Integrating Equations of Motion

9 March 2020

PHY2004W KDSMIL001

Contents

1 Introduction

This computational activity was an introduction into the world of numerical integration, which is one of the methods that we can use to solve differential equations, something that we have to do quite often in physics as we are often solving second order ordinary differential equations that come out of applying Newton's second law. This is often reduced to two coupled equations:

$$\frac{d\vec{p}}{dt} = \vec{F} \quad \text{and} \quad \frac{d\vec{q}}{dt} = \frac{\vec{p}}{m} \tag{1}$$

where $\vec{p}$ is a momentum, $\vec{F}$ is a net force, $\vec{q}$ is a coordinate, and m is a mass.
In order to numerically solve these equations, we approximate the derivatives, which results in 2 equations, which we call update equations:

$$\begin{aligned} \vec{p}(t+\Delta t) &= \vec{p}(t) + \vec{F}(\vec{q}(t))\Delta t \\ \vec{q}(t+\Delta t) &= \vec{q}(t) + \left(\frac{\vec{p}(t)}{m}\right)\Delta t \end{aligned} \tag{2}$$

These are what we will use in this Activity in order to approximate the motion of, in this case, a simple pendulum. These equations are very tedious to evaluate for all t, but that's where the computer comes in.

2 Activity

The first thing that we need to do is check the accuracy of these equations (2). In order to do this, we perform a Taylor expansion of $y(t+\Delta t)$, truncating after only 2 terms, a first order approximation, making the error small, but still significant:

$$\begin{aligned} y(t+\Delta t) &= \sum_{n=0}^{\infty} \frac{y^{(n)}(t)\Delta t^n}{n!} \\ &= y(t) + \dot{y}(t)\Delta t + \ddot{y}(t)\Delta t^2 + \dots \end{aligned}$$

When truncating this at the second term, this gives us something of the form of the equation for $\vec{p}$ in Equation 2, which confirms that we are using first order approximations in this activity.
For the sake of interest, we can try to get a second order approximation of the first derivative. For the sake of computation, we have approximated the derivative to be

$$\frac{dy}{dt} \approx \frac{y(t+\Delta t) - y(t)}{\Delta t} \tag{3}$$

From this, we can substitute in a second order approximation for $y(t+\Delta t)$, which returns

$$\begin{aligned}\frac{dy}{dt} &\approx \frac{y(t) + \dot{y}(t)\Delta t + \ddot{y}(t)\Delta t^2 - y(t)}{\Delta t} \\ &= \dot{y}(t) + \ddot{y}(t)\Delta t\end{aligned} \tag{4}$$

We see that this gives us something of the form of the equation for $\vec{q}$ in Equation 2. Next, we need to create the model that we will be using to simulate the system of interest. In its most complete form, the equation of motion for a pendulum is:

$$\frac{d^2\theta}{dt^2} + \Omega_0^2 \sin\theta = 0 \tag{5}$$

where $\Omega_0^2 = \frac{g}{L}$, θ is the angle from the vertical, and L is the length of the pendulum. This equation comes from an analysis of the forces on the system. We know that the torque $\tau = I\alpha$, where $I = mL^2$ is the moment of inertia for a pendulum with a massless string and $\alpha = \ddot{\theta}$ is the angular acceleration. Then, by equating all the forces, we get

$$\begin{aligned}\tau = I\alpha &= mL^2\ddot{\theta} \\ \text{Equating forces: } mL^2\ddot{\theta} &= -mg\sin\theta L \\ \implies \ddot{\theta} + \frac{g}{L}\sin\theta &= 0\end{aligned}$$

In order to simplify this, we can take what we call a small angle approximation, where $\sin\theta = \theta$ for angles $<\sim 10^\circ$. This leads us to the equation

$$\ddot{\theta} = -\Omega_0^2\theta \tag{6}$$

which we know is the general form for a simple harmonic oscillator.

Now, going back to Equation 2, if we let $\Omega_0 = \sqrt{\frac{g}{L}}$, then we get the equation as in (5). More manipulation will lead to a coupled form of this equation:

$$\frac{d\omega}{dt} = -\Omega_0^2 \sin\theta \quad \text{and} \quad \frac{d\theta}{dt} = \omega \tag{7}$$

Given the initial conditions of $\theta(0) = 90^\circ, \omega(0) = 0$, we interpret this as the pendulum being held at 90° to the vertical and at rest (angular velocity $\omega = 0$), then let go. In the small angle approximation, the system should oscillate with an angular frequency of Ω_0^2, but these initial conditions do not satisfy the small angle approximation, so we will need another method in order to figure out what the motion of the system will look like, for example numerical integration.

In order to perform this numerical integration, we need to have some equations in the form of Equation 2, which we can adapt to have the following:

$$\begin{aligned} p_{i+1} &= p_i - \Omega_0^2 \sin(q_i)\Delta t \\ q_{i+1} &= q_i + p_i \Delta t \end{aligned} \tag{8}$$

where $i = 0, 1, 2 \ldots$ and $q \equiv \theta, p \equiv \omega$.
We can do this because we got the equations for the motion of the pendulum into a similar form in Equation 7 as in Equation 2. This is known as Euler's Method and is a commonly used method for numerically solving systems that are hard or impossible to solve analytically. Now we can begin to implement this in Python in order for the computer to do the heavy lifting.

The entire code for this calculation is in Appendix 1 but the most important part is the loop we use a loop to continuously update the values of θ and ω, appending them to an array so that we can plot them later on, which can be seen below.

```
25 for i in range(N-1):
26     time[i+1] = (i+1)*deltat
27     p[i+1] = p[i] - (omega0**2)*np.sin(q[i])*deltat
28     q[i+1] = q[i] + p[i]*deltat
29     E[i+1] = 0.5*p[i]**2 + 9.8*(1-np.cos(q[i]))
```

Euler's Method Loop

We also calculate a value for the total energy of the system at each step, the equation for which is

$$E = E_K + E_P = \frac{1}{2} m L^2 \omega^2 + mgL(1 - \cos\theta) \tag{9}$$

In this case, we are not given a value for the mass or the length of the pendulum, so we assume them to be 1 as this shouldn't affect the shape of the plot, just the scale of it. Below is the first plot [Figure 1] which has initial conditions of $\theta = 90^\circ$ and $\omega = 0$. This clearly does not satisfy the conditions for a small angle approximation, and thus at around 6.5s, the plots go haywire, with θ and ω becoming only negative, which is not a natural way for this system to behave.

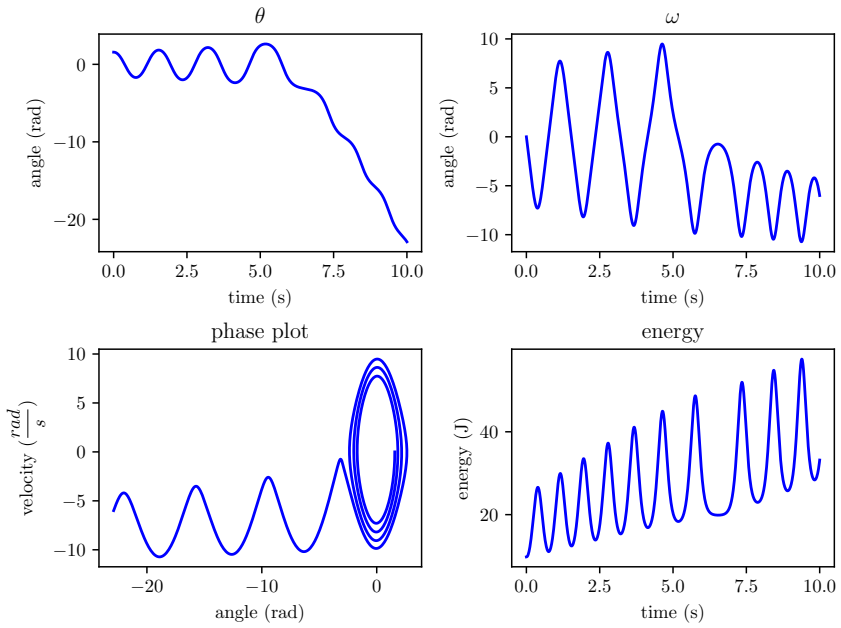

Figure 1: Plotting Values for Large θ

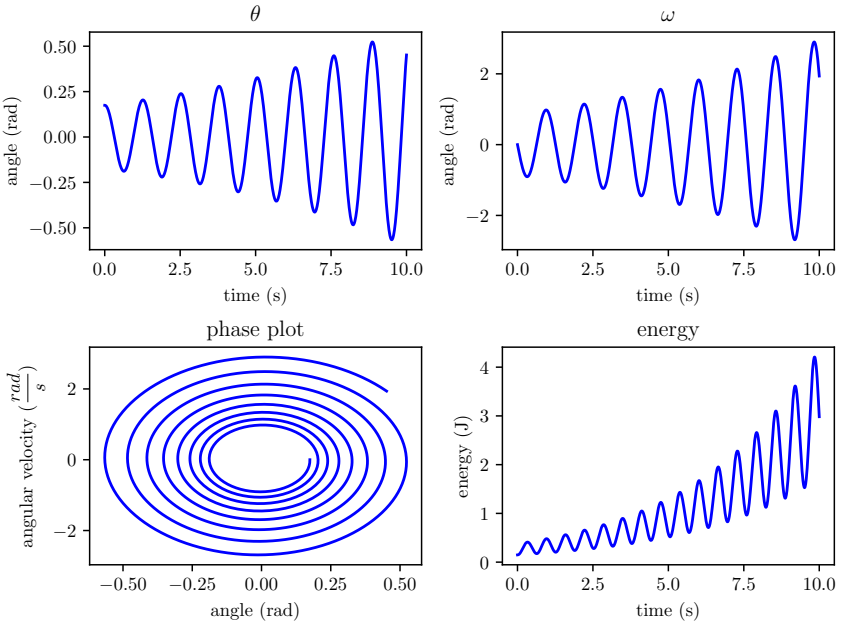

Figure 2: Plotting Values for Small θ

Figure 2 is another plot of the same thing, except with a initial angle of 10°. This is well within the small angle approximation conditions and thus it seems to behave somewhat naturally, oscillating around $\theta, \omega = 0$.

The plot on the bottom left of the Figure is what we call a phase plot, effectively plotting the position of the bob against the velocity of the bob. It reveals information about the solution to the differential equation that describes the motion we're considering. In this case, for the two plots above, we notice that the solution is periodic in some sense but seems to be spiralling out of control, which we can attribute to the increasing amplitude of the oscillations. It's also clear that energy is not being conserved, which we will discuss next.
Firstly, θ seems to increase in amplitude as time goes on, as well as ω. This obviously has an effect on the energy of the system, which seems to be increasing at some kind of exponential rate. It becomes obvious that the reason for this is rounding errors in the code when we change Δt to be 0.001, rather than its initial value of 0.01, producing the plots below in Figure 3.

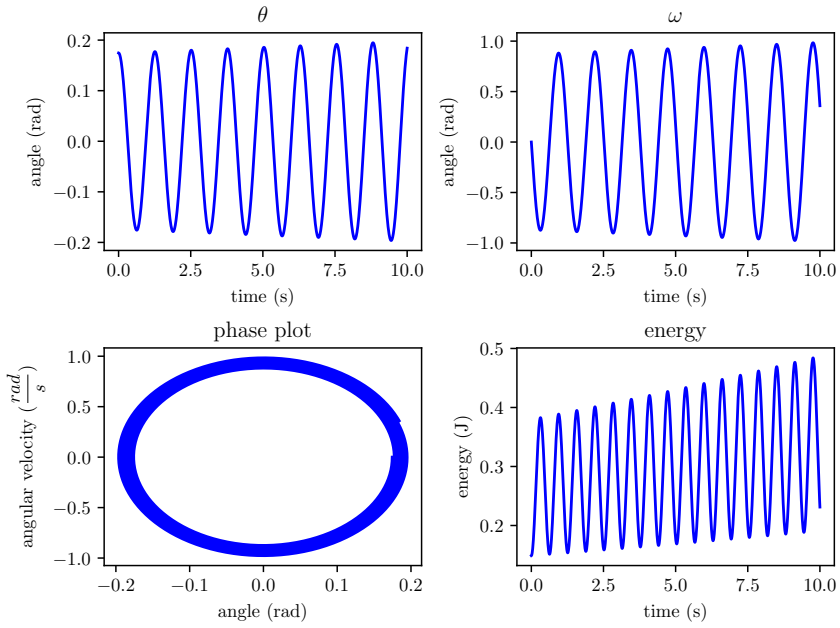

Figure 3: Plotting values for Small θ with $\Delta t = 0.001$

Here we can see that, while energy isn't conserved entirely, it's doing a much better job of it than it was in the previous example. We can also observe that the phase plot is much more elliptical and therefore purely periodic, which we expect when analysing a pendulum oscillating with small θ. In order to analytically prove that the energy in-

creases with each step when looking at small angle approximations, we look at Equation 7:

$$p_{i+1} = p_i - \Omega_0^2 \sin(q_i)\Delta t$$
$$q_{i+1} = q_i + p_i \Delta t$$

At small angles we can approximate

$$\sin\theta = \theta$$
$$(1 - \cos\theta) = 1$$
$$\implies p_{i+1} = p_i - \Omega_0^2 q_i \Delta t$$

Now recall

$$E = \frac{1}{2}mL^2\omega^2 + mgL(1 - \cos\theta)$$
$$\implies E = \frac{1}{2}mL^2\omega^2 + mgL$$

It's important to notice that this E depends on ω, which, when calculated in the program, is done so using a Taylor expansion which is truncated after the second term. While this error is not great, when squared and compounded 1000 times as it is in the first two examples, it's obvious that this is the source of the error. The only problem is that this error seems to be coming from a quadratic term, while the increase in energy seems to be exponential, which we can't seem to make sense of.

One solution to the energy conservation problem is an adjustment to the method we use to approximate the values of, in this case, ω and θ. The equations are valid for the entire interval of interest and so we can evaluate a point later in time in order to better approximate the derivative of the function of interest. This is an improvement to the Euler's method that we have been using and it doesn't quite solve the issue of energy not being conserved, but it does give a bound to the energy, making it periodic. This method is known as Symplectic Integration or the Symplectic Euler Method. Below, Figure 4, is the set of plots for this method.

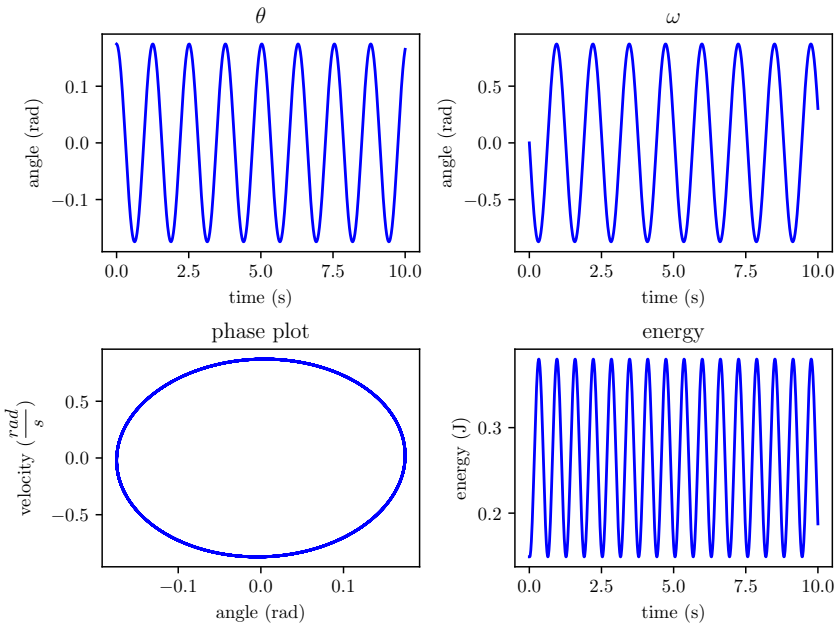

Figure 4: Symplectic Integration Method with θ small

The code for this entire section is in Appendix 2 and it can be seen that it is practically identical to the first section with the exception of the loop, which can be seen below.

```
25 for i in range(N-1):
26     time[i+1] = (i+1)*deltat
27     p[i+1] = p[i] - (omega0**2)*np.sin(q[i])*deltat
28     q[i+1] = q[i] + p[i+1]*deltat
29     E[i+1] = 0.5*p[i]**2 + 9.8*(1-np.cos(q[i]))
```

Symplectic Euler's Method Loop

As can be seen in Figure 4, this method is much better at keeping the energy bounded, with a total variation of ~ 0.2. Compared to the previous results, this is a great improvement in terms of accurately approximating the motion. We can again notice that the phase plot is purely elliptical, with no decay or growth, showing that our energy is conserved to some extent. We have effectively removed the approximation from the calculation, rather approximating the entire problem but accurately calculating the values.

Finally, we will investigate the dependence of period on either the amplitude or the length. In our case, we will investigate length. In order to do this, we have written some code, Appendix 3, that is in most senses the exact same as the first two sections with a few extra arrays and calculations added. Most importantly, we added an

extra loop so that we can simulate the motion of the pendulum for a range of different lengths, namely 1m to 100m, evaluating every metre. From this, we performed a Levenberg-Marquardt fit using the `curve_fit` function in `scipy.optimize` and could then extract the angular frequency and from that find the period. Below are the most important lines from the code.

```
32 for j in range(NL):
33     time = np.zeros(N)
34     omega[j] = np.sqrt(9.8/L[j])
35     for i in range(N-1):
36         time[i+1] = (i+1)*deltat
37         p[j, i+1] = p[j, i] - (omega[j]**2)*np.sin(q[j, i])*deltat
38         q[j, i+1] = q[j, i] + p[j, i+1]*deltat
39     popt, pcov = curve_fit(f, time, q[j], p0, maxfev=5000)
40     T[j] = abs(2*np.pi/popt[2])
41     TSmallAngles[j] = 2*np.pi/omega[j]
```

Length Dependence of Period Code

Line 40 is the calculation of the observed period and line 41 is a calculation of the expected period given the angular frequency Ω_0 supplied in line 37. Below, Figure 5, is the plot of the observed period and the predicted period. Importantly, the predicted period is based off the small angle approximation, where $T = \frac{2\pi}{\Omega_0}$.

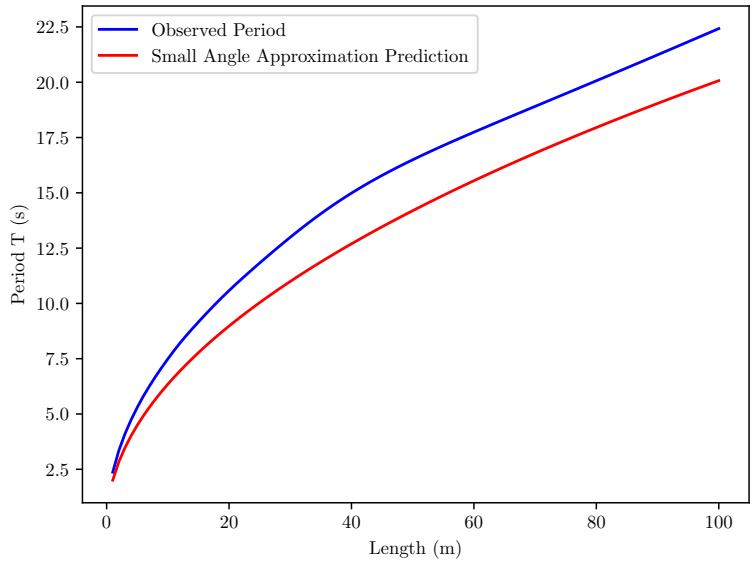

Figure 5: Length Dependence of Period with Starting $\theta = 90°$

As can be seen, the prediction underestimates the period by an increasingly significant amount. However, when oscillating at small angles, as below in Figure 6, the small angle approximation predicts the length dependence of period quite accurately. It might not be very visible but the two plots are practically on top of one another.

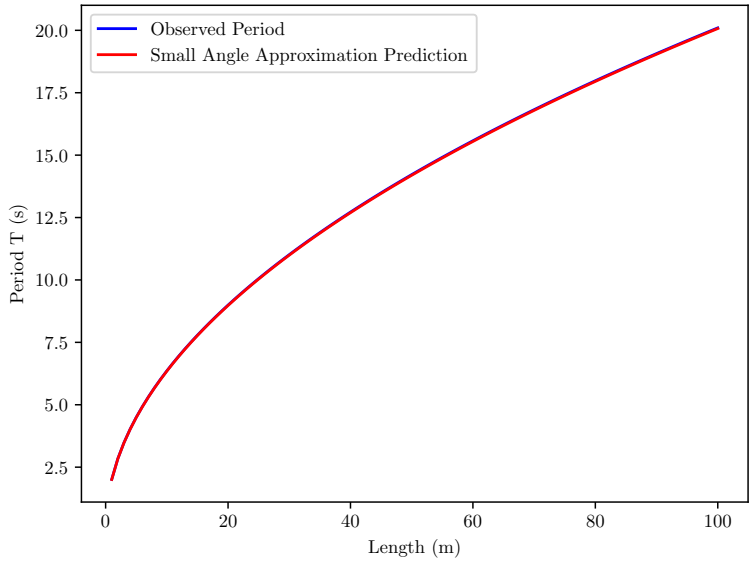

Figure 6: Length Dependence of Period with Starting $\theta = 10^{\circ}$

From these two results we can conclude that at small angles Euler's Method, when using Symplectic Integration, is quite good at modelling something like an simple oscillator. However, when deviating slightly from the bounds of the small angle approximation, these numerical methods do well but are not accurate enough to trust entirely.

In conclusion,

3 Appendix

Appendix 1: Pendulum Update Code

```
from matplotlib import pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
import matplotlib
matplotlib.use('pgf')
matplotlib.rcParams.update({
    'pgf.texsystem': 'pdflatex',
    'font.family': 'serif',
    'text.usetex': True,
    'pgf.rcfonts': False,
})
# Defines a few starting conditions and constants we will need later on
deltat = 0.01       # The time step
omega0 = 5          # The angular frequency
T = 10
N = int(T/deltat+1.5)
time = np.zeros(N)
p = np.zeros(N)
q = np.zeros(N)
E = np.zeros(N)
q[0] = 10*np.pi/180 # Our initial condition of omega at 10 degrees to the vertical
p[0] = 0            # Another initial condition that says we start from rest
E[0] = 0.5*p[0]**2 + 9.8*(1-np.cos(q[0])) # The loop will start assigning from 1
# The Loop. Does all the calculations of theta and omega for each time step
for i in range(N-1):
    time[i+1] = (i+1)*deltat
    p[i+1] = p[i] - (omega0**2)*np.sin(q[i])*deltat
    q[i+1] = q[i] + p[i]*deltat
    E[i+1] = 0.5*p[i]**2 + 9.8*(1-np.cos(q[i]))
# Plots the value of theta, omega, the phase plot, and the energy respectively
plt.subplot(221)
plt.plot(time, q, 'b-')
plt.xlabel('time (s)')
plt.ylabel('angle (rad)')
plt.title('$\\theta$')
plt.subplot(222)
plt.plot(time, p, 'b-')
plt.xlabel('time (s)')
plt.ylabel('angle (rad)')
plt.title('$\omega$')
```

```
plt.subplot(223)
plt.plot(q, p, 'b-')
plt.xlabel('angle (rad)')
plt.ylabel('angular velocity ($\\frac{rad}{s}$)')
plt.title('phase plot')
plt.subplot(224)
plt.plot(time, E, 'b-')
plt.xlabel('time (s)')
plt.ylabel('energy (J)')
plt.title('energy')
plt.tight_layout(pad=0.8)
# Saves the plots to a pgf for inclusion in the report
plt.savefig('PHY2004W Computational\CP3\CP3 10.pgf')
```

Appendix 2: Symplectic Euler Method

```
from matplotlib import pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
import matplotlib
matplotlib.use('pgf')
matplotlib.rcParams.update({
    'pgf.texsystem': 'pdflatex',
    'font.family': 'serif',
    'text.usetex': True,
    'pgf.rcfonts': False,
})
# Defines a few starting conditions and constants we will need later on
deltat = 0.01       # The time step
omega0 = 5          # The angular frequency
T = 10
N = int(T/deltat+1.5)
time = np.zeros(N)
p = np.zeros(N)
q = np.zeros(N)
E = np.zeros(N)
q[0] = 90*np.pi/180 # Our initial condition of omega at 10 degrees to the vertical
p[0] = 0            # Another initial condition that says we start from rest
E[0] = 0.5*p[0]**2 + 9.8*(1-np.cos(q[0])) # The loop will start assigning from 1
# The Loop. Does all the calculations of theta and omega for each time step
for i in range(N-1):
    time[i+1] = (i+1)*deltat
    p[i+1] = p[i] - (omega0**2)*np.sin(q[i])*deltat
    q[i+1] = q[i] + p[i+1]*deltat
    E[i+1] = 0.5*p[i]**2 + 9.8*(1-np.cos(q[i]))
```

```
# Plots the value of theta, omega, the phase plot, and the energy
    respectively
plt.subplot(221)
plt.plot(time, q, 'b-')
plt.xlabel('time (s)')
plt.ylabel('angle (rad)')
plt.title('$\\theta$')
plt.subplot(222)
plt.plot(time, p, 'b-')
plt.xlabel('time (s)')
plt.ylabel('angle (rad)')
plt.title('$\omega$')
plt.subplot(223)
plt.plot(q, p, 'b-')
plt.xlabel('angle (rad)')
plt.ylabel('velocity ($\\frac{rad}{s}$)')
plt.title('phase plot')
plt.subplot(224)
plt.plot(time, E, 'b-')
plt.xlabel('time (s)')
plt.ylabel('energy (J)')
plt.title('energy')
plt.tight_layout(pad=0.8)
# Saves the plots to a pgf for inclusion in the report
plt.savefig('PHY2004W Computational\CP3\CP3b 10.pgf')
```

Appendix 3: Length Dependence of Period

```
from matplotlib import pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
import matplotlib
matplotlib.use('pgf')
matplotlib.rcParams.update({
    'pgf.texsystem': 'pdflatex',
    'font.family': 'serif',
    'text.usetex': True,
    'pgf.rcfonts': False,
})
# Defines a few starting conditions and constants we will need later
    on
deltat = 0.01       # The time step
omega0 = 5          # The angular frequency
TotalT = 10
N = int(TotalT/deltat+1.5)
NL = 100
L = np.linspace(1, 100, NL)
p = np.zeros((NL, N))
q = np.zeros((NL, N))
E = np.zeros((NL, N))
```

```
T = np.zeros(NL)
TSmallAngles = np.zeros(NL)
yfit = np.zeros((NL, N))
omega = np.zeros(NL)
p[:, 0] = 0            # Another initial condition that says we start from rest
q[:, 0] = 90*np.pi/180 # Our initial condition of theta at 90 degrees to the vertical
p0 = [1, 0, 1, 1]
def f(t, A, gamma, omega, alpha):
    return A*np.exp(-1*gamma*t/2)*np.cos(omega*t-alpha)
# The Loop. Does all the calculations of theta and omega for each time step and then does it again for a different omega
for j in range(NL):
    time = np.zeros(N)
    omega[j] = np.sqrt(9.8/L[j])
    for i in range(N-1):
        time[i+1] = (i+1)*deltat
        p[j, i+1] = p[j, i] - (omega[j]**2)*np.sin(q[j, i])*deltat
        q[j, i+1] = q[j, i] + p[j, i+1]*deltat
    popt, pcov = curve_fit(f, time, q[j], p0, maxfev=5000)
    T[j] = abs(2*np.pi/popt[2])
    TSmallAngles[j] = 2*np.pi/omega[j]
# Plots the two calculated periods per length
plt.plot(L, T, 'b-', label='Observed Period')
plt.plot(L, TSmallAngles, 'r-', label='Small Angle Approximation Prediction')
plt.xlabel('Length (m)')
plt.ylabel('Period T (s)')
plt.legend()
# Saves the plots to a pgf for inclusion in the report
plt.savefig('PHY2004W Computational\CP3\CP3c 90.pgf')
```