§8 Ordinary Differential Equations

Many of the problems encountered in physics involve ordinary differential equations for which some initial values or boundary values are specified. For example, both the equation of motion of a harmonic oscillator and the equation governing the behavior of a LCR electrical circuit are second-order ordinary differential equations. Thus seeking the numerical solution of ordinary differential equations is an important topic in computational physics. In this chapter, we study some common numerical methods for solving ordinary differential equations.

8.1 First-order Differential Equations

The simplest type of ordinary differential equation is a first-order equation with a single variable such as

$$\frac{dx}{dt} = \frac{2x}{t}$$

Obviously, this equation can be solved analytically by hand using the separation of variables. There is no need to use a computer to find the solution in this case. But suppose you instead have

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

Now the equation is no longer separable. In addition, 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 single-variable ordinary differential equation is:

$$\frac{dx}{dt} = f(x,t) \tag{8.1}$$

where $f(x,t)$ is some function we specify. In the differential equation $dx/dt = 2x/t + 3x^2/t^3$, the independent variable is denoted as t which often refers to time in physics problems. Of course there are other possibilities. We could have just as well written our equation as

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

From now on, we will stick with time t for the independent variable. But we should bear in mind that there are plenty of examples where the independent variable is not time.

To calculate a full solution to Eq. (8.1), we also require an initial condition or boundary condition. That is to say, we have to specify the value of x at one particular value of t, for instance $t = 0$. In all the problems we will tackle in this chapter, we will assume that both the equation and its initial or boundary condition are given.

Suppose we are given an equation in the form of Eq. (8.1) 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 series expansion:

$$x(t+h) = x(t) + h\frac{dx}{dt} + \frac{1}{2}h^2\frac{d^2x}{dt^2} + \ldots = x(t) + hf(x,t) + O(h^2) \tag{8.2}$$

where we have used Eq. (8.1) 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 h^2 and higher in the above expression to obtain

$$x(t+h) \approx x(t) + hf(x,t) \tag{8.3}$$

If we know the value of x at time t, then we can use this equation to calculate the value a short time later. Next we can just repeat the procedure to calculate x at another interval h after that and so forth. Thus we compute x at a succession of evenly spaced points for as long as we want. We would obtain $x(t)$ only at a finite set of points from this calculation instead for all values of t. However, if h is small enough, then we can get a pretty good picture of what the solution to the equation looks like. As we saw in Section 4.2, we can make a convincing curve by approximating it with a set of closely spaced points.

For example, we might be given a differential equation for x with an initial condition at $t = a$ and asked to make a graph of $x(t)$ for all values of t from a to b. To do this, we first divide the interval from a to b into steps of size h. Then we use Eq. (8.3) repeatedly to calculate $x(t)$ and plot the results. This method for solving differential equations is known as **Euler's method** after its inventor Leonard Euler.

◇◇◇

Example 8.1

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

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

with the initial condition $x = 0$ at $t = 0$. Here is the Python program `euler.py` doing the calculation from $t = 0$ to $t = 10$ in 1000 steps and then plotting the result.

```
# euler.py
# This program solves the ODE dx/dt = −x^3 + sin t with x(0) = 0
# using Euler's method and plots the results.
# Last update on 4 Dec 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

def f(x,t):
    """ Function to compute the function f(x, t) = −x^3 + sin t """
    return −x**3 + np.sin(t)

# Use Euler method to solve the ODE
a, b = 0.0, 10.0    # Start and end of the interval
N = 1000            # Number of steps
h = (b−a)/N         # Size of a single step
x = 0.0             # Initial condition
tpoints = np.linspace(a,b,N+1)
xpoints = [x]
for t in tpoints[:−1]:
    x += h*f(x,t)
    xpoints.append(x)

# Plot the solution of the ODE
fig, ax = plt.subplots()
ax.plot(tpoints,xpoints)
ax.set_xlabel("t")
ax.set_ylabel("x(t)")
plt.show()
```

Running this program generates the graph as shown in Figure 8.1 which turns out to be a pretty good approximation to the shape of the true solution to the equation. Euler's method does a good job in this case.

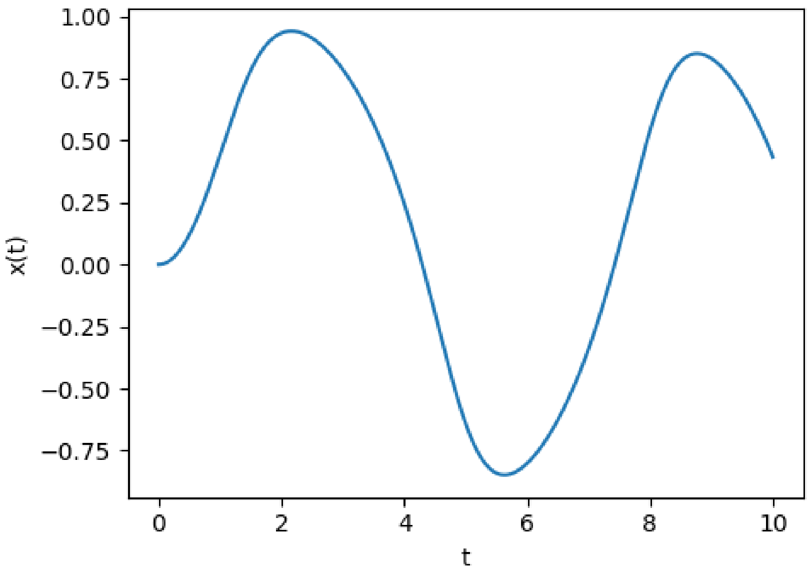

Figure 8.1: A solution to $dx/dt = -x^3 + \sin t$ with $x(0) = 0$ from $t = 0$ to $t = 10$ computed by Euler's method with 1000 steps.

◇◇

In general, Euler's method is not bad. It gives reasonable answers in many cases. However, we never use Euler's method in practice. It is because there is a better method called **Runge-Kutta method** which requires a very little extra work to program, much more accurate, and runs just as fast and often faster. Before discussing the Runge-Kutta method, let us first take a closer look at Euler's method to understand why it's not ideal.

Euler's method only gives approximate solutions. The approximation arises because we neglected h^2 and all higher-order terms in Eq. (8.2). The size of the h^2 term is $(1/2)h^2 d^2x/dt^2$, which tells us the error introduced on a single step of the method to leading order. 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. 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 value of x 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 of each step. Thus this error is equal to

$$\begin{aligned}\sum_{k=0}^{N-1} \frac{1}{2}h^2 \left.\frac{d^2x}{dt^2}\right|_{x=x_k,t=t_k} = \frac{1}{2}h \sum_{k=0}^{N-1} h \left.\frac{d^2x}{dt^2}\right|_{x=x_k,t=t_k} &\approx \frac{1}{2}h \int_a^b \frac{\partial f}{\partial t} dt \\ &= \frac{1}{2}h[f(x(b), b) - f(x(a), a)]\end{aligned}$$

Here we have approximated the sum by an integral which is a good approximation for small h.

Notice that the final expression for the total error is linear in h although the individual errors are of order h^2. It implies that the total error goes down by a factor of two when h is halved. In principle, this allows us to make the error as small as we like. But we also increases the number of steps $N = (b - a)/h$ when we make h smaller and hence the calculation will take proportionately longer — a calculation that is twice as accurate will take twice as long. Perhaps this doesn't sound too bad. If this is the way it had to be, then we could live with it. But it doesn't have to be that way. The Runge-Kutta method does much better.

You might think that the way to improve on Euler's method would be to use the Taylor series expansion in Eq. (8.2) again, but keep terms to higher order. For example, in addition to the order h term, we could keep the order h^2 term which is equal to

$$\frac{1}{2}h^2\frac{d^2x}{dt^2} = \frac{1}{2}h^2\frac{\partial f}{\partial t}.$$

This would give us a more accurate expression for $x(t + h)$. This approach might work in some cases, but in a lot of cases it would not. It requires us to know the derivative $\partial f/\partial t$ which we can compute only if we have an explicit expression for f. We often have no such expression because, for instance, the function f is calculated as the output of another computer program or function and thus 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 actually a set of methods — there are many of them of different orders, which gives results of varying degree of accuracy. In fact, Euler method is technically the first-order Runge-Kutta method. Let us look at the next method in the series, the **second-order Runge-Kutta method** which is also sometimes called the **midpoint method**.

Euler's method can be represented in graphical fashion as depicted in Figure 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 we can calculate the slope at time t given the value of x at that point. 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 Figure 8.2. 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 $x(t+h)$.

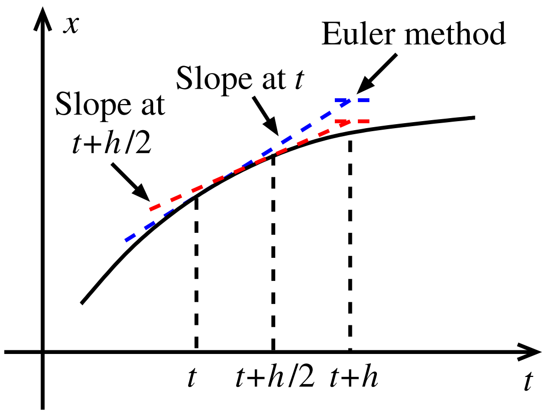

Figure 8.2: Euler's method is equivalent to taking the slope dx/dt at time t and extrapolating it to time $t+h$. A better approximation is to perform the extrapolation using the slope at time $t+h/2$.

Now suppose we do the same calculation but instead use the slope at the midpoint $t+h/2$ to do our extrapolation as shown in Figure 8.2. If we extrapolate using this slope, then we get a different estimate of $x(t+h)$ which is usually better than Euler's method. This is the basis for the second-order Runge-Kutta method.

In mathematical terms, the method involves a Taylor series expansion around $t+h/2$ to get the value of $x(t+h)$:

$$x(t+h) = x\left(t+\frac{1}{2}h\right) + \frac{1}{2}h\left.\frac{dx}{dt}\right|_{t+\frac{1}{2}h} + \frac{1}{8}h^2\left.\frac{d^2x}{dt^2}\right|_{t+\frac{1}{2}h} + O(h^3)$$

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

$$x(t) = x\left(t+\frac{1}{2}h\right) - \frac{1}{2}h\left.\frac{dx}{dt}\right|_{t+\frac{1}{2}h} + \frac{1}{8}h^2\left.\frac{d^2x}{dt^2}\right|_{t+\frac{1}{2}h} + O(h^3)$$

Subtracting the second expression from the first one and rearranging then yields:

$$x(t+h) = x(t) + h\left.\frac{dx}{dt}\right|_{t+\frac{1}{2}h} + O(h^3) = x(t) + hf\left(x\left(t+\frac{1}{2}h\right), t+\frac{1}{2}h\right) + O(h^3) \quad (8.4)$$

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

Though it looks promising, there is a problem with this approach: Eq. (8.4) requires a knowledge of $x(t+h/2)$ which we don't have. We only know the value at $x(t)$. We overcome this by approximating $x(t+h/2)$ using Euler's method $x(t+h/2)=x(t)+(h/2)f(x,t)$ and then substituting into the above equation. The complete calculation for a single step can be written like this:

$$k_1 = hf(x,t), \tag{8.5}$$

$$k_2 = hf\left(x+\frac{1}{2}k_1, t+\frac{1}{2}h\right), \tag{8.6}$$

$$x(t+h) \approx x(t) + k_2 \tag{8.7}$$

Notice how Eq. (8.5) gives a value of k_1 which gives us our estimate of $x(t+h/2)$ when inserted into Eq. (8.6). Then the resulting value of k_2 gives us the final Runge-Kutta estimate for $x(t+h)$ after inserted into Eq. (8.7).

Equations (8.5) to (8.7) are the equations for the second-order Runge-Kutta method. As with the methods for performing integrals studied in Chapter 6, a "second-order" method here is a method *accurate* to order h^2, meaning that the error is of order h^3. By contrast, Euler's method is a first-order method with an error of order h^2. Beware that these designations refer to just a single step of each method. As discussed before, real calculations involve doing many steps one after another with errors that accumulate. So 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. Thus we could make h bigger — and so take fewer steps — while getting the same level of accuracy as Euler's method. This creates a program that achieves the same result as Euler's method with a much shorter runtime.

However, we are not entirely done with our derivation yet. Since we don't have an exact value of $x(t+h/2)$ and had to approximate it using Euler's method. So there is an extra source of error in Eq. (8.4) coming from this second approximation, in addition to the $O(h^3)$ error that we have already acknowledged. How do we know that this second error isn't larger than $O(h^3)$ and does not make the accuracy of our calculation worse?

We can show that in fact this is not a problem by expanding the quantity $f(x+k_1/2, t+h/2)$ in Eq. (8.6) in its first argument only around $x(t+h/2)$:

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

But from Eq. (8.2), we have

$$x\left(t+\frac{1}{2}h\right) = x(t) + \frac{1}{2}hf(x,t) + O(h^2) = x(t) + \frac{1}{2}k_1 + O(h^2).$$

So $x(t) + k_1/2 - x(t+h/2) = O(h^2)$ and

$$f\left(x(t) + \frac{1}{2}k_1, t + \frac{1}{2}h\right) = f\left(x\left(t+\frac{1}{2}h\right), t + \frac{1}{2}h\right) + O(h^2).$$

This means that Eq. (8.6) gives $k_2 = hf(x(t+h/2), t+h/2) + O(h^3)$ and hence there's no problem. Our Euler's approximation for $x(t+h/2)$ does introduce an additional error into the calculation, but the error goes like h^3 and thus our second-order Runge-Kutta method is still accurate to $O(h^3)$ overall.

◇◇

Example 8.2

Let us use the second-order Runge-Kutta method to solve the differential equation in Example 8.1 again. Below is the Python program `rk2.py` which is a minor modification of our program for Euler's method.

```python
# rk2.py
# This program solves the ODE dx/dt = -x^3 + sin t with x(0) = 0
# using the second-order Runge-Kutta method and plots the results.
# Last update on 4 Dec 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

def f(x,t):
    """ Function to compute the function f(x, t) = -x^3 + sin t """
    return -x**3 + np.sin(t)

# Use the 2nd-order Runge-Kutta method to solve the ODE for different
# number of steps N and plot the solutions on the same graph
a, b = 0.0, 10.0     # Start and end of the interval
fig, ax = plt.subplots()
for N in [10, 20, 50, 100]:
    h = (b-a)/N      # Size of a single step
    x = 0.0          # Initial condition
    tpoints = np.linspace(a,b,N+1)
    xpoints = [x]
    for t in tpoints[:−1]:
        k1 = h*f(x,t)
        k2 = h*f(x+0.5*k1,t+0.5*h)
        x += k2
        xpoints.append(x)
    l = "N = {}".format(N)
    ax.plot(tpoints,xpoints,label=l)
ax.set_xlabel("t")
ax.set_ylabel("x(t)")
ax.legend()
plt.show()
```

Figure 8.3 shows the output of this program. We can see that the solutions for number of steps $N = 10$ and $N = 20$ are quite poor. But the solutions for $N = 50$ and $N = 100$ look very similar, indicating that the method has converged to a result close to the true solution. Indeed a comparison with Figure 8.1 shows good agreement with our Euler's method solution which used 1000 steps.

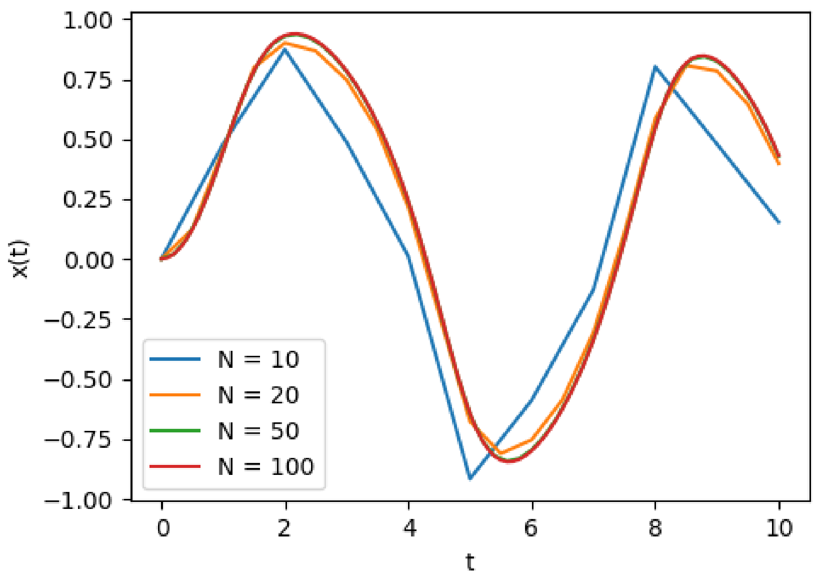

Figure 8.3: A solution to $dx/dt = -x^3 + \sin t$ with $x(0) = 0$ from $t = 0$ to $t = 10$ computed by the second-order Runge-Kutta method with different number of steps N.

◇◇

We can take this approach further. By performing Taylor's series 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 in our expressions. And this yields more and more accurate rules for solving differential equations. The downside is that the equations become more complicated as we go to higher order. However, many people feel 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:

$$k_1 = hf(x,t), \tag{8.8}$$

$$k_2 = hf\left(x + \frac{1}{2}k_1, t + \frac{1}{2}h\right), \tag{8.9}$$

$$k_3 = hf\left(x + \frac{1}{2}k_2, t + \frac{1}{2}h\right), \tag{8.10}$$

$$k_4 = hf(x + k_3, t + h), \tag{8.11}$$

$$x(t+h) \approx x(t) + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{8.12}$$

This is the **fourth-order Runge-Kutta method** which is by far the most common method for 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 derivation here as 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 methods for steps of the same size. In practice this can make the fourth-order method as much as million times more accurate than Euler's method. In fact, the fourth-order method is significantly better even than the second-order method. 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 much 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 a computer. It's easy to program and gives excellent results. It's the workhorse of differential equation solvers and one of the best known computer algorithms of any kind.

◇◇

Example 8.3

Let us solve the differential equation in Example 8.1 once more using the fourth-order Runge-Kutta method. Below is the Python program `rk4.py` which is again only a minor modification of our previous program.

```
# rk4.py
# This program solves the ODE dx/dt = -x^3 + sin t with x(0) = 0
# using the fourth-order Runge-Kutta method and plots the results.
# Last update on 4 Dec 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

def f(x,t):
    """ Function to compute the function f(x, t) = -x^3 + sin t """
    return -x**3 + np.sin(t)

# Use the 4th-order Runge-Kutta method to solve the ODE for different
# number of steps N and plot the solutions on the same graph
a, b = 0.0, 10.0    # Start and end of the interval
fig, ax = plt.subplots()
for N in [10, 20, 50, 100]:
    h = (b-a)/N     # Size of a single step
    x = 0.0         # Initial condition
    tpoints = np.linspace(a,b,N+1)
    xpoints = [x]
    for t in tpoints[:-1]:
        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
        xpoints.append(x)
    l = "N = {}".format(N)
    ax.plot(tpoints,xpoints,label=l)
ax.set_xlabel("t")
ax.set_ylabel("x(t)")
ax.legend()
plt.show()
```

Figure 8.4 shows the output of this program. Now we can see that even the solution with number of steps $N = 20$ is close to the final converged solution of the equation. We get a jagged curve with only 20 steps, but the points nonetheless lie close to the final solution of the equation. With only 20 steps, the fourth-order Runge-Kutta method has calculated a solution almost as accurate as Euler's method with 1000 steps.

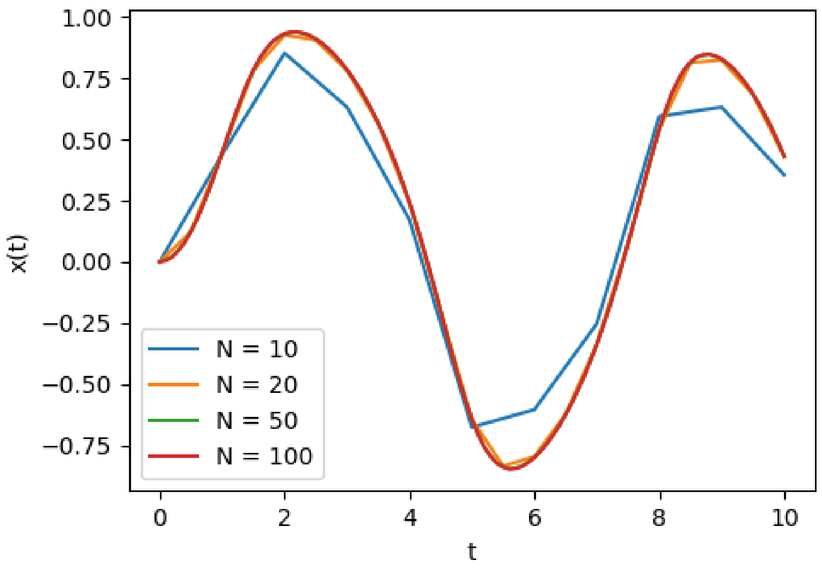

Figure 8.4: A solution to $dx/dt = -x^3 + \sin t$ with $x(0) = 0$ from $t = 0$ to $t = 10$ computed by the fourth-order Runge-Kutta method with different number of steps N.

◇◇

One minor downside of the fourth-order Runge-Kutta method, and indeed of all Runge-Kutta methods, is that if you get the equation wrong, it may not be obvious in the solution they produce. For example, if you miss one of the factors of $1/2$ or 2, or have a minus sign when you should use 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 Equations (8.8) to (8.12), then you will probably get a solution about as accurate as Euler's method which is much worse. It means that you must be careful when writing programs that use the Runge-Kutta method. You must check your code in detail to make sure that all the equations are exactly correct. If you make a mistake, then 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 making even a small error in the program will likely produce ridiculous results that are obviously wrong and so the error is relatively easy to spot.

8.2 Simultaneous Differential Equations

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 to say, 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,

$$\frac{dx}{dt} = xy - x, \qquad \frac{dy}{dt} = y - xy + \sin^3 \omega t$$

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

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

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 in vector notation as

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

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

Simultaneous differential equations are often a lot harder to solve analytically than single equations. However, they are not much more difficult than single-variable case when they are solved using computational methods. For instance, we can Taylor expand the vector $\mathbf{r}$ to obtain

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

Dropping the terms of order h^2 and higher yields Euler's method for multivariable case:

$$\mathbf{r}(t+h) \approx \mathbf{r}(t) + h\mathbf{f}(\mathbf{r}, t).$$

The Taylor series expansions used to derive the Runge-Kutta rules can be also generalized directly to the multivariable case. In particular, the multivariable version of the fourth-order Runge-Kutta method is an obvious vector generalization of the single-variable case:

$$\mathbf{k}_1 = h\mathbf{f}(\mathbf{r}, t), \tag{8.13}$$

$$\mathbf{k}_2 = h\mathbf{f}\left(\mathbf{r} + \frac{1}{2}\mathbf{k}_1, t + \frac{1}{2}h\right), \tag{8.14}$$

$$\mathbf{k}_3 = h\mathbf{f}\left(\mathbf{r} + \frac{1}{2}\mathbf{k}_2, t + \frac{1}{2}h\right), \tag{8.15}$$

$$\mathbf{k}_4 = h\mathbf{f}(\mathbf{r} + \mathbf{k}_3, t + h), \tag{8.16}$$

$$\mathbf{r}(t + h) \approx \mathbf{r}(t) + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \tag{8.17}$$

Equations (8.13) to (8.17) can be conveniently translated into Python code using arrays to represent the vectors. Since Python allows arithmetic to be performed with vectors directly and vectors to be both the argument and the output of functions, the code is just slightly more complicated than that for the single-variable case.

◇◇

Example 8.4

Let us calculate a solution to the simultaneous differential equations:

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

with the initial conditions $x = y = 1$ at $t = 0$ using the fourth-order Runge-Kutta method. Here is the Python program `odesim.py` which is again based on a slight modification of our earlier programs. The output of this program is depicted in Figure 8.5.

```python
# odesim.py
# This program solves the simultaneous ODEs dx/dt = xy - x and dy/dt
# = y - xy - sin^2 t with x(0) = y(0) = 1 using the fourth-order
# Runge-Kutta method and plots the results.
# Last update on 4 Dec 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

def f(r,t):
    """ Function to compute the function f(r, t) = (fx(r, t),
        fy(r, t)) where x = r[0] and y = r[1] """
    fx = r[0]*r[1] - r[0]
    fy = r[1] - r[0]*r[1] + np.sin(t)**2
    return np.array([fx,fy])

# Use the 4th-order Runge-Kutta method to solve the simultaneous ODEs
a, b = 0.0, 10.0          # Start and end of the interval
```

```
N = 1000                   # Number of steps
h = (b-a)/N                # Size of a single step
r = np.array([1.0,1.0]) # Initial conditions
tpoints = np.linspace(a,b,N+1)
xpoints, ypoints = [r[0]], [r[1]]
for t in tpoints[:-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
    xpoints.append(r[0])
    ypoints.append(r[1])

# Plot the solution of the simultaneous ODEs
fig, ax = plt.subplots()
ax.plot(tpoints,xpoints,label="x(t)")
ax.plot(tpoints,ypoints,label="y(t)")
ax.set_xlabel("t")
ax.legend()
plt.show()
```

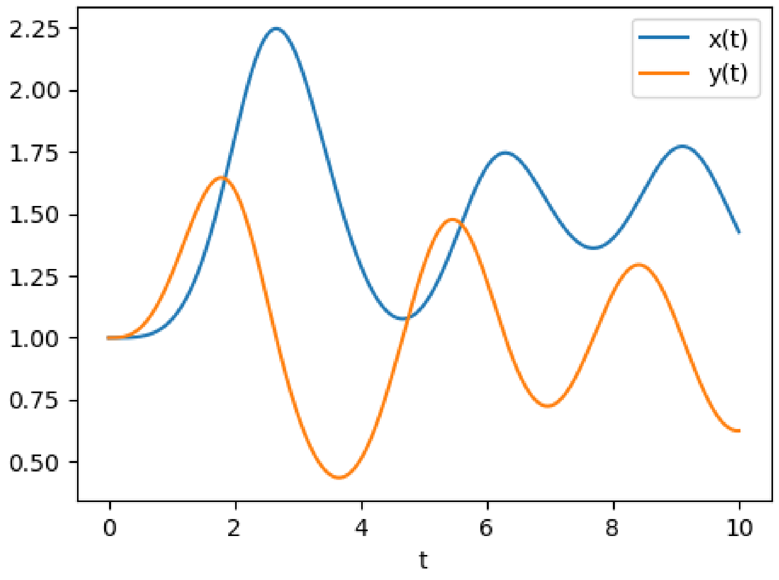

Figure 8.5: A solution to $dx/dt = xy - x$ and $dy/dt = y - xy + \sin^3 t$ with $x(0) = y(0) = 1$ from $t = 0$ to $t = 10$ computed by the fourth-order Runge-Kutta method.

Note that the function `f(r, t)` breaks the argument `r` apart into its components x and y, forms the values of f_x and f_y from them, and then put the values together into an array which is the final output of the function. Indeed, the construction of this function is the only complicated part of the program.

◇◇

8.3 Second-order Differential Equations

Up to now, we have only studied first-order differential equations. But first-order equations are in fact quite rare in physics. Many of the equations encountered in physics are second-order or higher. Luckily, solving second-order equations is pretty easy as we now know how to solve first-order ones due to the following trick.

Let us first consider the simple case where there is only one dependent variable x. The general form of a second-order differential equation with one dependent variable is

$$\frac{d^2x}{dt^2} = f\left(x, \frac{dx}{dt}, t\right), \tag{8.18}$$

In other words, the second derivative can be an arbitrary function (including possibly a nonlinear function) of x, t, and the derivative dx/dt. For instance, we could have

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

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

$$\frac{dx}{dt} = y \tag{8.19}$$

so that Eq. (8.18) can be written as

$$\frac{dy}{dt} = f(x, y, t). \tag{8.20}$$

Equations (8.19) and (8.20) are equivalent to the single second-order equation we started with, as we can prove by substituting Eq. (8.19) into Eq. (8.20) to recover Eq. (8.18) again. But both Equations (8.19) and (8.20) are first order. So this process reduces our second-order equation to two 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 example, the general form of a third-order equation is

$$\frac{d^3x}{dt^3} = f\left(x, \frac{dx}{dt}, \frac{d^2x}{dt^2}, t\right) \tag{8.21}$$

We define two additional variables y and z by

$$\frac{dx}{dt} = y, \qquad \frac{dy}{dt} = z \tag{8.22}$$

so that Eq. (8.21) becomes

$$\frac{dz}{dt} = f(x, y, z, t) \tag{8.23}$$

Between them, Equations (8.22) and (8.23) give us three first-order equations that are equivalent to our single third-order equation. So again we can solve the third-order equation using the methods we already know about for simultaneous first-order equations. This approach can be generalized to equations of any-order. But equations of order higher than three are rare in physics and so you probably won't need to solve them often.

The method can be also 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

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

which is equivalent to the first-order equations

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

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

◇◇

Example 8.5

A standard problem in physics is the linear pendulum in which you approximate the behavior of a pendulum by a linear differential equation. But a real pendulum is nonlinear. Consider a pendulum with an arm of length ℓ holding a bob of mass m (see Figure 8.6).

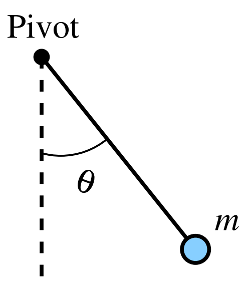

Figure 8.6: A nonlinear pendulum formed by an arm holding a bob.

In terms of the angular displacement θ 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. For the sake of simplicity, we ignore friction and assume the arm to be massless. The component of this force in the tangential direction is $mg\sin\theta$ which is always toward the rest point at $\theta = 0$. Hence Newton's second law gives us an equation of motion for the pendulum of the form

$$m\ell\frac{d^2\theta}{dt^2} = -mg\sin\theta$$

$$\Rightarrow\ \frac{d^2\theta}{dt^2} = -\frac{g}{\ell}\sin\theta$$

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

$$\frac{d\theta}{dt} = \omega$$

Then the above second-order equation becomes

$$\frac{d\omega}{dt} = -\frac{g}{\ell}\sin\theta$$

Between them, these two first-order equations are equivalent to the 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 indeed only 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 8.4 except that the function $\mathbf{f}(\mathbf{r}, t)$ must be redefined appropriately. Below is the Python program `nlpendrk4.py` which performs the calculation and plots the angular displacement θ as a function of time t for a pendulum with arm length $\ell = 0.1\,\mathrm{m}$ and the initial conditions $\theta = 179\pi/180$ and $\omega = 0$ at time $t = 0$. From the resultant plot in Figure 8.7, we can see a wavelike form curve which indicates that the pendulum is swinging back and forth.

```
# nlpendrk4.py
# This program solves the 2nd order ODE d^2 theta/dt^2 = -(g/l) sin
# theta for a nonlinear pendulum with arm length l = 0.1m, theta(0)
# = 179pi/180, and omega(0) = 0 using the fourth-order Runge-Kutta
# method.
```

```python
# Last update on 4 Dec 2020 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

g, l = 9.81, 0.1

def f(r,t):
    """ Function to compute the function f(r, t) = (ftheta(r, t),
        fomega(r, t)) where theta = r[0] and omega = r[1] """
    ftheta = r[1]
    fomega = -(g/l)*np.sin(r[0])
    return np.array([ftheta,fomega])

# Use the 4th-order Runge-Kutta method to solve the 2nd order ODE
a, b = 0.0, 10.0                # Start and end of the interval
N = 1000                        # Number of steps
h = (b-a)/N                     # Size of a single step
r = np.array([179*np.pi/180,0]) # Initial conditions
tpoints = np.linspace(a,b,N+1)
thetapoints, omegapoints = [r[0]], [r[1]]
for t in tpoints[:-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
    thetapoints.append(r[0])
    omegapoints.append(r[1])

# Plot the solution of the 2nd order ODE
fig, ax = plt.subplots()
ax.plot(tpoints,thetapoints)
ax.set_xlabel(r"Time $t$")
ax.set_ylabel(r"Angular displacement $\theta$")
ax.set_xlim([0, 10])
ax.set_ylim([-4, 4])
plt.show()
```

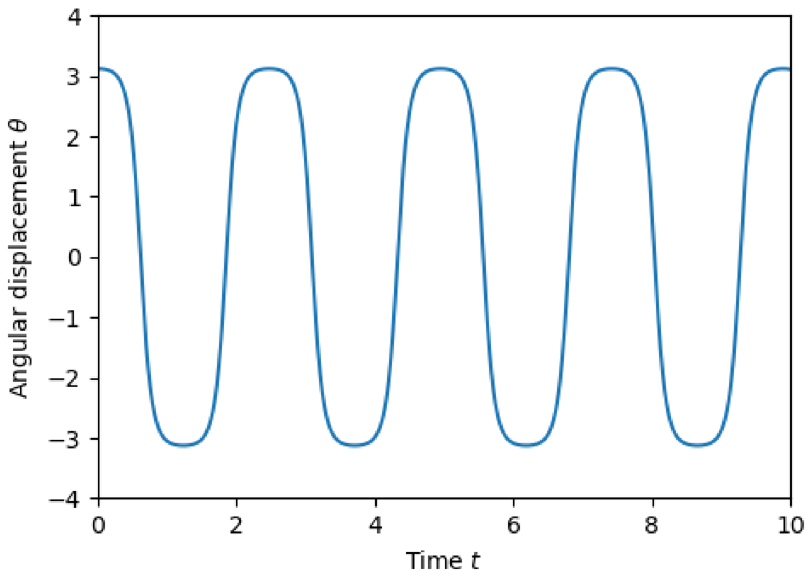

Figure 8.7: The angular displacement θ of the nonlinear pendulum as a function of time t computed by the fourth-order Runge-Kutta method.

◇◇

8.4 Adaptive Step Size Method

All the methods we have seen so far in this chapter use repeated steps of a fixed size h which is the size chosen by the programmer. However, we can get better results in most situations if we allow the step size to vary during the execution 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 the solution $x(t)$ as a function of time t looks like Figure 8.8. The function is varying slowly in some regions for which 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 larger in the regions where the solution varies little and smaller in the region we need more detail, then we can calculate the whole solution faster but still very accurately. 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.

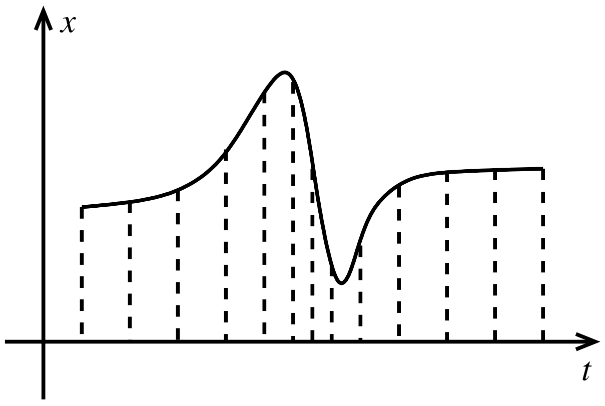

Figure 8.8: A differential equation whose solution $x(t)$ varies rapidly with time t in some regions but more slowly in others.

The basic idea behind an adaptive step size scheme is to vary the step size h so that the error introduced per unit interval in t is roughly constant. For instance, we might specify that we want an error of 0.001 or less per unit time 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 to each individual step will have to be more accurate overall.

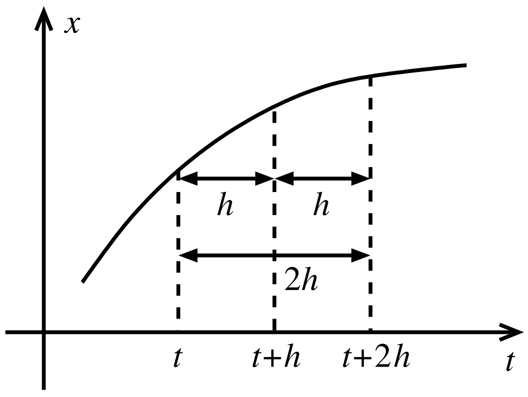

Figure 8.9: Starting from some time t, the adaptive step size method takes two steps of size h and a single step of size $2h$ to obtain two estimates of the solution at time $t + 2h$.

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 this approach works when it is applied to the fourth-order Runge-Kutta method. We choose some initial value of h which is typically very small, to be on the safe side. And we first do two steps of the solution with each of size h and one after another using our ordinary Runge-Kutta method (see Figure 8.8). So if we start at time t, we will get to time $t + 2h$

after two steps and get an estimate of $x(t+2h)$. Now here's the clever part: we go back to start again at time t and we do one more Runge-Kutta step, but this time with twice the size, i. e. with 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 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. In other words, the size of the error on a single step is ch^5 to leading order with 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 to say, the true value of $x(t+2h)$ is related to our estimated value, call it x_1, by

$$x(t+2h) = x_1 + 2ch^5.$$

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

$$x(t+2h) = x_2 + 32ch^5,$$

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

$$\epsilon = ch^5 = \frac{1}{30}(x_1 - x_2) \tag{8.24}$$

Our goal is to make the size of this error exactly equal to some target accuracy that we choose. In general, the two will not be exactly equal unless we are very lucky. Either Eq. (8.24) 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 calculation that meets the specified target accuracy.

So let us ask the following question: what value of the step size would make the size of the error in Eq. (8.24) exactly equal to the target, making our calculation exactly as accurate as we need but not more? Let us denote this step size h'. If we were to take steps of size h', then the error on a single step would be

$$\epsilon' = ch'^5 = ch^5\left(\frac{h'}{h}\right)^5 = \frac{1}{30}(x_1 - x_2)\left(\frac{h'}{h}\right)^5 \tag{8.25}$$

where we have used Eq. (8.24). Suppose 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 Eq. (8.25) is equal to the target accuracy. We are only interested in the absolute magnitude of the error, not its sign. So we want the value of h' that satisfies

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

Rearranging for h' we then find that

$$h' = h\left(\frac{30h\delta}{|x_1 - x_2|}\right)^{1/4} = h\rho^{1/4} \tag{8.26}$$

where

$$\rho = \frac{30h\delta}{|x_1 - x_2|} \tag{8.27}$$

Note that ρ is precisely the ratio of the target accuracy $h\delta$ and the actual accuracy $|x_1 - x_2|/30$ for steps of size h.

Thus the complete procedure of the adaptive step size method is as follows. Starting from the same point $x(t)$, we perform two steps of size h as well as one step of size $2h$. This gives us two estimates x_1 and x_2 of $x(t + 2h)$. We use these to calculate the ratio ρ in Eq. (8.27). If $\rho > 1$, then we know that the actual accuracy of our Runge-Kutta step 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 to avoid this waste. Plugging our value of ρ into Eq. (8.26) tells us exactly what the new larger value of h' of the step size should be to achieve this. On the other hand, if $\rho < 1$, then the actual accuracy of our calculation is poorer than the target accuracy. In other words, 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. (8.26) tells us what that step size should be.

As a result, 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 larger steps 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 the methods using fixed step size. We have to do at least three Rung-Kutta steps for every two that we actually used in calculating the solution, and sometimes more than three in cases where we have to repeat a step due to missing of our target accuracy. However, the extra effort usually pays off because the method gives 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 for the two estimates x_1 and x_2 to coincidentally agree with one another by chance. Errors are inherently unpredictable and the two can occasionally be the same or roughly the same just by luck. If this happens, then h' in Eq. (8.26) 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 example, a common rule of thumb is that it should not increase by more than a factor of two on any given pair of steps.

The adaptive step size method can be also used to solve simultaneous differential equations. In such cases, we need to decide how to generalize Eq. (8.24) which gives the error, or equivalently Eq. (8.27) which gives the ratio ρ, to the case of more than one dependent variable. The derivation leading to Eq. (8.24) can be duplicated for each variable to show that the variables x, y, etc. have separate errors:

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

and so forth. However, there is more than one way that these separate errors can be combined into a single overall error for use in Eq. (8.27), 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, then we might wish to perform a calculation that ensures the Euclidean error in the position of the point meets a certain target where Euclidean error refers to $\sqrt{\epsilon_x^2 + \epsilon_y^2}$. In that case it is straightforward to see that we would use the same formulas for the adaptive method as before except that $|x_1 - x_2|/30$ in Eq. (8.27) should be replaced with $\sqrt{\epsilon_x^2 + \epsilon_y^2}$. On the other hand, suppose we are performing a calculation like that in Example 8.5 for the nonlinear pendulum, where we are solving a single second-order equation for the angular displacement 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. (8.27) directly with x replaced by θ and ignore ω in the calculation of step sizes. Thus it may take a little thought to determine what is the appropriate generalization of the adaptive method to simultaneous equations for any particular calculation. But the answer usually becomes clear once the correct definition for the error on the calculation is determined.

There is a further point to note 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, i. e. 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 is really 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.

◇◇

Example 8.6

As an example of the adaptive step size method, let us return to the nonlinear pendulum in Example 8.5. Below is the Python program `nlpendrk4a.py` which performs the calculation for the pendulum using the fourth-order Runge-Kutta method with adaptive step sizes. The resultant plot of this program is depicted in Figure 8.10. In the plot, the solid curve shows the angular displacement θ of the pendulum versus time t while the vertical dotted lines show the position of every twentieth Runge-Kutta step in the calculation. We can observe the method makes the step sizes longer in the flat portions of the curve at the top and bottom of each swing of the pendulum. However, the step sizes are much smaller in the steep portions where the pendulum is moving rapidly which ensures accurate calculations of the pendulum motion.

```
# nlpendrk4a.py
# This program solves the 2nd order ODE d^2 theta/dt^2 = -(g/l) sin
# theta for a nonlinear pendulum with arm length l = 0.1m, theta(0)
# = 179pi/180, and omega(0) = 0 using the fourth-order Runge-Kutta
# method with adaptive step sizes.
# Last update on 7 Jan 2022 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

g, l = 9.81, 0.1

def f(r,t):
    """ Function to compute the function f(r, t) = (ftheta(r, t),
        fomega(r, t)) where theta = r[0] and omega = r[1] """
    ftheta = r[1]
    fomega = -(g/l)*np.sin(r[0])
    return np.array([ftheta,fomega])

def rn(r,t,h):
    """ Function to compute the new value of r using the 4th-order
        Rung-Kutta method with step size h """
    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)
    rn = r + (k1+2*k2+2*k3+k4)/6
    return rn

# Use the 4th-order Runge-Kutta method with adaptive step sizes to
# solve the 2nd order ODE
a, b = 0.0, 10.0                # Start and end of the interval
N = 1000                        # Number of steps
h = (b-a)/N                     # Size of a single step
r = np.array([179*np.pi/180,0]) # Initial conditions
delta = 1e-6                    # Target accuracy for each step
t = a
tpoints = [t]
thetapoints, omegapoints = [r[0]], [r[1]]
```

```python
ctr = 0
vlinelist = []
while t < b:
    ctr += 1
    while True:
        r0 = rn(r,t,h)
        r1 = rn(r0,t+h,h)
        r2 = rn(r,t,2*h)
        if np.isclose(r1[0], r2[0], rtol=0) == 0:
            rho = 30*h*delta/abs(r1[0] - r2[0])
        else:
            rho = 10000
        if rho >= 1:
            tpoints.extend([t+h, t+2*h])
            thetapoints.extend([r0[0], r1[0]])
            omegapoints.extend([r0[1], r1[1]])
            t += 2*h
            r = r1
            h *= min(rho**0.25, 2.0)
            break
        h *= rho**0.25
    if (ctr % 10) == 0:
        vlinelist.append(t)

# Plot the solution of the 2nd order ODE
fig, ax = plt.subplots()
for i in vlinelist:
    ax.axvline(i, linestyle="--", color="#a9a9a9")
ax.plot(tpoints,thetapoints)
ax.set_xlabel(r"Time $t$")
ax.set_ylabel(r"Angular displacement $\theta$")
ax.set_xlim([0, 10])
ax.set_ylim([-4, 4])
plt.show()
```

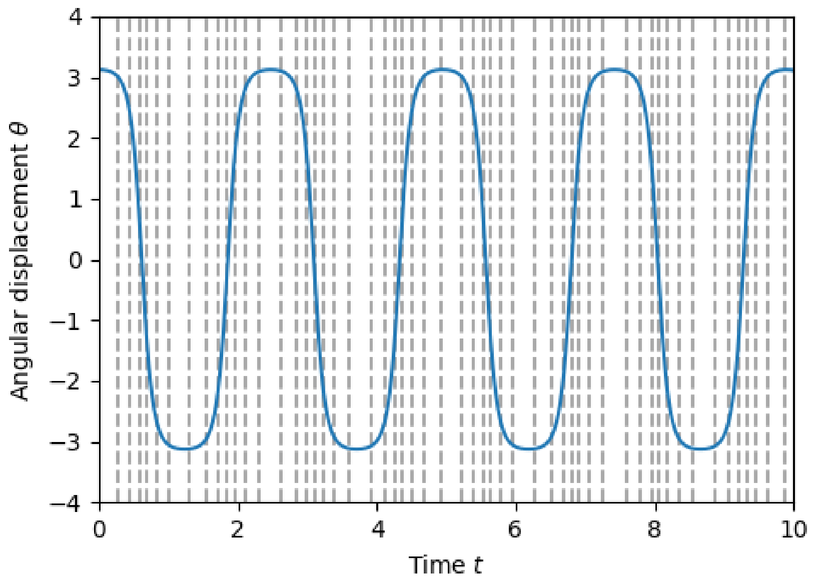

Figure 8.10: The angular displacement θ of the nonlinear pendulum as a function of time computed by the fourth-order Runge-Kutta method with adaptive step sizes. The vertical dotted lines show the position of every twentieth Runge-Kutta step.

◇◇

All the examples we have considered so far in this chapter have been **initial value problems**, i. e. differential equations with the initial values of the variables given. This is the most common form of differential equation problems encountered in physics. But it is not the only one. There are also **boundary value problems**, i. e. differential equations with the values of the variables given at more than one points. In general, it is harder to solve boundary value problems than initial value problems. We will not go into the discussion of solving boundary value problems here as it's out of our current scope.