Numerical Integration of Differential Equations: Analysis of The Damped Harmonic Oscillator

Jiachen Guo
10090371

School of Physics and Astronomy
The University of Manchester

Second Year Laboratory Report

April 2018

Abstract

The accuracy of four numerical methods in modelling the motion of an unforced harmonic oscillator was investigated by comparing to the analytical solution. The four numerical methods used were the Euler method, the Improved Euler method, the Verlet method and the Euler-Cromer method. The Verlet method was found to be the most accurate method when used with a step size of 0.001 s. The Euler and Improved Euler methods were found to be non-symplectic. The Verlet method was used to model forced harmonic oscillations, with the external force applied being either instantaneous or sinusoidal. Applying an instantaneous force was found to change the amplitude of oscillation while the frequency of oscillation remained constant. Sinusoidal forced oscillations were found to oscillate at the driving frequency after a brief transient period.

1 Introduction

The most basic form of periodic motion is the simple harmonic motion. It is most often found in systems where amplitude of oscillation is small, such as in the microscopic vibrations of atoms in a crystal lattice and between two nuclei of a hydrogen molecule [1]. In the presence of dissipative forces, the motion becomes the damped and the system loses energy steadily. Such systems are called the damped harmonic oscillators. Analytical solutions of simple or damped harmonic motion can be obtained in cases where no external driving force is applied to the system and in cases where a sinusoidal driving force is applied. For other cases such as in the case of application of an instantaneous external force, the equation of motion can be solved numerically [2]. The numerical methods used here are the Euler method, the Improved Euler method, the Verlet method and the Euler-Cromer method.

2 Theory

The equation of motion for a damped harmonic oscillator with displacement $x(t)$, mass m, spring constant k, damping term b and external driving force $F(t)$ at time t is

$$m\frac{d^2x(t)}{dt^2} + b\frac{dx(t)}{dt} + kx(t) + F(t) = 0 \tag{1}$$

2.1 Analytical Solution

In the case where external driving force $F(t) = 0$, an analytical solution for $x(t)$ can be found. The general form of solution for $x(t)$ in this case depends on the damping term b.

We define a critical damping term $b_{critical}$ given by

$$b_{critical} = 2\sqrt{km} \tag{2}$$

When the damping term b of the damped harmonic oscillator is less than $b_{critical}$, i.e. $b < b_{critical}$, light damping occurs and the solution for $x(t)$ is given by

$$x(t) = A_0 \exp\left(\frac{-\gamma}{2}t\right)\sin(wt + \phi), \tag{3}$$

where A_0 is given by

$$A_0 = \sqrt{x_0^2 + \frac{v_0^2 m}{k}}, \tag{4}$$

where x_0 and v_0 are the initial displacement and initial velocity of the oscillator at $t = 0$,

γ is given by

$$\gamma = \frac{b}{m}, \tag{5}$$

w is given by

$$w = \sqrt{w_0^2 - \frac{\gamma^2}{4}}, \tag{6}$$

where $w_0 = \sqrt{\frac{k}{m}}$ is the natural frequency of the oscillator,

and ϕ is given by

$$\phi = \sin^{-1}(\frac{x_0}{A_o}) = \cos^{-1}(\frac{v_0}{A_0 w_0}) \tag{7}$$

It is noted that ϕ can take multiple values due to the repeating nature of the trigonometric functions. It is thus convenient to fix $0 \leq \phi < 2\pi$ to obtain a single unique solution for $x(t)$.

When the damping term b of the damped harmonic oscillator is equal to $b_{critical}$, i.e. $b = b_{critical}$, the general form of solution for $x(t)$ is given by

$$x(t) = A\, exp\left(-\frac{\gamma t}{2}\right) + Bt \exp\left(-\frac{\gamma t}{2}\right) \tag{8}$$

where A is given by

$$A = x_0, \tag{9}$$

B is given by

$$B = \frac{\gamma}{2} x_0 + v_0 \tag{10}$$

and x_0, v_0 and γ are as defined in Equations (4) and (5).

When the damping term b of the damped harmonic oscillator is greater than $b_{critical}$, i.e. $b > b_{critical}$, the general form of solution for $x(t)$ is given by

$$x(t) = C \exp\left[\left(-\frac{\gamma}{2} + \alpha\right) t\right] + D \exp\left[\left(-\frac{\gamma}{2} - \alpha\right) t\right] \tag{11}$$

where C is given by

$$C = x_0 - D, \tag{12}$$

D is given by

$$D = \frac{1}{2\alpha}\left[\left(-\frac{\gamma}{2} + \alpha\right) x_0 - v_0\right], \tag{13}$$

α is given by

$$\alpha = \sqrt{\frac{\gamma^2}{4} - w_0^2} \tag{14}$$

and x_0, v_0, γ and w_0 are as defined in Equations (4), (5) and (6).

Solutions of $v(t)$ where $v(t)$ is the velocity function of the oscillator for each of these three cases can then be found easily by differentiating the corresponding $x(t)$ functions with respect to time.

They are given by

$$v(t) = A_0 w \exp\left(\frac{-\gamma}{2}t\right)\cos(wt + \phi) - A_0\frac{\gamma}{2}\exp\left(\frac{-\gamma}{2}t\right)\sin(wt + \phi) \tag{15}$$

for a lightly damped oscillator,

$$v(t) = -\frac{\gamma}{2}A\ exp\left(-\frac{\gamma t}{2}\right) + B\ exp\left(-\frac{\gamma t}{2}\right) - \frac{\gamma}{2}Bt\ exp\left(-\frac{\gamma t}{2}\right) \tag{16}$$

for a critically damped oscillator and

$$v(t) = \left(-\frac{\gamma}{2} + \alpha\right)C\exp\left[\left(-\frac{\gamma}{2} + \alpha\right)t\right] - \left(\frac{\gamma}{2} + \alpha\right)D\exp\left[\left(-\frac{\gamma}{2} - \alpha\right)t\right] \tag{17}$$

for a heavily damped oscillator.

The energy of the oscillator at any point in time is then given by

$$E = \frac{1}{2}mv(t)^2 + \frac{1}{2}kx(t)^2 \tag{18}$$

where E is the energy of the oscillator.

2.2 Numerical Solutions

Besides using the analytical method, Equation (1) can be solved numerically by applying algorithms that use numerical approximation. First we note that Equation (1) can be rearranged to give

$$a(t) = -\frac{b}{m}v(t) - \frac{k}{m}x(t) - F(t) \tag{19}$$

where $a(t)$is the acceleration of the oscillator at time t.

In the case of numerical integration of n simulation steps and step size h, Equation (18) becomes

$$a_n = -\frac{b}{m}v_n - \frac{k}{m}x_n - F_n \tag{20}$$

Euler Method	$v_{n+1} = v_n + ha_n + O(h^2)$ (21) $x_{n+1} = x_n + hv_n + O(h^2)$ (22)
Improved Euler Method	$v_{n+1} = v_n + ha_n + O(h^2)$ (23) $x_{n+1} = x_n + hv_n + \frac{1}{2}h^2 a_n + O(h^3)$ (24)
Euler-Cromer Method	$v_{n+1} = v_n + ha_n + O(h^2)$ (25) $x_{n+1} = x_n + hv_{n+1} + O(h^2)$ (26)
Verlet Method	$x_{n+1} = Ax_n + Bx_{n-1} + 2\frac{F_n h^2}{D} + O(h^4)$ (27) $A = \frac{2(2m - kh^2)}{D}, B = \frac{bh - 2m}{D}, D = 2m + bh$ (28)

Table 1: Summary of core equations of the four numerical methods. The Verlet method has the smallest truncation error if h $\ll 1$ and is expected to be the most accurate.

where a_n, v_n, x_n and F_n are the acceleration, velocity, displacement and external driving force at step n respectively. This is the acceleration used in each simulation step of the four numerical methods.

A summary of the core equations used to calculate the displacement and velocity of the oscillator at each simulation step for each numerical method is presented in Table 1. In each case there is a truncation error of at least of the order h^2 which affects the accuracy of each method.

From Table 1 and Equation (20), it can be seen that the Euler, Improved Euler and Euler Cromer methods are self-starting. Numerical solutions of v_n and x_n can be found iteratively if given an initial velocity v_{n_0} and initial displacement x_{n_0} for all $n > n_0$ if F_n is given. For the Verlet method, the first simulation step needs to be carried out by the Improved Euler method before using the Verlet method for subsequent steps.

3 Method

The harmonic oscillator simulated in this experiment had a mass of 2.16 kg, spring constant of 0.89 Nm^{-1}, an initial displacement of 0 m and an initial velocity of -1 m.

An external driving force of 0 N was used to test the accuracy of the four numerical methods in approximating the solutions of the harmonic oscillations. Damping terms of 0.0 kgs^{-1}, 0.3 kgs^{-1}, 0.5 kgs^{-1}, 2.8 kgs^{-1} and 4.0kgs^{-1} were used. For each case of damping, the analytical solutions of displacement and energy were first calculated. The numerical solutions for displacement and energy were then calculated using 500 steps with a step size of 0.1 s. The accuracy of the numerical solutions were determined by considering how well the energy solution calculated numerically compares with the energy solution calculated analytically. This was done by calculating the total accumulated energy difference of each numerical method from the analytical energy solution at each simulation step. These energy

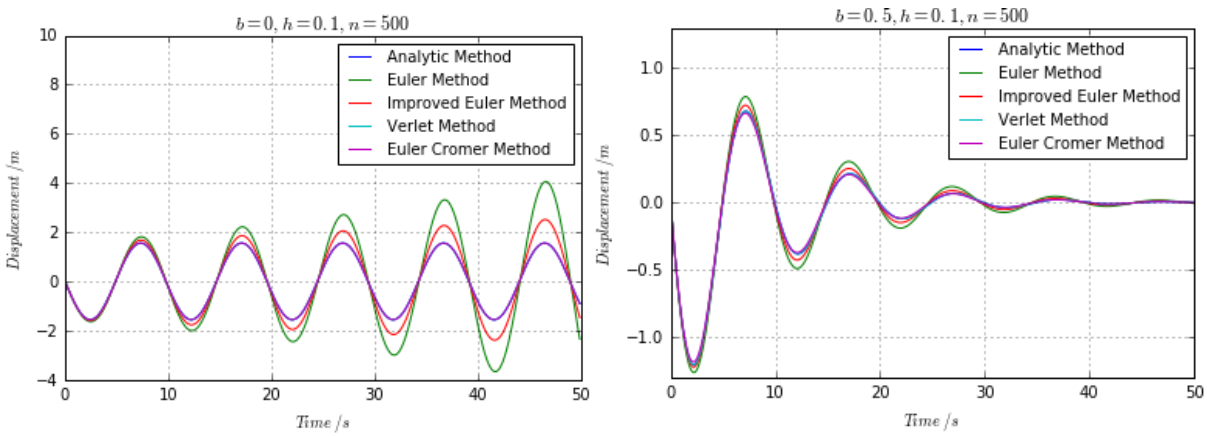

Figure 1: Displacement graphs with damping terms of 0.0 kgs^{-1}(left) and 0.5 kgs^{-1}(right), produced with a step size of 0.1 s for 50 s.

difference curves were plotted for comparison. The numerical method that consistently produced the least energy difference was selected at the best method. This method was used in subsequent parts of the simulation.

The dependence of the accuracy of the best method on its step size was then investigated. Step sizes from 10^{-6} to 1, with each next step size increased by a factor of 10 were used. The energy difference from the analytical solution after a simulation of 10 s was calculated for each step size. The step size that produced the least energy difference was determined as the optimum step size and used in subsequent parts of the simulation.

The phenomenon of light damping, critical damping and heavy damping were then investigated using the best numerical method in 20000 steps. The damping terms used were $0.5b_{critical}$, $b_{critical}$ and $2b_{critical}$. Displacement and energy graphs were plotted to show the solutions.

Next, instantaneous forced oscillations with a damping term of 0.3 kgs^{-1} were simulated with 80000 simulation steps. An instantaneous driving force of magnitude 15 N lasting 0.05 s was chosen for this part of the simulation. It was applied in either the same direction or the negative direction of the instantaneous velocity, at $t = 21\ s$ when the oscillation is approaching its peak point and at $t = 24.6\ s$ when the oscillation is approaching equilibrium point.

Lastly, sinusoidal forced oscillations with a damping term of 0.1 kgs^{-1} were simulated with 200000 steps. A sinusoidal driving force of amplitude 1 N was applied to the oscillator from $t = 0\ s$. The driving frequencies chosen were $0.5w_0$ and $2w_0$. Solutions of the sinusoidal forces oscillations were compared with the unforced oscillation.

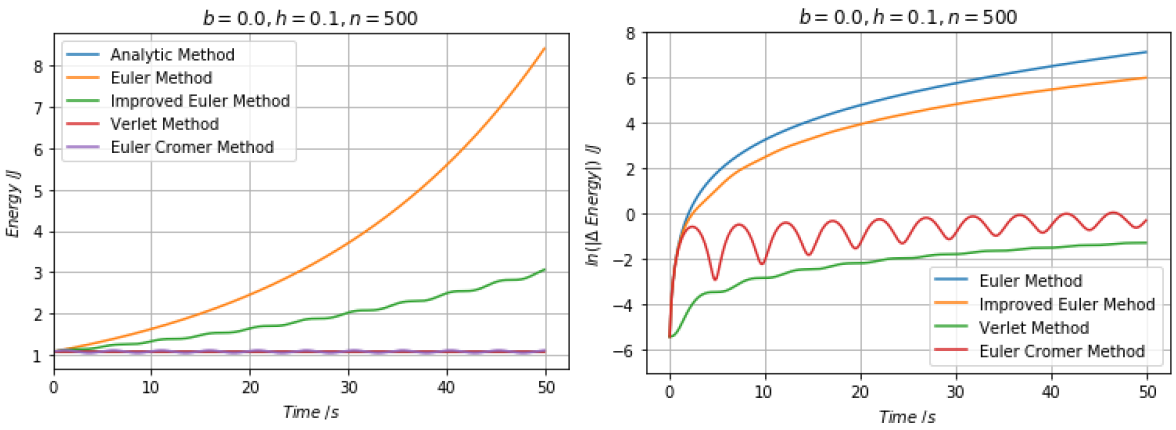

Figure 2: (Left) Graph of energy of oscillator at every point in time with damping term 0.0 kgs^{-1}. (Right) Graph showing total energy difference of each numerical method from analytical solution over time, plotted on a logarithmic scale.

4 Results

4.1 Determining Accuracy of Numerical Methods

The displacement graphs produced by the analytical and numerical methods with damping terms of 0.0 kgs^{-1} and 0.5 kgs^{-1} are shown in Figure 1. It is observed that the least accurate method is the Euler method, followed by the Improved Euler method. Displacement graphs of damping terms 0.3 kgs^{-1}, 2.8 kgs^{-1} and 4.0kgs^{-1} showed similar results.

It was also observed that for a damping term of 0.0 kgs^{-1}, amplitudes calculated by the Euler and Improved Euler methods steadily increases, suggesting an increase in overall energy over time, as confirmed by the energy graph in Figure 2. This can be explained by the fact that the Euler and Improved Euler methods are non-sympletic. The non-sympletic nature of the these two methods are further shown in Figure 3, which shows the phase space graphs of damping terms of 0.0 kgs^{-1} and 0.3 kgs^{-1}.

The energy difference curves of the four numerical methods for a damping term of 0.0 kgs^{-1} are shown in Figure 2. As the energy differences for each numerical method vary over a large range, a logarithmic scale is used for the energy difference. It can be seen that over a period of 50 s, the energy difference curve for the Verlet method is consistently the most negative, corresponding to the smallest energy difference from the analytical energy solution. The next most negative curve is of the Euler-Cromer method, which shows periodic variations as its energy varies periodically over one oscillation. Energy difference graphs for damping terms of 0.3 kgs^{-1}, 0.5 kgs^{-1}, 2.8 kgs^{-1} and 4.0kgs^{-1} showed similar results. The Verlet method was thus selected as the best numerical method.

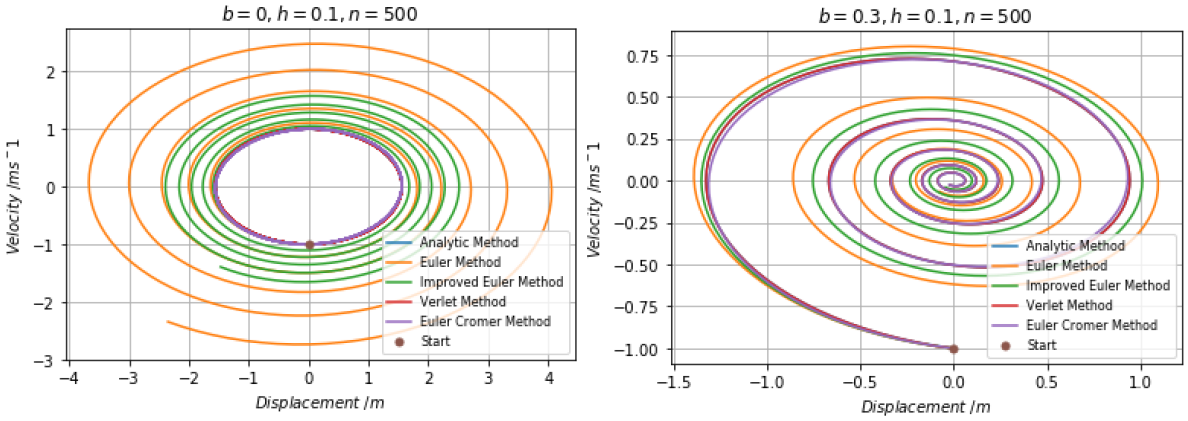

Figure 3: Phase space graphs for damping terms of 0.0 kgs^{-1}(left) and 0.3 kgs^{-1}(right). For damping term of 0.0 kgs^{-1}, the analytical solution is an ellipse which corresponds to conserved energy while the Euler methods spirals outwards with increasing energy.

$h\ (s^{-1})$	$\lvert\Delta E\rvert\ (J)$
1	9.96×10^{-1}
10^{-1}	$5.89\ \times 10^{-2}$
10^{-2}	5.49×10^{-3}
10^{-3}	5.46×10^{-4}
10^{-4}	1.38×10^{-3}
10^{-5}	1.13
10^{-6}	9.97 × 10^2

Table 2: Table showing how total energy difference |ΔE| of the Verlet method from the analytical solution after a simulation of 10 s varies with step size h.

Table 2 shows how accuracy of the Verlet method varies with decreasing step size from 1 to 10^{-6} , for the case of a damping term of 0.0 kgs^{-1}. The accuracy is represented by |ΔE|, the total energy difference of the Verlet method from the analytical energy solution after a simulation of 10 s. A smaller |ΔE| corresponds to a higher accuracy. It was observed that |ΔE| initially decreased as step size decreased, up to until a step size of 10^{-3}. For step sizes of 10^{-4} onwards, |ΔE| increased. This can be explained by the fact that initially a decreasing step size h brought about a decrease in magnitude of truncation error of the Verlet method, increasing its accuracy. However, for step sizes of 10^{-4} and below, the effects of range errors caused by computation become large and caused a decrease in accuracy of the Verlet method. The optimum step size to use for subsequent simulations was thus taken to be 10^{-3} in this case.

4.2 Light, Critical and Heavy Damping

The solutions calculated using the Verlet method for lightly damped, critically damped and heavily damped unforced oscillations are shown in Figure 4 for 20000 simulations steps. Damping terms of $0.5b_{critical}$, $b_{critical}$ and $2b_{critical}$ were used.

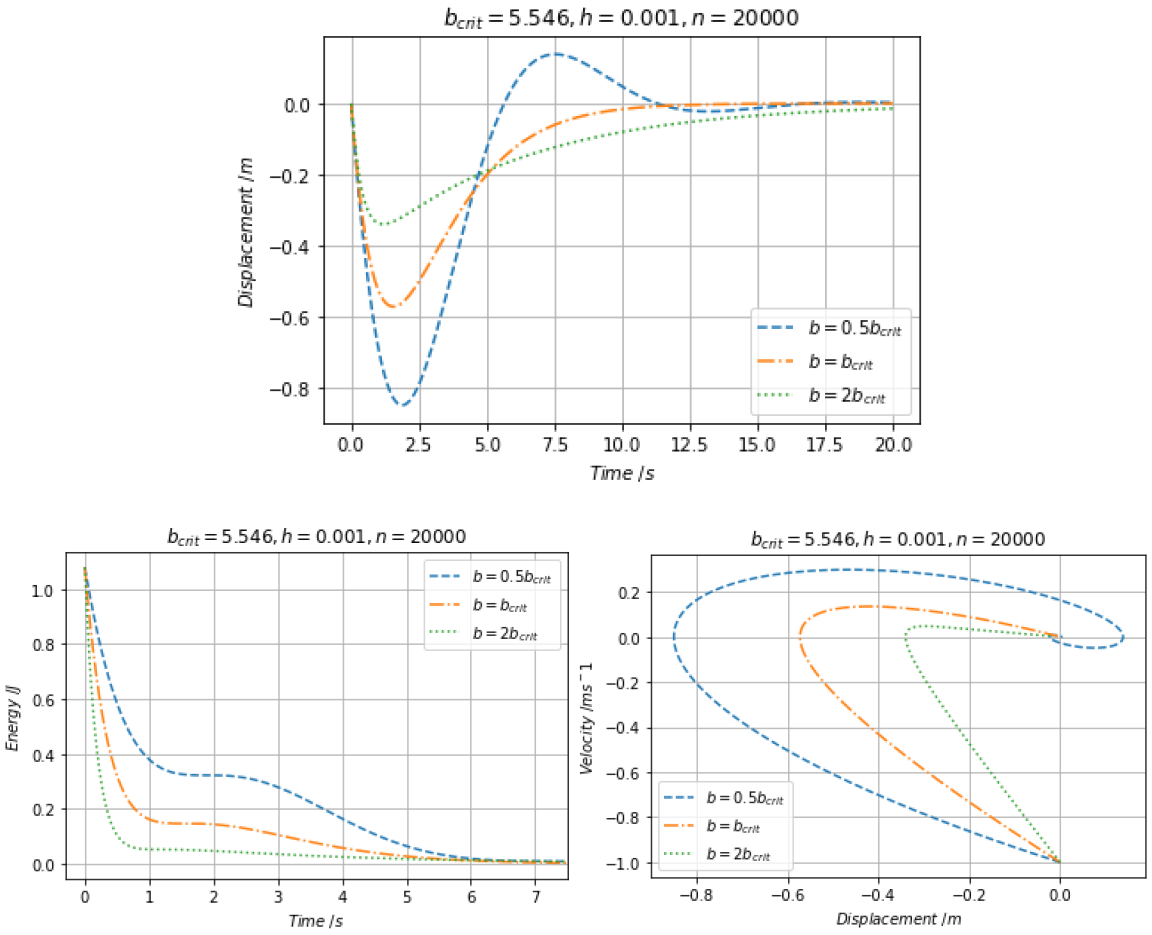

Figure 4: Graphs showing the displacement (upper), energy (lower left) and phase space (lower right) calculated using the Verlet method for damping terms of $0.5b_{critical}$, $b_{critical}$ and $2b_{critical}$. A step size of 0.001 was used for 20000 simulation steps.

The critically damped oscillator was found to return to equilibrium in the shortest time after half a cycle of oscillation, as shown in the phase space graph in Figure 4. The lightly damped oscillator returned to equilibrium after completing at least 1 full cycle of oscillation. The heavily damped oscillator was observed to infinitely approach equilibrium without crossing the equilibirum point at all, as shown in the displacement graph in Figure 4.

Figure 4 also shows that the lightly damped oscillator has the largest magnitude of displacement and the largest energy. It is also worth noting that although the heavily damped oscillator has the smallest magnitude of displacement and smallest energy, it took the longest time to return to equilibrium. In fact it had not returned to equilibrium even after 20 s.

Thus it can be concluded that critically damped oscillators return to equilibirum in the shortest time without further oscillations, lightly damped oscillators return to equilibrium after a few oscillations and heavily damped oscillators approach equilibirum infinitely without further oscillations.

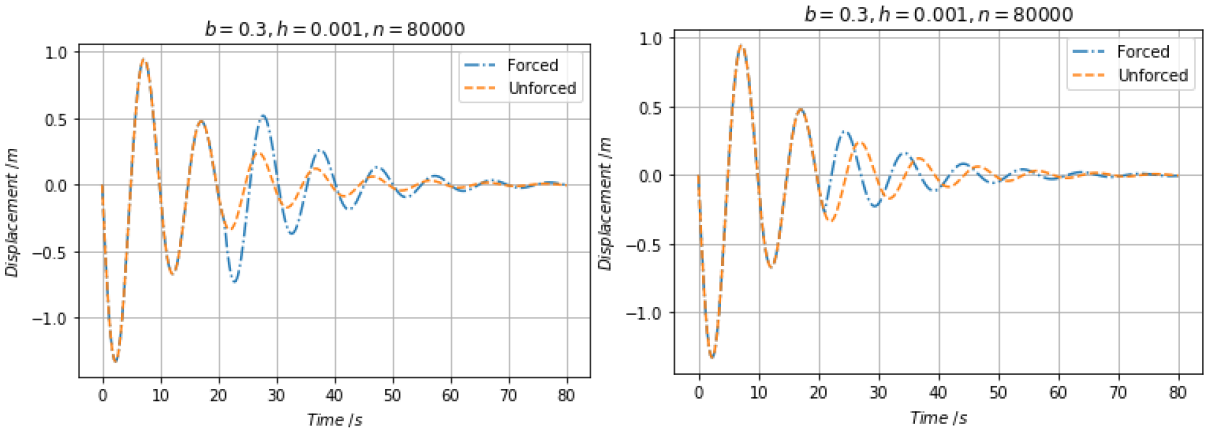

Figure 5: Graphs showing the effects of applying an instantaneous force of 15 N at time 21 s for a duration of 0.05 s on the oscillator in the same direction as the instantaneous velocity (left) and in the opposite direction of the instantaneous velocity (right).

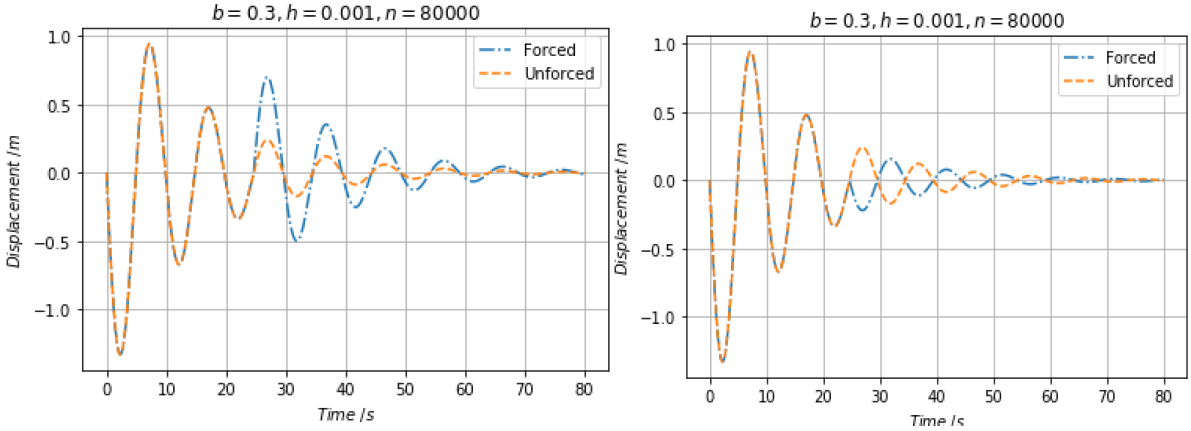

Figure 6: Graphs showing the effects of applying an instantaneous force of 15 N at time 24.6 s for a duration of 0.05 s on the oscillator in the same direction as the instantaneous velocity (left) and in the opposite direction of the instantaneous velocity (right).

4.3 Instantaneous Forced Oscillation

Figures 5 and 6 show the effects of applying an instantaneous external driving force of 15 N on an oscillator for a duration of 0.05 s at different times and in different directions. This was calculated using the Verlet method in 80000 steps. In Figure 5 the instantaneous force was applied when the oscillator was reaching its point of maximum magnitude of displacement. In Figure 6 the instantaneous force was applied when the oscillator was near its equilibrium point.

When the force was applied in the same direction as the instantaneous velocity, the amplitude of oscillation increased and the oscillation took a longer time to die away compared to the unforced case. This suggests that postive work was done on the oscillator, increasing its energy. When the force applied was in the opposite direction to the

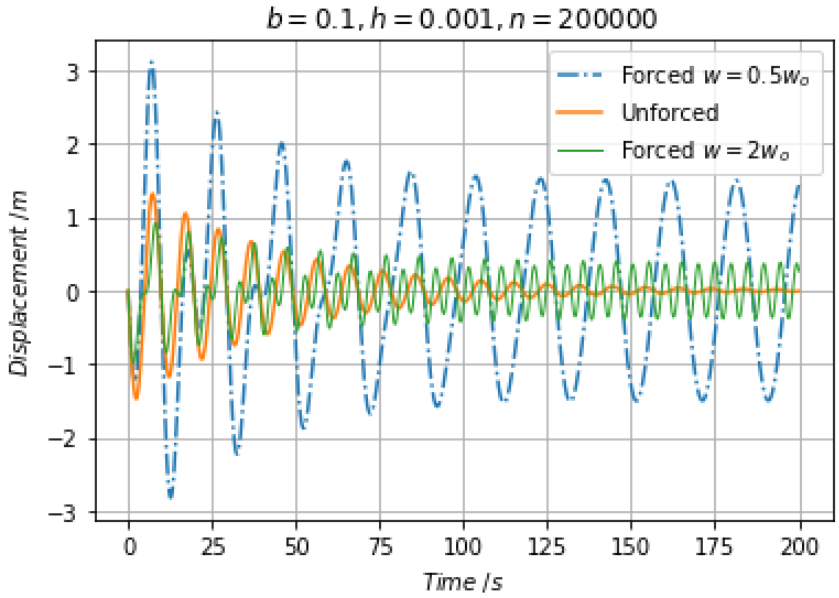

Figure 7: Graph showing displacements of oscillators driven by sinusoidal forces of frequency $0.5w_0$ and $2w_0$. The displacement for the unforced case was also showed for comparison with the sinusoidal forced oscillations.

instantaneous velocity, the amplitude of oscillation decreased, suggesting that negative work was done on the oscillator.

In all cases the oscillator oscillated at a constant frequency and maintained a constant phase differnence with the unforced oscillation solution after the driving force ceased to apply. This suggests that an instantaneous driving force does not change the frequency of oscillation and only changes the amplitude or energy of oscillation.

4.4 Sinusoidal Forced Oscillations

Figure 7 shows the displacements of an oscillator driven by sinusoidal forces of amplitude 1 N and at frequencies of 0.5 w_0 (lower than its natural frequency) and 2 w_0 (higher than its natural frequency). The displacements were calculated using the Verlet method in 200000 steps. In each case there was in initial transient period of relatively irregular motion before a steady oscillations took over.

It was observed that the oscillator reached steady state at a faster rate for the case of a lower driving frequency. It took longer time to reach steady state for the case of a higher driving frequency.

It was also observed that at steady state, the oscillator oscillated at the frequency of the driving force instead of its natural frequency.

Thus it was concluded that oscillators driven by a sinusoidal force will oscillate at the driving frequency after a brief transient period.

5 Discussion of Errors

There are two main sources of error in this computer simulated experiment. The first comes from the round-off errors of the computer when performing calculations. This was shown by the results in Table 2, where round off errors decreased the accuracy of the Verlet method when step sizes were too small. The second comes from the truncation errors from the numerical methods which are small but still finite.
Also, the optimum value chosen for step size in later simulations, 10^{-3}, is close to the true optimum value but not exactly the true optimum value. The errors of this experiment can be reduced if an more appropriate optimum value is found. However, due to computer limitations and time limitations, this analysis was not performed too detailedly.

6 Conclusions

Comparison of the Euler, Improved Euler, Verlet and Euler-Cromer methods showed that the most accurate method to model damped harmonic oscillations is the Verlet method. The optimum step size chosen for the Verlet method produced good results that agree with theory. Instantaneous forced oscillations were found to only change the displacment of oscillation but not the frequency of oscillation. Sinusoidal forced oscillations were found to change but the displacement and the frequency of oscillation. After an intitial transient response, sinusoidal oscillators oscillated at the driving frequency.

References

[1] King, G. (2013). *Vibrations and Waves*. New York, NY: John Wiley & Sons.

[2] Garcia, A. (2000). *Numerical Methods for Physics*. 2nd ed. New Jersey: Prentice Hall.

7 Appendix

7.1 Part I

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 15 10:26:26 2018
@author: jiachen

COMPUTATIONAL PHYSICS PROJECT 2
MODELLING SIMPLE HARMONIC MOTION
PART I

INTERACTIVE PROGRAM THAT:
    1.CALCULATES SOLUTIONS OF DAMPED UNFORCED HARMONIC MOTION GIVEN INITIAL
CONDITIONS USING:
        ANALYTICAL METHOD
        EULER METHOD
        IMPROVED EULER METHOD
        VERLET METHOD
        EULER CROMER METHOD
    2.SHOWS ACCURACY OF EACH NUMERICAL METHOD BY COMPARING ENERGY SOLUTIONS
TO ANALYTICAL METHOD
    3.SHOWS EFFECT OF STEP SIZE ON VERLET METHOD
    4.CALCULATES SOLUTIONS OF LIGHT,CRITICAL AND HEAVY DAMPING USING VERLET
METHOD

"""

import numpy as np
import math
import matplotlib.pyplot as plt

#=======================
#      FUNCTIONS
#=======================
def get_user_input(s):
    """
    string -> float
    Get a valid float value for 's' from user input
    """
    userInput = input('Please input a value for %s in SI units' % s)
    while True:
        try:
            float(userInput)
            break
        except:
            userInput = input('Input for %s is not a number. Please re
enter a valid value. ' % s)
    return float(userInput)

def get_degree(m,k,b):
    """
    float, float, float -> int
    Determine degree of damping from b,m,k values
    Returns int that represents degree of damping
    """
    degree = 0
    if b ==0: degree = 0
```

```
    elif b < 2*math.sqrt(m*k): degree = 1
    elif b == 2*math.sqrt(m*k): degree = 2
    else: degree = 3
    return degree

def get_ana(degree,x_i,v_i,t,k,m,b,h,n):
    """
    float,float,np,float,float,float,float,int -> (np,np,np)
    Calculates x,v,E at every step using Analytical Solution
    Returns values x,v,E calculated
    """
    x = np.zeros(n)
    v = np.zeros(n)
    E = np.zeros(n)

    # if free oscillation or light damping
    if degree == 0 or degree==1:
        # calculate amplitude term
        amp = math.sqrt(x_i*x_i + (v_i*v_i*m/k))
        # calculate phase angle term
        if x_i/amp >= 0:
            # fix phi to be within 0 and 2pi
            phi = [math.asin(x_i/amp), math.pi-math.asin(x_i/amp)]
            # compare possible phi values with initial velocity
            if abs(v_i - amp*(math.sqrt(k/m))*math.cos(phi[0])) < 1e-5:
                phi = float(phi[0])
            elif abs(v_i - amp*(math.sqrt(k/m))*math.cos(phi[1])) < 1e-5:
                phi = float(phi[1])
        else:
            # possible phi values between 0 and 2pi
            phi = [math.pi-math.asin(x_i/amp),
2*math.pi+math.asin(x_i/amp)]
            # compare possible phi values with initial velocity
            if abs(v_i - amp*(math.sqrt(k/m))*math.cos(phi[0])) < 1e-5:
                phi = float(phi[0])
            elif abs(v_i - amp*(math.sqrt(k/m))*math.cos(phi[1])) < 1e-5:
                phi = float(phi[1])
        if degree == 0:
            # calculate angular frequency
            w = math.sqrt(k/m)
            # caclculate x,v,E
            for i in range(n):
                x[i] = amp * math.sin( w*t[i] + phi )
                v[i] = amp * w * math.cos( w*t[i] + phi )
                E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
        elif degree == 1:
            # calculate gamma
            g = b/m
            # calculate angular frequency
            w = math.sqrt(k/m - g*g/4)
            # calculate x,v,E
            for i in range(n):
                x[i] = amp * math.exp( -g/2*t[i] ) * math.sin( w*t[i] + phi
)
                v[i] = -g/2 * amp * math.exp(-g/2*t[i]) * math.sin(w*t[i] +
phi) + amp * w * math.exp(-g/2*t[i]) * math.cos(w*t[i] + phi)
                E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
#
    # critical damping
    elif degree == 2:
        # calculate gamma
```

```
        g = b/m
        # calculate A and B term
        A = x_i
        B = v_i + g/2*x_i

        for i in range(n):
            x[i] = A * math.exp(-g/2*t[i]) + B * t[i] * math.exp(-g/2*t[i])
            v[i] = -g/2 * A * math.exp(-g/2*t[i]) + B * math.exp(-g/2*t[i])
- g/2 * B * t[i] * math.exp(-g/2*t[i])
            E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
    else:
        # calculate gamma and alpha
        g = b/m
        a = math.sqrt(g*g/4 - k/m)
        # calculate A and B term
        B = (1/2/a) * ( (-g/2 + a)*x_i - v_i )
        A = x_i - B
        # calculate x,v,E
        for i in range(n):
            x[i] = A*math.exp( (-g/2 + a)*t[i] ) + B*math.exp( (-g/2-
a)*t[i] )
            v[i] = (-g/2+a)*A*math.exp( (-g/2 + a)*t[i] ) -
(g/2+a)*B*math.exp( (-g/2-a)*t[i] )
            E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
    return (x,v,E)

def get_euler(x_i,v_i,t,k,m,b,h,n):
    """
    Calculates x, v, a, E at every step using Euler Method
    Returns values of x, v, a, E calculated
    """
    x = np.zeros(n)
    v = np.zeros(n)
    a = np.zeros(n)
    E = np.zeros(n)
    for i in range(n):
        if i == 0:
            x[i] = x_i
            v[i] = v_i
            a[i] = - (b*v[i]/m) - (k*x[i]/m)
            E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
        else:
            v[i] = v[i-1] + h*a[i-1]
            x[i] = x[i-1] + h*v[i-1]
            a[i] = - (b*v[i]/m) - (k*x[i]/m)
            E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
    return (x,v,a,E)

def get_euler_improved(x_i,v_i,t,k,m,b,h,n):
    """
    Calculates x,v,a,E at every step using Improved Euler Method
    Returns values of x,v,a,E calculated
    """
    x = np.zeros(n)
    v = np.zeros(n)
    a = np.zeros(n)
    E = np.zeros(n)
    for i in range(n):
        if i == 0:
            x[i] = x_i
            v[i] = v_i
```

```
            a[i] = - (b*v[i]/m) - (k*x[i]/m)
            E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
        else:
            v[i] = v[i-1] + h*a[i-1]
            x[i] = x[i-1] + h*v[i-1] + 0.5*h*h*a[i-1]
            a[i] = - (b*v[i]/m) - (k*x[i]/m)
            E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
    return (x,v,a,E)

def get_verlet(x_i,v_i,t,k,m,b,h,n):
    """
    Calculates x,v,a,E at every step using Verlet Method
    Returns values of x,v,a,E calculated
    """
    x = np.zeros(n)
    v = np.zeros(n)
    a = np.zeros(n)
    E = np.zeros(n)
    for i in range(int(n)):
        if i == 0:
            x[i] = x_i
            v[i] = v_i
            a[i] = - (b*v[i]/m) - (k*x[i]/m)
        elif i == 1:    # Improved Euler Method for First Step
            v[i] = v[i-1] + h*a[i-1]
            x[i] = x[i-1] + h*v[i-1] + 0.5*h*h*a[i-1]
            a[i] = - (b*v[i]/m) - (k*x[i]/m)
        else:           # Verlet Method for Second Step Onwards
            x[i] = 2*(2*m-k*h*h)/(2*m+b*h)*x[i-1] + (b*h-
2*m)/(2*m+b*h)*x[i-2]
            if i >= 3:  # can only calculate v[i] when both x[i+1] and x[i-
1] is known
                v[i-1] = (x[i] - x[i-2])/2/h
            if i == n-1: # need to calculate extra point of x if want to
find velocity of last point
                v[i] = ((2*(2*m-k*h*h)/(2*m+b*h)*x[i] + (b*h-
2*m)/(2*m+b*h)*x[i-1])-x[i-1])/2/h
    for i in range(n):
        E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
    return (x,v,a,E)

def get_euler_cromer(x_i,v_i,t,k,m,b,h,n):
    """
    Calculates x,v,a,E at every step using Euler Cromer Method
    Returns values of x,v,a,E calculated
    """
    x = np.zeros(n)
    v = np.zeros(n)
    a = np.zeros(n)
    E = np.zeros(n)
    for i in range(n):
        if i == 0:
            x[i] = x_i
            v[i] = v_i
            a[i] = - (b*v[i]/m) - (k*x[i]/m)
            E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
        else:
            v[i] = v[i-1] + h*a[i-1]
            x[i] = x[i-1] + h*v[i]
            a[i] = - (b*v[i]/m) - (k*x[i]/m)
            E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
```

```
    return (x,v,a,E)

def E_sum(E):
    """
    np array -> np array
    Calcualte E_sum[i] = E[0]+E[1]+E[2]+...+E[i]
    Returns all values of E_sum[i] calculated
    """
    E_sum = np.zeros(len(E))
    for i in range(len(E)):
        if i == 0:
            E_sum[i] = E[i]
        else:
            E_sum[i] = E[i] + E_sum[i-1]
    return E_sum

def get_t(h,n):
    """
    Calculate the time values from steps and step size
    """
    t = np.zeros(n)
    for i in range(n):
        if i==0:
            t[i] = 0
        else:
            t[i] = t[i-1] + h
    return t

#=======================
#    INTRODUCTION
#=======================
text = 'This program calculates the solutions of damped harmonic motion of
an oscillator using  analytical and other numerical methods, if given the
initial conditions of displacement, velocity and values for spring
constant, mass of object, damping term. Also needed are the values of step
size and number of steps taken. A step size of 0.001 is recommended.'
print(text)

#=======================
#   INITIAL CONDITIONS
#=======================
k = get_user_input('Spring Constant, k')
m = get_user_input('Mass of Object, m')
b = get_user_input('Damping Term, b')
x_i = get_user_input('Initial Displacement, x')
v_i = get_user_input('Initial Velocity, v')
h = get_user_input('Steph Size, h')
n = int(get_user_input('Number of Steps, n'))

#====================================
#  DETERMINE DEGREE OF DAMPING
# i.e. free, light, critical or heavy
#====================================
degree = get_degree(m,k,b)      # 0-free 1-light 2-critical 3-heavy

#=========================
#   CALCULATE TIME VALUES
#=========================
t = get_t(h,n)

#=======================
```

```
#  CALCULATE SOLUTIONS
#=======================
# get x values by analytical solution
(x_ana, v_ana, E_ana) = get_ana(degree,x_i,v_i,t,k,m,b,h,n)
# write to file
data_ana = np.array([x_ana, v_ana, E_ana]).T  # transpose into columns
with open('Analytical Solution Data', 'w+') as file_object:
    np.savetxt(file_object, data_ana, fmt=['%f','%f','%f'])

# get x values by eulor method
(x_euler, v_euler, a_euler, E_euler) = get_euler(x_i,v_i,t,k,m,b,h,n)
# write to file
data_euler = np.array([x_euler, v_euler, a_euler, E_euler]).T
with open('Euler Solution Data', 'w+') as file_object:
    np.savetxt(file_object, data_euler, fmt=['%f','%f','%f','%f'])

# get x values by improved eulor method
(x_euler_improved, v_euler_improved, a_euler_improved, E_euler_improved) =
get_euler_improved(x_i,v_i,t,k,m,b,h,n)
# write to file
data_euler_improved = np.array([x_euler_improved, v_euler_improved,
a_euler_improved, E_euler_improved]).T
with open('Improved Euler Solution Data', 'w+') as file_object:
    np.savetxt(file_object, data_euler_improved, fmt=['%f','%f','%f','%f'])

# get x values by verlet method
(x_verlet, v_verlet, a_verlet, E_verlet) = get_verlet(x_i,v_i,t,k,m,b,h,n)
# write to file
data_verlet = np.array([x_verlet, v_verlet, a_verlet, E_verlet]).T
with open('Verlet Solution Data', 'w+') as file_object:
    np.savetxt(file_object, data_verlet, fmt=['%f','%f','%f','%f'])

# get x values by eulor cromer method
(x_euler_cromer, v_euler_cromer, a_euler_cromer, E_euler_cromer) =
get_euler_cromer(x_i,v_i,t,k,m,b,h,n)
# write to file
data_euler_cromer = np.array([x_euler_cromer, v_euler_cromer,
a_euler_cromer, E_euler_cromer]).T
with open('Euler Cromer Solution Data', 'w+') as file_object:
    np.savetxt(file_object, data_euler_cromer, fmt=['%f','%f','%f','%f'])

#====================================
#        DISPLACEMENT
#====================================
# displacement against time for every method used
plt.plot(t,x_ana,label='Analytic Method')
plt.plot(t,x_euler, label='Euler Method')
plt.plot(t,x_euler_improved, label='Improved Euler Method')
plt.plot(t,x_verlet, label='Verlet Method')
plt.plot(t,x_euler_cromer, label='Euler Cromer Method')
plt.grid()
plt.legend( fontsize=10)
plt.title('$b=%.1f, h=%.1f, n=%i$' % (b,h,n))
plt.ylabel('$Displacement$ $/m$')
plt.xlabel('$Time$ $/s$')
plt.show()

#====================================
#       PHASE SPACE
#====================================
# read in data from files
```

```
data_ana = np.loadtxt('Analytical Solution Data')
(x_ana, v_ana, E_ana) = (data_ana[:,0], data_ana[:,1], data_ana[:,2])

data_euler = np.loadtxt('Euler Solution Data')
(x_euler, v_euler, E_euler) = (data_euler[:,0], data_euler[:,1],
data_euler[:,3])

data_euler_improved = np.loadtxt('Improved Euler Solution Data')
(x_euler_improved, v_euler_improved, E_euler_improved) =
(data_euler_improved[:,0], data_euler_improved[:,1],
data_euler_improved[:,3])

data_verlet = np.loadtxt('Verlet Solution Data')
(x_verlet, v_verlet, E_verlet) = (data_verlet[:,0], data_verlet[:,1],
data_verlet[:,3])

data_euler_cromer = np.loadtxt('Euler Cromer Solution Data')
(x_euler_cromer, v_euler_cromer, E_euler_cromer) = (data_euler_cromer[:,0],
data_euler_cromer[:,1], data_euler_cromer[:,3])

# plot phase space
plt.plot(x_ana, v_ana, label='Analytic Method')
plt.plot(x_euler, v_euler, label='Euler Method')
plt.plot(x_euler_improved, v_euler_improved, label='Improved Euler Method')
plt.plot(x_verlet, v_verlet, label='Verlet Method')
plt.plot(x_euler_cromer, v_euler_cromer, label='Euler Cromer Method')
plt.legend(loc='lower right', fontsize=8)
plt.title('$b=%.1f, h=%.1f, n=%i$' % (b,h,n))
plt.xlabel('$Displacement$ $/m$')
plt.ylabel('$Velocity$ $/ms^-1$')
plt.grid()
plt.show()

#============================
#          ENERGY
#============================
# plot energy against time for every method
plt.plot(t, E_ana, label='Analytic Method')
plt.plot(t, E_euler, label='Euler Method')
plt.plot(t, E_euler_improved, label='Improved Euler Method')
plt.plot(t, E_verlet, label='Verlet Method')
plt.plot(t, E_euler_cromer, label='Euler Cromer Method')
plt.title('$b=%.1f, h=%.1f, n=%i$' % (b,h,n))
plt.xlabel('$Time$ $/s$')
plt.ylabel('$Energy$ $/J$')
plt.xlim(0)
plt.grid()
plt.legend()
plt.show()


#================================
#   SUMMED ENERGY DIFFERENCE
#================================
# calculate the total summ of energy from every previous point in time
E_ana_sum = E_sum(E_ana)
E_euler_sum = E_sum(E_euler)
E_euler_improved_sum = E_sum(E_euler_improved)
E_verlet_sum = E_sum(E_verlet)
E_euler_cromer_sum = E_sum(E_euler_cromer)
```

```
# plot summed energy difference graphs
# ln(summed energy of numerical method - summed energy of analytic
solution) against time
plt.plot(t, np.log(abs(E_euler_sum - E_ana_sum)),  label='Euler Method')
plt.plot(t, np.log(abs(E_euler_improved_sum - E_ana_sum)), label='Improved
Euler Mehod')
plt.plot(t, np.log(abs(E_verlet_sum - E_ana_sum)), label='Verlet Method')
plt.plot(t, np.log(abs(E_euler_cromer_sum - E_ana_sum)), label='Euler
Cromer Method')
plt.grid()
plt.legend( fontsize=10)
plt.title('$b=%.1f, h=%.1f, n=%i$' % (b,h,n))
plt.ylabel('$ln(|\Delta$ $Energy|$) $/J$')
plt.xlabel('$Time$ $/s$')
plt.show()

""" OPTIONAL CODE TO SEE NUMERICAL VALUES OF SUMMED ENERGY DIFFERENCE AFTER
N STEPS
# print numerical value E_method_sum - E_ana_sum after n steps
print('After %i steps, b = %f, h = %f,' % (n,b,h))
print('abs(E_euler_sum - E_ana_sum) =', (abs(E_euler_sum - E_ana_sum))[-1])
print('abs(E_euler_improved_sum - E_ana_sum) =', (abs(E_euler_improved_sum
- E_ana_sum))[-1])
print('abs(E_verlet_sum - E_ana_sum) =', (abs(E_verlet_sum - E_ana_sum))[-
1])
print('abs(E_euler_cromer_sum - E_ana_sum) =', (abs(E_euler_cromer_sum -
E_ana_sum))[-1])
"""

# Note: This Part takes Some Time to Calculate. Can be Left out
#=============================================================
#       DETERMINING ACCURACY OF STEP SIZES
#=============================================================
# determine how varying step size affects accuracy of verlet method
# Note: Use Zero Damping for This Part
# step sizes used : 1, 0.1,0.01,0.001,0.0001,0.00001,0.000001'
# accuracy : difference of total sum of E of numerical solution from that
that of analytical solution after 10s

# store energy sum for different h values after 10s
E_verlet_sum_h = []

# store energy sum for different h values after 10s
E_ana_sum_h = []

# calculate energy sum values for verlet method
for i in [1, 0.1,0.01,0.001,0.0001,0.00001,0.000001]:
    h = i
    n = int(10/i)
    E_verlet = get_verlet(x_i,v_i,t,k,m,b,h,n)[3]
    E_verlet_sum = E_sum(E_verlet)
    E_verlet_sum_h.append(E_verlet_sum[-1])

# calculate energy sum values for analytical method
for i in [1, 0.1,0.01,0.001,0.0001,0.00001,0.000001]:
    E_ana_sum_h.append( E_ana[0] * (10/i) )

print('h : 1, 0.1,0.01,0.001,0.0001,0.00001,0.000001')
print('Energy Difference after 10s :', np.array(E_verlet_sum_h) -
np.array(E_ana_sum_h))
```

```
#====================================
#    SOLUTION GRAPHS FOR DIFFFERENT
#        DAMPING TERMS
#====================================
# Verlet Method, Step Size = 0.001, Steps = 20000
h = 0.001
n = 20000
t = get_t(h,n)
b_crit = 2 * math.sqrt(k * m)

# Calculate Solutions
# Lightly Damped Oscillation, b = 0.5*b_crit
b = 0.5 * b_crit
(x_light, v_light, a_light, E_light) = get_verlet(x_i,v_i,t,k,m,b,h,n)
# Critically Damped Oscillation, b = b_crit
b = b_crit
(x_crit, v_crit, a_crit, E_crit) = get_verlet(x_i,v_i,t,k,m,b,h,n)
# Heavily Damped Oscillation, b = 2*b_crit
b = 2 * b_crit
(x_heavy, v_heavy, a_heavy, E_heavy) = get_verlet(x_i,v_i,t,k,m,b,h,n)

# Plot Displacement Graphs
plt.plot(t, x_light, '--', label='$b=0.5b_{crit}$' )
plt.plot(t, x_crit, '-.', label='$b=b_{crit}$')
plt.plot(t, x_heavy, ':', label = '$b=2b_{crit}$')
plt.title('$b_{crit}=%.3f, h=%.3f, n=%i$' % (b,h,n))
plt.xlabel('$Time$ $/s$')
plt.ylabel('$Displacement$ $/m$')
plt.grid()
plt.legend()
plt.show()

# Plot Energy Graphs
plt.plot(t, E_light, '--', label='$b=0.5b_{crit}$' )
plt.plot(t, E_crit, '-.', label='$b=b_{crit}$')
plt.plot(t, E_heavy, ':', label = '$b=2b_{crit}$')
plt.title('$b_{crit}=%.3f, h=%.3f, n=%i$' % (b,h,n))
plt.xlabel('$Time$ $/s$')
plt.ylabel('$Energy$ $/J$')
plt.grid()
plt.legend()
plt.show()

# Plot Phase Space
plt.plot(x_light, v_light, '--', label='$b=0.5b_{crit}$' )
plt.plot(x_crit, v_crit, '-.', label='$b=b_{crit}$')
plt.plot(x_heavy, v_heavy, ':', label = '$b=2b_{crit}$')
plt.title('$b_{crit}=%.3f, h=%.3f, n=%i$' % (b,h,n))
plt.xlabel('$Displacement$ $/m$')
plt.ylabel('$Velocity$ $/ms^-1$')
plt.grid()
plt.legend()
plt.show()
```

7.2 Part II

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 15 18:01:39 2018

@author: aiyuan
COMPUTATIONAL PHYSICS PROJECT 2
MODELLING SIMPLE HARMONIC MOTION
PART II

PROGRAM THAT CALCULATES THE SOLUTIONS OF DAMPED HARMONICE MOTION
WITH AN INSTANTANEOUS FORCE APPLIED AT DIFFERENT PHASES OF THE MOTION
USING THE VERLET METHOD AND STEP SIZE 0.001
"""

import math
import numpy as np
import matplotlib.pyplot as plt

#==============
#  Functions
#==============
def get_verlet(x_i,v_i,t,k,m,b,h,n,F_inst,t_inst, F_duration):
    """
    Calculates x,v,a,E at every step using Verlet Method
    Returns values of x,v,a,E calculated
    """
    x = np.zeros(n)
    v = np.zeros(n)
    a = np.zeros(n)
    E = np.zeros(n)
    F = np.zeros(n)
    F[int(t_inst/h) : int(t_inst/h) + int(F_duration/h)] = F_inst # assign
instantaneous force
    for i in range(int(n)):
        if i == 0:
            x[i] = x_i
            v[i] = v_i
            a[i] = - (b*v[i]/m) - (k*x[i]/m) + F[i]/m
        elif i == 1:    # Improved Euler Method for First Step
            v[i] = v[i-1] + h*a[i-1]
            x[i] = x[i-1] + h*v[i-1] + 0.5*h*h*a[i-1]
            a[i] = - (b*v[i]/m) - (k*x[i]/m) + F[i]/m
        else:           # Verlet Method for Second Step Onwards
            x[i] = 2*(2*m-k*h*h)/(2*m+b*h)*x[i-1] + (b*h-
2*m)/(2*m+b*h)*x[i-2] + 2*F[i-1]*h*h/(2*m+b*h)
            if i >= 3:  # can only calculate v[i] when both x[i+1] and x[i-
1] is known
                v[i-1] = (x[i] - x[i-2])/2/h
    for i in range(n):
        E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
    return (x,v,a,E)

def get_t(h,n):
    """
    Calculate the time values from steps and step size
    """
    t = np.zeros(n)
    for i in range(n):
```

```
        if i==0:
            t[i] = 0
        else:
            t[i] = t[i-1] + h
    return t

#=======================
#   INITIAL CONDITIONS
#=======================
k = 0.89
m = 2.16
b = 0.3
x_i = 0
v_i = -1
h = 0.001
n = 80000
F_inst = 0      # Force applied
t_inst = 0   # Time instant when force applied
F_duration = 0     # Duration force is applied for

#==========================
#   CALCULATE TIME VALUES
#==========================
t = get_t(h,n)

#==========================
#  CALCULATE SOLUTIONS
#==========================
x_unforced = get_verlet(x_i,v_i,t,k,m,b,h,n,F_inst,t_inst,F_duration)[0]

F_inst = -15
t_inst = 21
F_duration = 0.05

x_forced = get_verlet(x_i,v_i,t,k,m,b,h,n,F_inst,t_inst,F_duration)[0]

#======================
#  PLOT SOLUTIONS
#======================
plt.plot(t, x_forced, '-.', label = 'Forced')
plt.plot(t, x_unforced, '--', label = 'Unforced')
plt.grid()
plt.legend()
plt.title('$b=%.1f, h=%.3f, n=%i$' % (b,h,n))
plt.xlabel('$Time$ $/s$')
plt.ylabel('$Displacement$ $/m$')
```

7.3 Part III

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 15 18:01:39 2018

@author: aiyuan
COMPUTATIONAL PHYSICS PROJECT 2
MODELLING SIMPLE HARMONIC MOTION
PART III
```

```
PROGRAM THAT CALCULATES THE SOLUTIONS OF DAMPED HARMONICE MOTION
WITH AN SINUSOIDAL FORCE APPLIED
USING THE VERLET METHOD AND STEP SIZE 0.001
"""

import math
import numpy as np
import matplotlib.pyplot as plt

#==============
#  Functions
#==============
def get_verlet(x_i,v_i,t,k,m,b,h,n,F_inst,t_inst,w):
    """
    Calculates x,v,a,E at every step using Verlet Method
    Returns values of x,v,a,E calculated
    """
    x = np.zeros(n)
    v = np.zeros(n)
    a = np.zeros(n)
    E = np.zeros(n)
    F = np.zeros(n)
    # calculate sinusoidal force
    for i in range(int(n)):
        if i >= int(t_inst/h):
            F[i] = F_inst * math.sin(w*t[i])
    # caculate displacement
    for i in range(int(n)):
        if i == 0:
            x[i] = x_i
            v[i] = v_i
            a[i] = - (b*v[i]/m) - (k*x[i]/m) + F[i]/m
        elif i == 1:     # Improved Euler Method for First Step
            v[i] = v[i-1] + h*a[i-1]
            x[i] = x[i-1] + h*v[i-1] + 0.5*h*h*a[i-1]
            a[i] = - (b*v[i]/m) - (k*x[i]/m) + F[i]/m
        else:            # Verlet Method for Second Step Onwards
            x[i] = 2*(2*m-k*h*h)/(2*m+b*h)*x[i-1] + (b*h-
2*m)/(2*m+b*h)*x[i-2] + 2*F[i-1]*h*h/(2*m+b*h)
            if i >= 3:  # can only calculate v[i] when both x[i+1] and x[i-
1] is known
                v[i-1] = (x[i] - x[i-2])/2/h
    for i in range(n):
        E[i] = 0.5*m*v[i]*v[i] + 0.5*k*x[i]*x[i]
    return (x,v,a,E,F)

def get_t(h,n):
    """
    Calculate the time values from steps and step size
    """
    t = np.zeros(n)
    for i in range(n):
        if i==0:
            t[i] = 0
        else:
            t[i] = t[i-1] + h
    return t

#======================
#    INITIAL CONDITIONS
```

```
#=========================
k = 0.89
m = 2.16
b = 0.1
x_i = 0
v_i = -1
h = 0.001
n = 200000
F_inst = 0      # Amplitude of Sinusoidal Force applied
t_inst = 0   # Time instant when force applied
w = 0        # Frequency of Sinusoidal Force
#============================
#   CALCULATE TIME VALUES
#============================
t = get_t(h,n)

#=========================
#  CALCULATE SOLUTIONS
#=========================
x_unforced = get_verlet(x_i,v_i,t,k,m,b,h,n,F_inst,t_inst,w)[0]

F_inst = 1
t_inst = 0

w = 0.5*math.sqrt(k/m)
x_forced_half = get_verlet(x_i,v_i,t,k,m,b,h,n,F_inst,t_inst,w)[0]

w = 2*math.sqrt(k/m)
x_forced_double = get_verlet(x_i,v_i,t,k,m,b,h,n,F_inst,t_inst,w)[0]

#======================
#  PLOT SOLUTIONS
#======================
plt.plot(t, x_forced_half, '-.', label = 'Forced $w=0.5 w_o$')
plt.plot(t, x_unforced, '-', label = 'Unforced')
plt.plot(t, x_forced_double, '-', linewidth=1, label = 'Forced $w=2w_o$')
plt.grid()
plt.legend()
plt.title('$b=%.1f, h=%.3f, n=%i$' % (b,h,n))
plt.xlabel('$Time$ $/s$')
plt.ylabel('$Displacement$ $/m$')
```