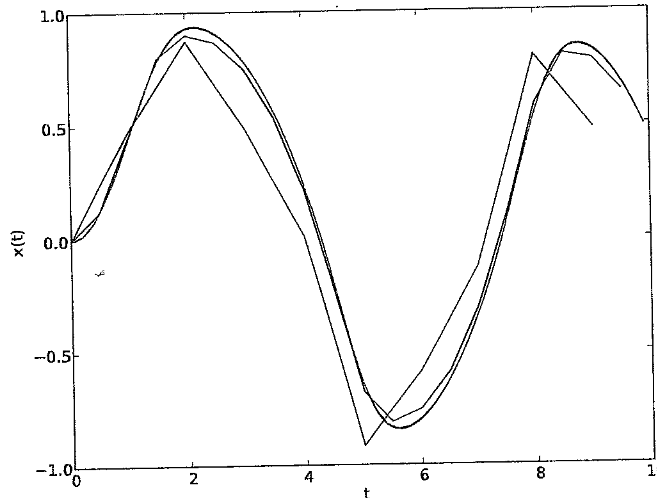


Figure 8.3: Solutions calculated with the second-order Runge–Kutta method. Solutions to Eq. (8.7) calculated using the second-order Runge–Kutta method with $N = 10$, 20, 50, and 100 steps.

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
```

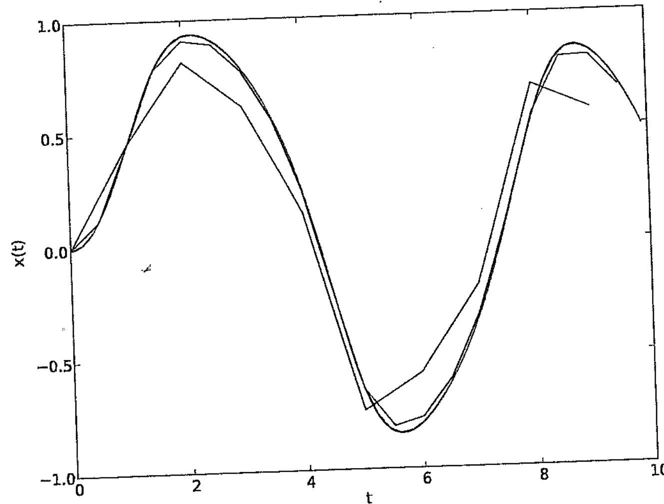


Figure 8.4: Solutions calculated with the fourth-order Runge–Kutta method. Solutions to Eq. (8.7) calculated using the fourth-order Runge–Kutta method with $N = 10$, 20, 50, and 100 steps.

```
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()
```

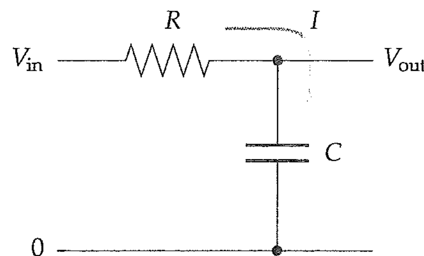
Again we run the program repeatedly with $N = 10, 20, 50$, and 100. Figure 8.4 shows the results. Now we see that, remarkably, even the solution with 20 points is close to the final converged solution for the equation. With only 20 points we get quite a jagged curve—20 points is not enough to make the 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

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

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

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