# Numerical Integration of Differential Equations: The Damped Harmonic Oscillator

*Tomasz Neska*

*10294857*

School of Physics and Astronomy

The University of Manchester

Second Year Computational Physics Report

April 2020

## Abstract

An analysis of four numerical integration methods has been conducted. It was done with a purpose of determining the method with the highest accuracy. The Verlet method was found to be most accurate for any time step used. The three other methods used were Euler’s method, Improved Euler’s method and Euler-Cromer method. The Verlet method and the Euler-Cromer method were found to be symplectic. The relationship between the size of a time step and the accuracy has been observed to be inversely proportional. Secondly the investigation into the effects of heavy damping onto oscillation has been conducted. This was done utilising the Verlet method. Lastly the investigation of an application of a constant and sinusoidal force onto an oscillator was conducted. This naturally led onto the investigation of resonance for a forced oscillator and the effect of the value of the damping coefficient on the resonance curve. It was found that heavy damping eliminates resonance.

## Introduction

Simple harmonic motion (SHM) is a type of motion that occurs in every area of physics where a small oscillation around a potential well occurs. This means that this type of motion is present in nearly every field of physics due to the presence of energy potentials. The applications of SHM can range from the description of a quantum particle to the behaviour of a pendulum for small displacements [1].

Simple harmonic motion is specified as a motion of a particle of mass *m* that is subject to a force that depends on its displacement. Hence the characterising property of simple harmonic motion is that

|  |  |  |
| --- | --- | --- |
|  | *.* | Equation 1 |

For *a* being acceleration of a particle and *x* being the displacement from the point of equilibrium. It is a motion in one dimension along the line with the position *x* being the magnitude of the displacement from the equilibrium position. The acceleration of the particle *a* is obtained from the application of Newton’s second law and the addition of vector forces that act on the particle. This system is going to be the baseline on which the investigation into numerical methods is going to be conducted. The method of an iterative integrator is going to be explored. The effects of its predictions on the energy of the system are going to be utilized to judge the accuracy of the methods.

## Theory

If you consider a particle in a spring-mass system with a damping force being dependent on the velocity of the object it is easy to derive that the equation of motion [2] is given by

|  |  |  |
| --- | --- | --- |
|  | *.* | Equation 2 |

This equation of motion represents a damped simple harmonic oscillator with *m* being the mass of the particle, *b* being the damping coefficient and *k* being the spring constant. The single dot represents a derivative with respect to time. This equation has 3 analytic solutions depending on the values of the constants. Those are: heavily damped oscillator, critically damped oscillator and lightly damped oscillator. A fourth solution exists if the damping constant equals 0. That solution is just the solution of a simple harmonic oscillator given by

|  |  |  |
| --- | --- | --- |
|  |  | Equation 3 |

with being the natural angular frequency of the oscillation which equals to . This solution if multiplied by a damping term gives the damped oscillator solution. However, the angular frequency is given by

|  |  |  |
| --- | --- | --- |
|  | *.* | Equation 4 |

The solutions of the damped simple harmonic oscillator depends on the value of . If the solution is heavily damped, but if < 0 the solution is lightly damped. However, if then the solution is critically damped and reaches the equilibrium point the fastest without any oscillation.

Due to the use of iteration it should be said that the acceleration of a particle at any point of the iteration is given by Equation 2 in the form

|  |  |  |
| --- | --- | --- |
|  |  | Equation 5 |

Furthermore, the energy of the particle at any given velocity and position is obtained by

|  |  |  |
| --- | --- | --- |
|  | *.* | Equation 6 |

The approximation described in section 3 is utilised during the investigation into a constant force being applied for a given time period. This approximation is given by

|  |  |  |
| --- | --- | --- |
|  | *.* | Equation 7 |

For being the frequency of the force acting on the particle and being its amplitude. For the case of a constant force the sinusoidal force expression is replaced with a constant.

### 2.1 Euler’s method

The first numerical method is Euler’s method which is defined by

|  |  |  |
| --- | --- | --- |
|  | and | Equation 8 |
|  | . | Equation 9 |

The variable *h* (in seconds) is the time step chosen for the simulation. [3]

This method was found to not be a symplectic integrator. This can be confirmed by the application of Equation 7 and 8 with Equation 9. This obtains

|  |  |  |
| --- | --- | --- |
|  | *.* | Equation 10 |

This shows that the energy of the oscillator increases with each step. This gives an error term to the energy of O().

### 2.2 Improved Euler’s method

The second numerical method is an improvement upon Euler’s method. It is defined by

|  |  |  |
| --- | --- | --- |
|  | and | Equation 11 |
|  | . | Equation 12 |

This method also doesn’t conserve energy and the energy error is given by

|  |  |  |
| --- | --- | --- |
|  | *.* | Equation 13 |

With *O(h)* being a polynomial with the dominant term being *h* (for *h* being small)*.* This means that the energy will also be increasing as in the previous method. [4][5]

### 2.3 Euler-Cromer method

The third numerical method is a symplectic integrator which conserves average energy over the whole time period. It is defined by

|  |  |  |
| --- | --- | --- |
|  | and | Equation 14 |
|  | . | Equation 15 |

Expanding the expression for energy we obtain

|  |  |  |
| --- | --- | --- |
|  | *.* | Equation 16 |

The second term averages out to be zero over the complete cycle what allows the energy to be conserved. [6]

### Verlet’s method

The last numerical integrator is also a symplectic integrator which provides an accurate projectile path with not much more computational cost than Euler’s method [7]. It is defined by

|  |  |  |
| --- | --- | --- |
|  | *.* | Equation 17 |

With and . With . All the constants have the previously defined meaning. This method requires two initial positions. The value of is obtained utilising a Taylor expansion up to the second order.

|  |  |  |
| --- | --- | --- |
|  |  | Equation 18 |

This allows for the computation of the positional terms. The velocity of the oscillator is computed utilising the Verlet-Störmer method along with the mean value theorem. The velocities are obtained by

|  |  |  |
| --- | --- | --- |
|  |  | Equation 19 |

This means that the error of the velocity utilising this method is given by the term in the expansion. The velocity is defined utilising an average. It means that the influence of an external force can be evaluated using Equation 7. It is later seen in section 4. [8]

The critical value of the damping factor is given by

|  |  |  |
| --- | --- | --- |
|  |  | Equation 20 |

## Method

The values of the constants were and . The initial position was chosen to be with the initial velocity of . No applied force was used for the first stage of the investigation. The methods have been compared utilising a maximum time of 100 seconds with the damping coefficient being equal to 0. This allows for the observation of the system at constant energy. The value of the time step was varied between 1 second to 0.001 seconds. This allowed for the observation of the relation between the magnitude of the time step and the accuracy of the simulation. The methods were compared visually along with the use of the amount of “fictitious energy” that each simulation generated. These are summarised in Figure 3. Those were then graphed to show the relation between the error and time. [9]

The program possesses an ability to write and load simulations. It does so by the means of a json file format and the imported json library. This was done to assure that the data is readable and can be utilised in different scripts. Furthermore, for the ease of data manipulation an object-oriented approach has been utilised. This was done to ensure maximum easiness in modifying the script later. This was done at the cost to memory usage that this script requires to run. This was addressed by resetting the variables after the results are obtained. This didn’t cause any problems on the machine it was tested but it may cause problems on a weaker system.

The investigation into the behaviour of a constant push force has been done with the use of Equation 7. The position of the oscillator was then plotted against time. The resonance curve was obtained using the same function. Instead of fixing the time during which the sinusoidal force acts it was altered to be the entirety of the running time. The amplitude of the oscillation was then found and plotted against the frequency of the force. The natural frequency of the oscillator was found to be . This gave the time period of The constant force was hence applied at This corresponds to 3, 3.75, and 3.50 of the periods respectively. At these times a constant force of *2 N* was applied. Meanwhile the sinusoidal force was investigated at the frequency of and with the magnitude of *2 N.*

The effects of the alteration of the damping coefficient were tested by using the Verlet method to calculate the position, velocity and energy. Plotting them as functions of time was done with the goal of visualising the behaviour. The damping term values investigated were multiples of the critical damping coefficient obtained by Equation 20. The mass, spring constant and initial conditions were used as before.

## Analysis

### 4.1 Comparison of the numerical methods

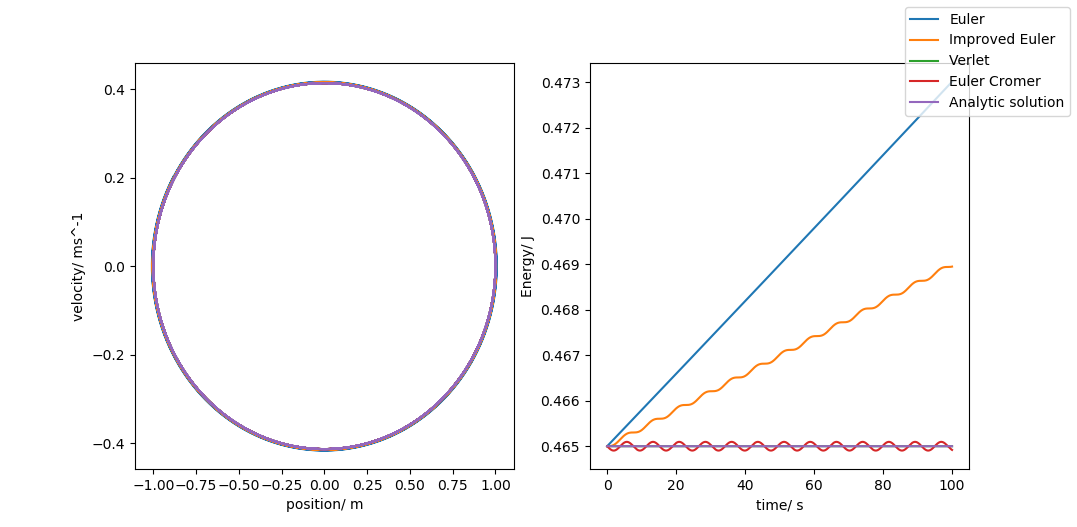
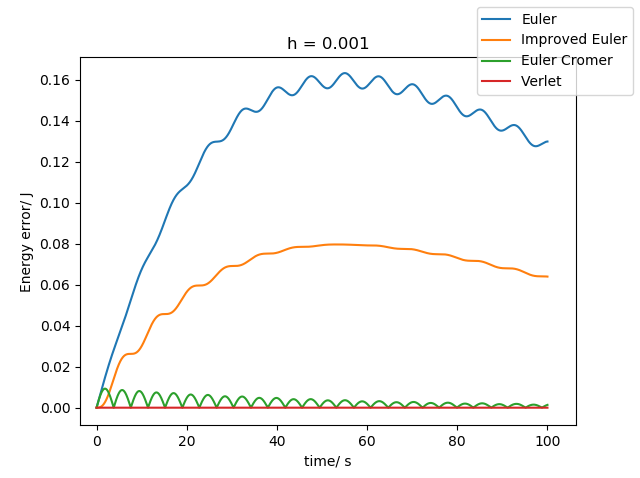
F Figure 1 shows that the Verlet methods gives the best results. This can be further confirmed by the fact that the error in energy generated over the whole runtime was . Meanwhile, the second lowest was the Euler-Cromer method which obtained for all the values tested. it is important to mention that the Improved Euler’s method provided a small error in the range of the damping coefficient passing the critical value. The inaccuracy in the Euler-Cromer method comes from the fact that its energy needs to be averaged over a complete cycle while the Verlet method doesn’t. The testing of simulations running over complete cycles wasn’t tested due to the desire to ensure fairness of the comparison method. The remaining two are not symplectic methods and it can be visually seen that they are not as accurate as the Verlet method in Figure 1 or 2. This means that the Verlet method was utilised for the rest of the investigation with modifications described in section 3.

Figure 2 – Energy error curve. This plot was obtained by subtracting the analytical energy values from the model energy values with the damping coefficient .

Figure 1 – The phase plot of the oscillation (left) and the energy as a function of time (right). This plot was obtained with the time-step being *0.001 s.* The phase plots show the paths obtained from all methods being drawn over each other.

Then the test was conducted for a damped oscillator. It consolidated that the Verlet method has the smallest error out of all the numerical integrators. As evidence by the error being barely visible on the graph. As visible in Figure 1 the error for the Verlet integrator is very close to zero for the entirety of the simulation while other integrators either increase or oscillate around the value. In Figure 2 the energy error curve is the least prominent for the Verlet method under damping as well. The time step value was chosen to be due to it being the best balance between the time taken and the accuracy of the computation. The accuracy increases with the decrease in the step size. This was more apparent for both Euler’s methods due to those two methods having the smallest degree of accuracy. It is important to note that all the methods perform very well when the value of the damping coefficient is large. Verlet’s method produced errors which were not exponential in contrast with Euler’s method. Verlet’s method was also the one that presented the error not increasing exponentially with time. This is very important for long term simulations. Even despite only a second order Taylor expansion used in Equation 18 the accuracy was much higher than anticipated.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| B ( | Euler (J) | Improved Euler (J) | Euler – Cromer (J) | Verlet (J) |
| 0 | 42107.75 | 20448.09 | 609.92 | 0.98 |
| 0.1 | 13530.68 | 6591.90 | 279.68 | 0.48 |
| 1 | 225.95 | 94.98 | 41.51 | 0.08 |
| 2 | 47.52 | 7.78 | 34.82 | 0.07 |
| 3 | 23.80 | 11.49 | 34.84 | 0.08 |

Figure 3 – The table showing the error in the total energy for a given set of damping constants. The values are rounded to two decimal places.

### 4.2 Unforced Oscillations

A step size of *0.001 s* and a maximum time of *100 s* were used to model the unforced simple harmonic oscillator with damping terms half, double and equal to the critical damping coefficient. This was obtained from Equation 20. The value was calculated to be . The energy plots show the expected behaviour. The critically damped oscillator returned to the equilibrium position in the shortest amount of time. The heavily damped oscillator took longer while the lightly damped oscillator moved past the equilibrium point twice. This can be observed in Figure 4. The energy was calculated using Equation 6. The energy decreased the fastest for the critically damped oscillator. This can be observed in Figure 5.

Figure 4 – The position against time plot for the critically damped oscillation with the value of the damping coefficient being half, equal to and double the critical value.

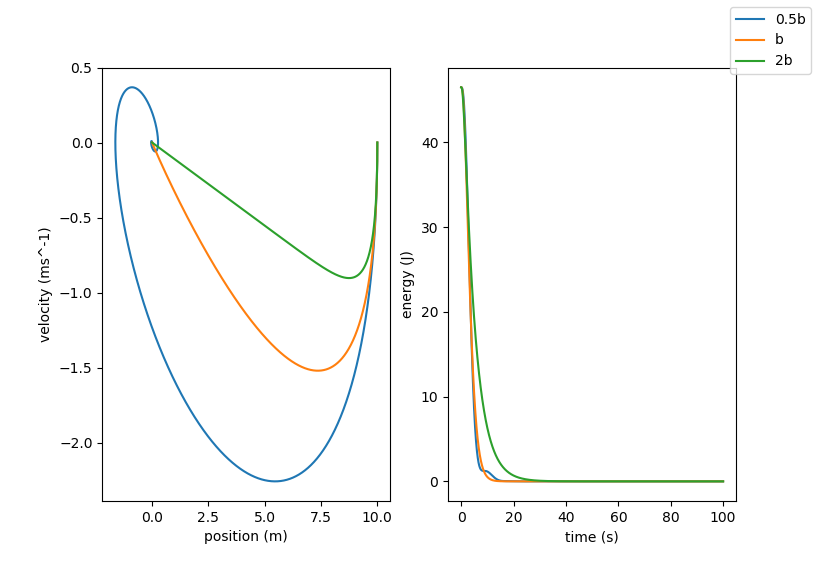
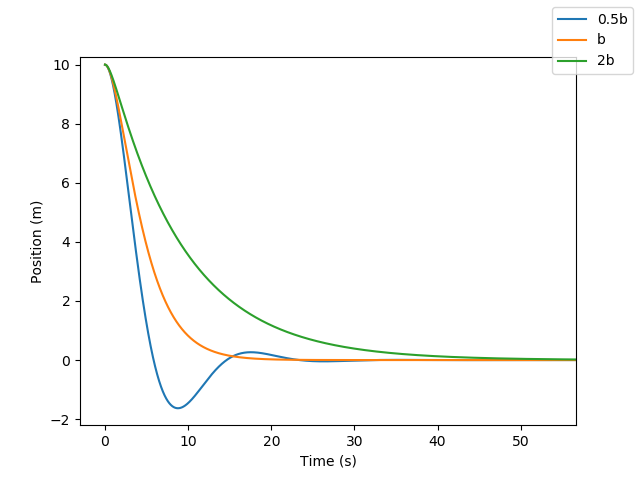


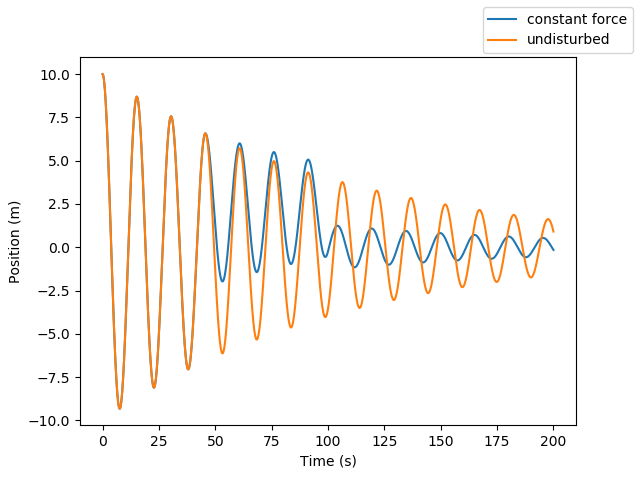
Figure 5 – The phase plot (left) and the energy plot (right) were obtained for the overdamped oscillator.

All the calculations were calculated utilising the Verlet method as it was chosen to be the most accurate as explained in section 4.1. The spiral behaviour presented in Figure 5 can be attributed to the damping forces present in the system arising from the damping coefficient.

### 4.3 Instantaneously Forced Oscillations

The effects of an application of a constant force and a sinusoidal force were investigated. In the case of a constant force the time at which it was applied determined the response. All the oscillations returned to a behaviour explained in section 4.1 after a short transition period. The amplitude changed after the force was applied. When the force was applied at the equilibrium point the amplitude changed the most as can be seen in Figure 6. For the half-cycle and quarter-cycle force applications the effect was weaker, but the amplitude still decreased. Hence the plots were not included.

Figure 6 – The position against time plot shows the change in the oscillation after the constant force was being applied. This force was applied at the equilibrium point.



The sinusoidal force caused an increase in the amplitude around the frequency of the force being equal to the natural frequency of the oscillation. This can be seen in Figure 7. This increase in amplitude occurred only around the natural frequency. If at the moment of the force being applied the direction of the force was in the same direction as the velocity the amplitude increased. Furthermore, the opposite was true as well. The magnitude of the force was found to be not as dominant as expected. Towards the upper values of the force the oscillation still occurred however, reached a rest point much faster. This force had the same effect as shifting the equilibrium point. This is analogous to a suspended spring system where gravity shifts the equilibrium point.

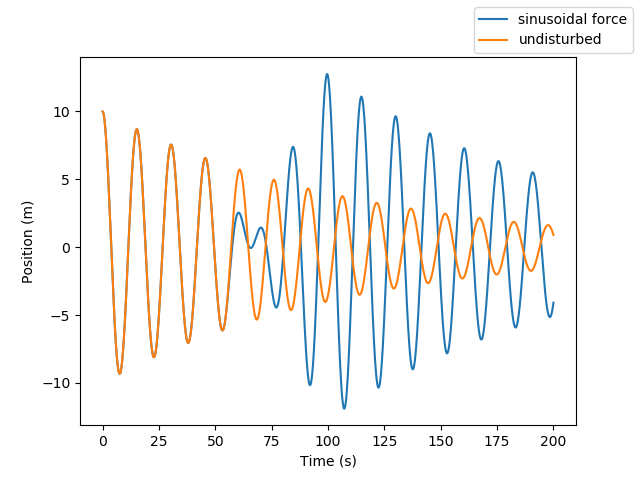


Figure 7 – The position against time plot for a forced oscillator. The force used is applied at the natural frequency of the oscillator hence its amplitude is increased. This is due to resonance.

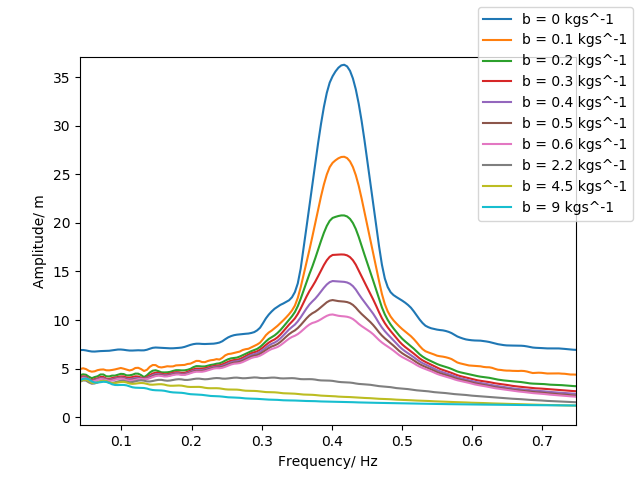


Figure 8 – The collection of resonance plots for different damping coefficients. The time step of was used with the total running time of .

Resonance was observed at the angular frequency calculated in section 3. It was characterised by a sharp increase in the amplitude of the oscillation at the natural frequency. A resonance curve with different damping terms is shown in Figure 8. The value of the damping term increasing caused the lowering of the resonance peak. At the heavy/critical damping the damping coefficient was too high to allow for the resonance to occur. This meant that the motion dissipated too quickly for any oscillation to occur.

## Error Analysis

The main sources of error in this simulation arose from the discretisation error, truncation error, initial conditions and the net growth in the error as the simulation progressed. The discretisation error arises from the fact that the time step at which values are evaluated isn’t infinitesimal but of a fixed value. This was mitigated by the utilisation of a very small step size. The data obtained could have been developed at higher accuracies, but the time constraints didn’t allow for this. The fact that the errors grow as the simulation progresses is since iterations utilise previous values to work out the path of the oscillator. This was mitigated by ensuring that the terms that were omitted, like in the case of a Taylor expansion, were powers of the time step allowing the convergence of the method to happen with minimal error present. The truncation error increased as the step interval decreased.

## Conclusion

The models were utilised and compared as described in section 4.1. The Verlet method was found to be the most accurate for modelling forced and unforced oscillation. The investigation into the effects of the step size onto the accuracy has been conducted. It has shown the inverse proportionality between the step size and the accuracy. This was especially apparent for the less accurate Euler’s method. It can be deduced that the smaller the size of the interval the better the prediction. It is advised to always use the smallest step size possible. However, it is apparent that the decrease of the time interval gives diminishing returns after passing the size.

The application of a constant force onto an oscillator produced a transition period after which the oscillation resumed with a shift in phase and amplitude. The sinusoidal force has been investigated with the use of a resonance plot. It was confirmed that the sharp amplitude increase occurs at the frequency close to the natural frequency of the oscillator. Lastly the Verlet method was utilised to observe the behaviour of the oscillator in the case of being heavily damped and critically damped (as described in section 4.2). It was also confirmed that resonance did not occur when the oscillator was heavily or critically damped.

## References

[1] Young, H. and Freedman, R., 2016. *University Physics With Modern Physics*. 14th ed. Pearson, pp.492-568.

[2] Shankar, R., 2014. *Fundamentals Of Physics I*. 1st ed. Yale University Press.

[3] Atkinson, K., 1989. *An Introduction To Numerical Analysis*. New York: John Wiley & Sons.

[4] Cromer, A., 1981. Stable solutions using the Euler approximation. *American Journal of Physics*, 49(5), pp.455-459.

[5] Süli, E. and Mayers, D., 2014. *An Introduction To Numerical Analysis*. Cambridge: Cambridge University Press.

[6] Giordano, N. and Nakanishi, H., 2006. *Computational Physics*. Upper Saddle River, NJ: Pearson Prentice Hall.

[7] Press, W. and Vetterling, W., 2007. *Numerical Recipes*. Cambridge: Cambridge Univ. Press.

[8] Verlet, L., 1967. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159(1), pp.98-103.

[9] Iott, J., Haftka, R. and Adelman, H., 1985. *Selecting Step Sizes In Sensitivity Analysis By Finite Difference. Technical Memorandum 86382*. NASA.

[10] Neska, T. (2020). *SplitSky/Scientific\_Programming*. [online] GitHub. Available at: https://github.com/SplitSky/Scientific\_Programming [Accessed 20 April. 2020].

## Appendix - Code

The history of changes can be found in the GitHub repository [10].

*# -\*- coding: utf-8 -\*-  
"""  
Author: Tomasz Neska  
Date: 06/03/2020  
Description: Project 2 - utilising three different methods it evaluates the effects of time step and accuracy on the  
behaviour of the simple harmonic oscillator  
"""  
# initialisation***import** string  
**from** math **import** \*  
**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** random  
**import** time  
**import** cmath  
**import** json  
**from** scipy **import** optimize  
  
plt.rcParams.update({**'font.size'**: 14})  
plt.style.use(**'default'**)  
figure = plt.figure()  
plt.rcParams.update({**'errorbar.capsize'**: 2})  
  
  
**class** SHO(object):  
 **def** \_\_init\_\_(self, time\_step, max\_time, b=0.01, m=1.0, k=1.0, init\_x=0.0, init\_v=0.0, fileNameSave=**"data.txt"**,  
 fileNameLoad=**"data.txt"**):  
 *'''  
 parameter name type description* **:param** *time\_step: float the time step used in calculations* **:param** *max\_time: float the time the simulation lasts* **:param** *b: float damping coefficient* **:param** *m: float mass of the oscillator* **:param** *k: float spring constant* **:param** *init\_x: float initial position* **:param** *init\_v: float initial velocity* **:param** *fileNameSave: string filename used for saving data* **:param** *fileNameLoad: string filename used for loading data  
 self.no\_steps float number of iterations  
 self.natural\_angular\_frequency float the natural frequency of the oscillation  
 self.gamma float the damping constant  
 self.quality\_factor float quality factor  
 self.analytic\_series\_pos array the position of the analytical solution  
 self.analytic\_series\_vel array the velocity of the analytical solution  
 self.analytic\_energy array the array containing energy data of the analytical solution  
 self.\_\_coefficients array coefficients used for the analytical solution  
 self.b\_britical float the critical damping constant  
 self.Euler\_data array the data from Euler's method  
 self.B\_Euler\_data array the data from improved Euler's method  
 self.Verley array the data from Verlet's method  
 self.Euler\_Cromer\_data array the data from Euler-Cromer method  
 self.analytical\_data array variable for storing analytical data  
 self.time array the time array used in all simulations  
 self.disturbed\_Verlet\_data array the data of the Verlet method with force applied  
 '''* self.fileNameSave = fileNameSave  
 self.fileNameLoad = fileNameLoad  
 self.b = b  
 self.m = m  
 self.k = k  
 self.h = time\_step  
 self.init\_v = init\_v  
 self.init\_x = init\_x  
 self.no\_steps = int(np.rint(max\_time / time\_step))  
 self.natural\_angular\_frequency = np.sqrt(self.k / self.m)  
 **if** self.b != 0:  
 self.gamma = self.b / self.m  
 self.quality\_factor = self.natural\_angular\_frequency / self.gamma  
 self.analytic\_series\_pos = []  
 self.analytic\_series\_vel = []  
 self.analytic\_energy = []  
 self.\_\_coefficients = []  
 self.solver()  
 self.analytic\_solution()  
 self.data = []  
 self.b\_critical = 2 \* np.sqrt(self.k \* self.m)  
 *# data variables* self.Euler\_data = []  
 self.B\_Euler\_data = []  
 self.Verlet\_data = []  
 self.Euler\_Cromer\_data = []  
 self.analytical\_data = []  
 self.time = np.array(range(0, self.no\_steps, 1)) \* self.h  
 self.disturbed\_Verlet\_data = []  
  
 **def** runSimulation(self):  
 *'''  
 Runs the integrators as a single function  
 '''* self.Euler\_integrator()  
 self.Better\_Euler\_integrator()  
 self.Verlet\_integrator()  
 self.Euler\_Cromer\_integrator()  
 print(**"Simulation has been executed"**)  
  
 **def** getCoefficients(self):  
 *# simple get function* **return** self.\_\_coefficients  
  
 **def** Euler\_integrator(self):  
 *'''  
 parameter name type description  
 position\_series array stores the position temporarily  
 velocity\_series array stores the velocity temporarily  
 v\_n float stores the nth velocity term  
 x\_n float stores the nth position term  
 a\_n float stores the nth acceleration term  
 '''* position\_series = [self.init\_x]  
 velocity\_series = [self.init\_v]  
 **for** counter **in** range(1, self.no\_steps, 1):  
 v\_n = velocity\_series[len(velocity\_series) - 1]  
 x\_n = position\_series[len(position\_series) - 1]  
 a\_n = (-self.b / self.m) \* v\_n + (-self.k / self.m) \* x\_n  
  
 position\_series.append(x\_n + self.h \* v\_n)  
 velocity\_series.append(v\_n + self.h \* a\_n)  
  
 self.Euler\_data = [position\_series, velocity\_series, self.energy\_function(position\_series, velocity\_series)]  
  
 **def** Better\_Euler\_integrator(self):  
 *'''  
 parameter name type description  
 position\_series array stores the position temporarily  
 velocity\_series array stores the velocity temporarily  
 v\_n float stores the nth velocity term  
 x\_n float stores the nth position term  
 a\_n float stores the nth acceleration term  
 '''* position\_series = [self.init\_x]  
 velocity\_series = [self.init\_v]  
 **for** counter **in** range(1, self.no\_steps, 1):  
 v\_n = velocity\_series[len(velocity\_series) - 1]  
 x\_n = position\_series[len(position\_series) - 1]  
 a\_0 = (-self.b / self.m) \* v\_n + (-self.k / self.m) \* x\_n  
  
 position\_series.append(x\_n + self.h \* v\_n + 0.5 \* self.h \*\* 2 \* a\_0)  
 velocity\_series.append(v\_n + self.h \* a\_0)  
  
 self.B\_Euler\_data = [position\_series, velocity\_series, self.energy\_function(position\_series, velocity\_series)]  
  
 **def** Euler\_Cromer\_integrator(self):  
 *'''  
 parameter name type description  
 position\_series array stores the position temporarily  
 velocity\_series array stores the velocity temporarily  
 v\_n float stores the nth velocity term  
 x\_n float stores the nth position term  
 a\_n float stores the nth acceleration term  
 temp float temporary variable - stores the v\_n+1 term of the velocity  
 '''* position\_series = [self.init\_x]  
 velocity\_series = [self.init\_v]  
 **for** counter **in** range(1, self.no\_steps, 1):  
 v\_n = velocity\_series[len(velocity\_series) - 1]  
 x\_n = position\_series[len(position\_series) - 1]  
  
 a\_0 = (-self.b / self.m) \* v\_n + (-self.k / self.m) \* x\_n  
  
 temp = v\_n + self.h \* a\_0 *# v\_n+1* velocity\_series.append(temp)  
 position\_series.append(x\_n + self.h \* temp)  
  
 self.Euler\_Cromer\_data = [position\_series, velocity\_series,  
 self.energy\_function(position\_series, velocity\_series)]  
  
 **def** Verlet\_integrator(self):  
 *'''  
 parameter name type description  
 position\_series array stores the position temporarily  
 velocity\_series array stores the velocity temporarily  
 v\_n float stores the nth velocity term  
 x\_n float stores the nth position term  
 a\_n float stores the nth acceleration term  
 x\_1 float stores the second position of the oscillation  
 D float temporary variable for ease of calculation  
 B float temporary variable for ease of calculation  
 A float temporary variable for ease of calculation  
 '''* position\_series = [self.init\_x]  
 velocity\_series = [self.init\_v]  
  
 D = 2 \* self.m + self.b \* self.h  
 B = ((self.b \* self.h) - (2 \* self.m)) / D  
 A = 2 \* (2 \* self.m - (self.k \* self.h \*\* 2)) / D  
  
 a\_0 = (-self.b / self.m) \* self.init\_v + (-self.k / self.m) \* self.init\_x  
 x\_1 = self.init\_x + self.init\_v \* self.h + 0.5 \* a\_0 \* self.h \*\* 2 *# obtained using a Taylor expansion of order 2* position\_series.append(x\_1)  
  
 **for** counter **in** range(1, self.no\_steps, 1):  
 position\_series.append(A \* position\_series[counter] + B \* position\_series[counter - 1])  
  
 *# calculating velocities using an approximation of O(h^2)  
 # the velocity is estimated using the mean value theorem* **for** counter **in** range(1, self.no\_steps, 1):  
 velocity\_series.append(  
 (position\_series[counter + 1] - position\_series[counter - 1]) / (2 \* self.h)) *# +O(h^2)* position\_series = position\_series[:len(position\_series) - 1]  
  
 self.Verlet\_data = [position\_series, velocity\_series, self.energy\_function(position\_series, velocity\_series)]  
  
 **def** energy\_function(self, position, velocity):  
 *'''  
 parameter name type description  
 temp\_pos numpy array stores the position array  
 temp\_vel numpy array stores the velocity array* **:return***: the array containing energy values  
 '''* temp\_pos = np.array(position)  
 temp\_vel = np.array(velocity)  
 **return** 0.5 \* self.m \* temp\_vel \*\* 2 + 0.5 \* self.k \* temp\_pos \*\* 2  
  
 **def** convert\_array(self, array):  
 *# operates on 1 dimensional arrays  
 '''  
 parameter name type description  
 temp numpy array the array holding the array being converting* **:param** *array:* **:return***: converted array  
 '''* temp = []  
 **for** entry **in** array:  
 temp.append(entry)  
  
 **return** temp  
  
 **def** analytic\_solution(self):  
 *# creates the analytic solution position series  
 '''  
 parameter name type description  
 t\_0 float the time that the simulation is at  
 '''* t\_0 = 0  
 **for** counter **in** range(0, self.no\_steps, 1):  
 self.analytic\_series\_pos.append(self.ana\_position(t\_0))  
 self.analytic\_series\_vel.append(self.ana\_velocity(t\_0))  
 t\_0 += self.h  
 print(**"solution found"**)  
 self.analytic\_energy = self.energy\_function(self.analytic\_series\_pos, self.analytic\_series\_vel)  
  
 **def** solver(self):  
 *'''  
 parameter name type description  
 A float function coefficient  
 B float function coefficient  
 marker int the marker indicating the type of a solution  
 p float function coefficient  
 q float function coefficient  
 K float function coefficient  
 '''* A = 0  
 B = 0  
 temp = (self.b \*\* 2 / (4 \* self.m \*\* 2))  
 **if** self.b == 0:  
 marker = 1  
 print(**"The analytic solution is a simple harmonic motion"**)  
 omega = np.sqrt(self.k / self.m)  
 A = self.init\_v / omega  
 B = self.init\_x  
 self.\_\_coefficients = [omega, 0, marker, A, B]  
 **elif** (self.k / self.m) > temp: *# imaginary* print(**"The solution is a lightly damped oscillation"**)  
 marker = 3  
 p = -1 \* self.b / (2 \* self.m)  
 q = np.sqrt((self.k / self.m) - self.b \*\* 2 / (4 \* self.m \*\* 2))  
 *# initial conditions* A = (- self.init\_x \* p + self.init\_v) / q  
 B = self.init\_x  
 self.\_\_coefficients = [p, q, marker, A, B]  
 **elif** (self.k / self.m) == temp:  
 print(**"The solution is a critically damped oscillation"**)  
 marker = 2 *# repeated real solutions* K = -1 \* self.b / 2 \* self.m  
 *# initial conditions* A = self.init\_x  
 self.\_\_coefficients = [K, 0, marker, A, B]  
 **elif** (self.k / self.m) < temp: *# overdamped oscillation* marker = 4  
 print(**"The solution is an overdamped oscillation"**)  
 p = -1 \* self.b / 2 \* self.m + np.sqrt(-(self.k / self.m) + self.b \*\* 2 / (4 \* self.m \*\* 2))  
 q = -1 \* self.b / 2 \* self.m - np.sqrt(-(self.k / self.m) + self.b \*\* 2 / (4 \* self.m \*\* 2))  
  
 *# initial conditions* A = (q \* self.init\_x - self.init\_v) / (q - p)  
 B = self.init\_x - A  
 self.\_\_coefficients = [p, q, marker, A, B]  
  
 **def** ana\_position(self, t):  
 *'''  
 parameter name type description  
 k\_1 float function coefficient  
 k\_2 float function coefficient  
 marker int marker dictating the solution type  
 A float function coefficient  
 B float function coefficient* **:return** *returns the position of an analytic solution at time t  
 '''* k\_1 = self.\_\_coefficients[0]  
 k\_2 = self.\_\_coefficients[1]  
 marker = self.\_\_coefficients[2]  
 A = self.\_\_coefficients[3]  
 B = self.\_\_coefficients[4]  
 **if** marker == 1: *# no damping solution* **return** A \* np.sin(self.natural\_angular\_frequency \* t) + B \* np.cos(self.natural\_angular\_frequency \* t)  
 **elif** marker == 2: *# regular damping (complex)* **return** A \* np.exp(k\_1 \* t)  
 **elif** marker == 3: *# repeated root* **return** np.exp(k\_1 \* t) \* (A \* np.sin(k\_2 \* t) + B \* np.cos(k\_2 \* t))  
 **elif** marker == 4: *# two real distinct solutions* **return** A \* np.exp(k\_1 \* t) + B \* np.exp(k\_2 \* t)  
  
 **def** ana\_velocity(self, t):  
 *'''  
 parameter name type description  
 k\_1 float function coefficient  
 k\_2 float function coefficient  
 marker int marker dictating the solution type  
 A float function coefficient  
 B float function coefficient* **:return** *returns the velocity of an analytic solution at time t  
 '''* k\_1 = self.\_\_coefficients[0]  
 k\_2 = self.\_\_coefficients[1]  
 marker = self.\_\_coefficients[2]  
 A = self.\_\_coefficients[3]  
 B = self.\_\_coefficients[4]  
 **if** marker == 1: *# no damping solution* **return** A \* k\_1 \* np.cos(k\_1 \* t) - B \* k\_1 \* np.sin(k\_1 \* t)  
 **elif** marker == 2: *# regular damping (complex)* **return** A \* k\_1 \* np.exp(k\_1 \* t)  
 **elif** marker == 3: *# repeated root* **return** k\_1 \* np.exp(k\_1 \* t) \* (A \* np.sin(k\_2 \* t) + B \* np.cos(k\_2 \* t)) + np.exp(k\_1 \* t) \* (  
 A \* k\_2 \* np.cos(k\_2 \* t) - B \* k\_2 \* np.sin(k\_2 \* t))  
 **elif** marker == 4: *# two real distinct solutions* **return** A \* k\_1 \* np.exp(k\_1 \* t) + B \* k\_2 \* np.exp(k\_2 \* t)  
  
 **def** plot\_data(self): *# plots all on separate graphs  
 # analytical solution  
 '''  
 parameter name type description  
 axes\_1 object subplot object  
 axes\_2 object subplot object  
 axes\_3 object subplot object  
 axes\_4 object subplot object  
 axes\_5 object subplot object  
 axes\_6 object subplot object  
 axes\_7 object subplot object  
 axes\_8 object subplot object  
 axes\_9 object subplot object  
 axes\_10 object subplot object  
 figure object figure object  
 figure2 object figure object  
 figure3 object figure object  
 figure4 object figure object  
 figure5 object figure object  
 '''* figure3 = plt.figure()  
 axes\_5 = figure3.add\_subplot(121)  
 axes\_5.plot(self.analytic\_series\_pos, self.analytic\_series\_vel, label=**"Analytical"**)  
 axes\_5.set\_xlabel(**"position/ m"**) *# edit later if the functions don't exist* axes\_5.set\_ylabel(**"velocity/ ms^-1"**) *# as above  
 # energy plotting* axes\_6 = figure3.add\_subplot(122)  
 axes\_6.plot(self.time, self.analytic\_energy)  
 axes\_6.set\_xlabel(**"time/ s"**)  
 axes\_6.set\_ylabel(**"energy/ J"**)  
 figure3.legend()  
  
 *# Euler method  
 # plotting* figure = plt.figure()  
 axes\_1 = figure.add\_subplot(121)  
 axes\_1.plot(self.Euler\_data[0], self.Euler\_data[1], label=**"Euler"**)  
 axes\_1.set\_xlabel(**"position/ m"**) *# edit later if the functions don't exist* axes\_1.set\_ylabel(**"velocity/ ms^-1"**) *# as above  
 # energy plotting* axes\_2 = figure.add\_subplot(122)  
 axes\_2.plot(self.time, self.Euler\_data[2])  
 axes\_2.set\_xlabel(**"time/ s"**)  
 axes\_2.set\_ylabel(**"energy/ J"**)  
 figure.legend()  
 *# end plotting  
  
 # Better Euler method  
 # plotting* figure2 = plt.figure()  
 axes\_3 = figure2.add\_subplot(121)  
 axes\_3.plot(self.B\_Euler\_data[0], self.B\_Euler\_data[1], label=**"Better Euler"**)  
 axes\_3.set\_xlabel(**"position/ m"**) *# edit later if the functions don't exist* axes\_3.set\_ylabel(**"velocity/ ms^-1"**) *# as above  
 # energy plotting* axes\_4 = figure2.add\_subplot(122)  
 axes\_4.plot(self.time, self.B\_Euler\_data[2])  
 axes\_4.set\_xlabel(**"time/ s"**)  
 axes\_4.set\_ylabel(**"energy/ J"**)  
 figure2.legend()  
 *# end plotting  
  
 # Verlet method* figure4 = plt.figure()  
 axes\_7 = figure4.add\_subplot(121)  
 axes\_7.plot(self.Verlet\_data[0], self.Verlet\_data[1], label=**"Verlet"**)  
 axes\_7.set\_xlabel(**"position/ m"**)  
 axes\_7.set\_ylabel(**"velocity/ ms^-1"**)  
 *# energy plotting* axes\_8 = figure4.add\_subplot(122)  
 axes\_8.plot(self.time, self.Verlet\_data[2])  
 axes\_8.set\_xlabel(**"time/ s"**)  
 axes\_8.set\_ylabel(**"energy/ J"**)  
 figure4.legend()  
  
 *# Euler Cromer method* figure5 = plt.figure()  
 axes\_9 = figure5.add\_subplot(121)  
 axes\_9.plot(self.Euler\_Cromer\_data[0], self.Euler\_Cromer\_data[1], label=**"Euler Cromer Method"**)  
 axes\_9.set\_xlabel(**"position/ m"**)  
 axes\_9.set\_ylabel(**"velocity/ ms^-1"**)  
 *# energy plotting* axes\_10 = figure5.add\_subplot(122)  
 axes\_10.plot(self.time, self.Euler\_Cromer\_data[2])  
 axes\_10.set\_xlabel(**"time/ s"**)  
 axes\_10.set\_ylabel(**"energy/ J"**)  
 figure5.legend()  
  
 **def** plot\_single(self):  
 *# plots the single  
 '''  
 parameter name type description  
 axes\_1 object subplot object  
 axes\_2 object subplot object  
 figure object figure object  
 '''* figure = plt.figure()  
 axes\_1 = figure.add\_subplot(121)  
 axes\_1.set\_xlabel(**"position/ m"**)  
 axes\_1.set\_ylabel(**"velocity/ ms^-1"**)  
 *# energy\_function* energy = []  
 axes\_2 = figure.add\_subplot(122)  
 axes\_2.set\_xlabel(**"time/ s"**)  
 axes\_2.set\_ylabel(**"Energy/ J"**)  
  
 axes\_1.plot(self.Euler\_data[0], self.Euler\_data[1], label=**"Euler"**)  
 axes\_2.plot(self.time, self.Euler\_data[2])  
  
 axes\_1.plot(self.B\_Euler\_data[0], self.B\_Euler\_data[1], label=**"Improved Euler"**)  
 axes\_2.plot(self.time, self.B\_Euler\_data[2])  
  
 axes\_1.plot(self.Verlet\_data[0], self.Verlet\_data[1], label=**"Verlet"**)  
 axes\_2.plot(self.time, self.Verlet\_data[2])  
  
 axes\_1.plot(self.Euler\_Cromer\_data[0], self.Euler\_Cromer\_data[1], label=**"Euler Cromer"**)  
 axes\_2.plot(self.time, self.Euler\_Cromer\_data[2])  
  
 axes\_1.plot(self.analytic\_series\_pos, self.analytic\_series\_vel, label=**"Analytic solution"**)  
 axes\_2.plot(self.time, self.analytic\_energy)  
 figure.legend()  
  
 **def** save\_data(self):  
 *'''  
 parameter name type description  
 temp array stores the analytic data  
 data dictionary stores the data to be saved  
 '''  
 # all files are saved as json dictionaries in the format "name of method": [position, velocity, energy]  
 # the header of the file headers named appropriately contains the* temp = [self.analytic\_series\_pos, self.analytic\_series\_vel, self.convert\_array(self.analytic\_energy)]  
  
 data = {}  
 data[**"Analytic"**] = temp  
 data[**"Euler"**] = [self.Euler\_data[0], self.Euler\_data[1], self.convert\_array(self.Euler\_data[2])]  
 data[**"Better Euler"**] = [self.B\_Euler\_data[0], self.B\_Euler\_data[1], self.convert\_array(self.B\_Euler\_data[2])]  
 data[**"Verlet"**] = [self.Verlet\_data[0], self.Verlet\_data[1], self.convert\_array(self.Verlet\_data[2])]  
 data[**"Euler Cromer"**] = [self.Euler\_Cromer\_data[0], self.Euler\_Cromer\_data[1],  
 self.convert\_array(self.Euler\_Cromer\_data[2])]  
 data[**"coefficients"**] = [self.h, self.no\_steps, self.b, self.m, self.k, self.init\_x,  
 self.init\_v] *# h, T, b, m, k, x, v* **with** open(**"data.txt"**, **'w'**) **as** outfile:  
 json.dump(data, outfile)  
 outfile.close()  
  
 **def** load\_data(self):  
 *'''  
 parameter name type description  
 data dictionary stores the data to be saved  
 json\_file object json file object  
 '''* **try**:  
 **with** open(self.fileNameLoad) **as** json\_file:  
 data = json.load(json\_file)  
 self.Euler\_data = data[**"Euler"**]  
 self.B\_Euler\_data = data[**"Better Euler"**]  
 self.Verlet\_data = data[**"Verlet"**]  
 self.analytic\_series\_pos = data[**"Analytic"**][0]  
 self.analytic\_series\_vel = data[**"Analytic"**][1]  
 self.analytic\_energy = data[**"Analytic"**][2]  
 self.Euler\_Cromer\_data = data[**"Euler Cromer"**]  
 self.h, self.no\_steps, self.b, self.m, self.k, self.init\_x, self.init\_v = data[**"coefficients"**]  
 self.time = np.array(range(0, self.no\_steps, 1)) \* self.h  
  
 json\_file.close()  
 **return True  
 except**:  
 print(**"The file was not found"**)  
 **return False  
  
 def** find\_accuracy(self):  
 *# finds the accuracy of the simulation by using the analytic energy as a baseline  
 # this assigns a number of "fictitious energy" and also graphs the growth of the errors with time  
 '''  
 parameter name type description  
 axes\_1 object subplot object  
 fict\_energy numpy array stores the error energy  
 baseline numpy array stores the analytic energy  
 temp numpy array stores the plotting value of the error energy  
 figure object figure object  
 '''* figure = plt.figure()  
 axes\_1 = figure.add\_subplot(111)  
 axes\_1.set\_ylabel(**"Energy error/ J"**)  
 axes\_1.set\_xlabel(**"time/ s"**)  
 fict\_energy = []  
 baseline = np.array(self.analytic\_energy)  
  
 *# Euler's method* temp = np.abs(np.array(self.Euler\_data[2]) - baseline)  
 axes\_1.plot(self.time, temp, label=**"Euler"**)  
 fict\_energy.append(np.sum(temp))  
 *# Better Euler* temp = np.abs(np.array(self.B\_Euler\_data[2]) - baseline)  
 axes\_1.plot(self.time, temp, label=**"Improved Euler"**)  
 fict\_energy.append(np.sum(temp))  
 *# Cromer* temp = np.abs(np.array(self.Euler\_Cromer\_data[2]) - baseline)  
 axes\_1.plot(self.time, temp, label=**"Euler Cromer"**)  
 fict\_energy.append(np.sum(temp))  
 *# Verlet* temp = np.abs(np.array(self.Verlet\_data[2]) - baseline)  
 axes\_1.plot(self.time, temp, label=**"Verlet"**)  
 fict\_energy.append(np.sum(temp))  
 temp = 0  
 print(**"The energy errors for b = "** + str(self.b))  
 print(**"Euler: "** + str(fict\_energy[0]) + **" J"**)  
 print(**"Improved Euler: "** + str(fict\_energy[1]) + **"J"**)  
 print(**"Euler Cromer: "** + str(fict\_energy[2]) + **"J"**)  
 print(**"Verlet: "** + str(fict\_energy[3]) + **"J"**)  
 figure.legend()  
 axes\_1.set\_title(**"h = "** + str(self.h))  
  
 **def** const\_dist\_Verlet\_integrator(self, force, min, max):  
 *'''  
 parameter name type description  
 position\_series array stores the position temporarily  
 velocity\_series array stores the velocity temporarily  
 v\_n float stores the nth velocity term  
 x\_n float stores the nth position term  
 a\_n float stores the nth acceleration term  
 x\_1 float stores the second position of the oscillation  
 D float temporary variable for ease of calculation  
 B float temporary variable for ease of calculation  
 A float temporary variable for ease of calculation  
 '''* position\_series = [self.init\_x]  
 velocity\_series = [self.init\_v]  
  
 D = 2 \* self.m + self.b \* self.h  
 B = (self.b \* self.h - 2 \* self.m) / D  
 A = 2 \* (2 \* self.m - self.k \* self.h \*\* 2) / D  
  
 a\_0 = (-self.b / self.m) \* self.init\_v + (-self.k / self.m) \* self.init\_x  
 x\_1 = self.init\_x + self.init\_v \* self.h + 0.5 \* a\_0 \* self.h \*\* 2 *# obtained using a Taylor expansion of order 2* position\_series.append(x\_1)  
  
 **for** counter **in** range(1, self.no\_steps, 1):  
 **if** (counter \* self.h > min) **and** (counter \* self.h < max):  
 position\_series.append(  
 A \* position\_series[counter] + B \* position\_series[counter - 1] + (force / self.m \* self.h \*\* 2))  
 **else**:  
 position\_series.append(A \* position\_series[counter] + B \* position\_series[counter - 1])  
  
 *# calculating velocities using an approximation of O(h^2)  
 # the velocity is estimated using the mean value theorem  
 # the velocity is independent of the equation of motion. It just utilises the definition of velocity. If h is small  
 # enough this approximation holds true* **for** counter **in** range(1, self.no\_steps, 1):  
 velocity\_series.append(  
 (position\_series[counter + 1] - position\_series[counter - 1]) / (2 \* self.h)) *# +O(h^2)* position\_series = position\_series[:len(position\_series) - 1]  
  
 self.disturbed\_Verlet\_data = [position\_series, velocity\_series,  
 self.energy\_function(position\_series, velocity\_series)]  
  
 **def** funct\_dist\_Verlet\_integrator(self, min, max, Amp, freq):  
 *'''  
 parameter name type description  
 position\_series array stores the position temporarily  
 velocity\_series array stores the velocity temporarily  
 v\_n float stores the nth velocity term  
 x\_n float stores the nth position term  
 a\_n float stores the nth acceleration term  
 x\_1 float stores the second position of the oscillation  
 D float temporary variable for ease of calculation  
 B float temporary variable for ease of calculation  
 A float temporary variable for ease of calculation  
 '''* position\_series = [self.init\_x]  
 velocity\_series = [self.init\_v]  
  
 D = 2 \* self.m + self.b \* self.h  
 B = (self.b \* self.h - 2 \* self.m) / D  
 A = 2 \* (2 \* self.m - self.k \* self.h \*\* 2) / D  
  
 a\_0 = (-self.b / self.m) \* self.init\_v + (-self.k / self.m) \* self.init\_x  
 x\_1 = self.init\_x + self.init\_v \* self.h + 0.5 \* a\_0 \* self.h \*\* 2 *# obtained using a Taylor expansion of order 2* position\_series.append(x\_1)  
  
 **for** counter **in** range(1, self.no\_steps, 1):  
 **if** (counter \* self.h > min) **and** (counter \* self.h < max):  
 position\_series.append(  
 A \* position\_series[counter] + B \* position\_series[counter - 1] + (  
 Amp \* np.sin(freq \* counter \* self.h) / self.m \* self.h \*\* 2))  
 **else**:  
 position\_series.append(A \* position\_series[counter] + B \* position\_series[counter - 1])  
  
 *# calculating velocities using an approximation of O(h^2)  
 # the velocity is estimated using the mean value theorem  
 # the velocity is independent of the equation of motion. It just utilises the definition of velocity. If h is small  
 # enough this approximation holds true* **for** counter **in** range(1, self.no\_steps, 1):  
 velocity\_series.append(  
 (position\_series[counter + 1] - position\_series[counter - 1]) / (2 \* self.h)) *# +O(h^2)* position\_series = position\_series[:len(position\_series) - 1]  
  
 self.disturbed\_Verlet\_data = [position\_series, velocity\_series,  
 self.energy\_function(position\_series, velocity\_series)]  
  
 **def** push\_testing(self, min, max, force, amp, freq):  
 *'''  
 parameter name type description  
 constant\_data array array of the data modified by a constant force  
 function\_data array array of the data modified by a sinusoidal force  
 axes\_1 object subplot object  
 figure object figure object  
 '''  
 # constant force* self.const\_dist\_Verlet\_integrator(force, min, max)  
 constant\_data = self.disturbed\_Verlet\_data  
  
 *# sinusoidal force* self.funct\_dist\_Verlet\_integrator(min, max, amp, freq)  
 function\_data = self.disturbed\_Verlet\_data  
  
 figure = plt.figure()  
 axes\_1 = figure.add\_subplot(111)  
 axes\_1.set\_xlabel(**"Time (s)"**)  
 axes\_1.set\_ylabel(**"Position (m)"**)  
 axes\_1.plot(self.time, constant\_data[0], label=**"constant force"**)  
 axes\_1.plot(self.time, function\_data[0], label=**"sinusoidal force"**)  
 axes\_1.plot(self.time, self.Verlet\_data[0], label=**"undisturbed"**)  
 figure.legend()  
  
 **def** search(self, arr, x):  
 *'''  
 variable name type description  
 i integer counter  
 arr list array  
 x float the value being searched  
 '''  
 # linear search function* **for** i **in** range(len(arr)):  
 **if** arr[i] == x:  
 **return** i  
 **return** -1  
  
 **def** resonance\_Plot(self):  
 *'''  
 parameter name type description  
 temp float/array temporary variable  
 freq\_array array array of angular frequencies  
 amplitude array sum of the amplotudes for a resonance plot  
 figure object figure object  
 axes\_1 object subplot object  
 '''* temp = np.sqrt(self.k / self.m)  
 freq\_array = []  
 **for** counter **in** range(0, 200, 1):  
 freq\_array.append(counter \* temp \* 0.01)  
  
 amplitude = []  
 **for** freq **in** freq\_array:  
 *# get the data sets for a specific frequency* self.funct\_dist\_Verlet\_integrator(0, self.no\_steps \* self.h, self.init\_x \* 0.5, freq)  
 temp = self.disturbed\_Verlet\_data[0]  
 *# calculate amplitude* temp = np.abs(np.array(temp))  
 amplitude.append(np.mean(temp))  
 *# append to the arrays  
 # plot the results* figure = plt.figure()  
 axes\_1 = figure.add\_subplot(111)  
 axes\_1.set\_xlabel(**"Frequency/ Hz"**)  
 axes\_1.set\_ylabel(**"Amplitude/ m"**)  
 axes\_1.plot(freq\_array, amplitude)  
  
 **def** Critical(self):  
 *'''  
 parameter name type description  
 temp float stores the b value to avoid change  
 b float damping coefficient  
 data array stores the data of all the integrations  
 figure4 object figure object  
 figure object figure object  
 axes\_1 object subplot object  
 axes\_7 object subplot object  
 axes\_8 object subplot object  
 '''* temp = self.b  
 b = [0.5 \* self.b\_critical, self.b\_critical, 2 \* self.b\_critical]  
 data = []  
 **for** entry **in** b:  
 self.b = entry  
 self.Verlet\_integrator()  
 data.append(self.Verlet\_data)  
  
 *# Verlet method  
 # phase plots* figure4 = plt.figure()  
 axes\_7 = figure4.add\_subplot(121)  
 axes\_7.plot(data[0][0], data[0][1], label=**"0.5b"**)  
 axes\_7.plot(data[1][0], data[1][1], label=**"b"**)  
 axes\_7.plot(data[2][0], data[2][1], label=**"2b"**)  
 axes\_7.set\_xlabel(**"position (m)"**)  
 axes\_7.set\_ylabel(**"velocity (ms^-1)"**)  
 *# energy plotting* axes\_8 = figure4.add\_subplot(122)  
 axes\_8.plot(self.time, data[0][2])  
 axes\_8.plot(self.time, data[1][2])  
 axes\_8.plot(self.time, data[2][2])  
 axes\_8.set\_xlabel(**"time (s)"**)  
 axes\_8.set\_ylabel(**"energy (J)"**)  
 *# position plot* figure = plt.figure()  
 axes\_1 = figure.add\_subplot(111)  
 axes\_1.plot(self.time, data[0][0], label=**"0.5b"**)  
 axes\_1.plot(self.time, data[1][0], label=**"b"**)  
 axes\_1.plot(self.time, data[2][0], label=**"2b"**)  
 axes\_1.set\_ylabel(**"Position (m)"**)  
 axes\_1.set\_xlabel(**"Time (s)"**)  
 figure4.legend()  
 figure.legend()  
  
 self.b = temp  
  
 **def** complete\_Resonance(self):  
 *'''  
 parameter name type description  
 temp 2 float stores the b value to avoid change  
 figure object figure object  
 axes\_1 object subplot object  
 freq\_array array array of angular frequencies  
 amplitude array sum of the amplotudes for a resonance plot  
 b\_prime array stores the damping coefficients  
 '''* figure = plt.figure()  
 axes\_1 = figure.add\_subplot(111)  
 axes\_1.set\_xlabel(**"Frequency/ Hz"**)  
 axes\_1.set\_ylabel(**"Amplitude/ m"**)  
  
 temp2 = self.b  
 temp = np.sqrt(self.k / self.m)  
 freq\_array = []  
 **for** counter **in** range(0, 220, 1):  
 freq\_array.append(counter \* temp \* 0.01)  
 b\_prime = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 2.2, 4.5, 9]  
 amplitude = []  
 **for** b **in** b\_prime:  
 self.b = b  
 **for** freq **in** freq\_array:  
 *# get the data sets for a specific frequency* self.funct\_dist\_Verlet\_integrator(0, self.no\_steps \* self.h, self.init\_x \* 0.5, freq)  
 temp = self.disturbed\_Verlet\_data[0]  
 *# calculate amplitude* temp = np.abs(np.array(temp))  
 amplitude.append(np.mean(temp))  
 *# append to the arrays* axes\_1.plot(freq\_array, amplitude, label=**"b = "** + str(b) + **" kgs^-1"**)  
 amplitude = []  
 *# plot the results* figure.legend()  
 self.b = temp2  
  
 **def** find\_accuracy\_complete(self):  
 *'''  
 parameter name type description  
 temp float stores the b variable to avoid change  
 '''* temp = self.b  
 self.b = 0  
 self.find\_accuracy()  
 self.b = 0.1  
 self.find\_accuracy()  
 self.b = 1  
 self.find\_accuracy()  
 self.b = 2  
 self.find\_accuracy()  
 self.b = 3  
 self.find\_accuracy()  
 self.b = temp  
  
  
**def** getData(): *# edit this function  
 '''  
 parameter name type description  
 m float mass  
 k float spring constant  
 b float damping coefficient  
 T float total time  
 h float time step  
 init\_x float initial position  
 init\_v float initial velocity  
 '''* m = float(input(**"Enter the value for the mass of the particle: "**))  
 k = float(input(**"Enter the value of the spring constant: "**))  
 b = float(input(**"Enter the value of the damping constant "**))  
 T = float(input(**"Enter the time you want the simulation to run: "**))  
 h = float(input(**"Enter the time step in seconds: "**))  
 init\_x = float(input(**"Enter the initial position"**))  
 init\_v = float(input(**"Enter the initial velocity"**))  
 **return** m, k, b, T, h, init\_x, init\_v  
  
  
**def** main():  
 *'''  
 parameter name type description  
 option int option chosen  
 name string file name  
 os object the class object  
 check boolean check variable  
 '''* option = 0  
 **while** option != **"9"**:  
  
 print(**"1. Run simulation"**)  
 print(**"2. Load old simulation"**)  
 option = input(**"Select an option:"**)  
 **if** option == **"1" or** option == **"2"**:  
 **if** option == **"1"**:  
 m, k, b, T, h, init\_x, init\_v = getData()  
 os = SHO(h, T, b, m, k, init\_x, init\_v)  
 **elif** option == **"2"**:  
 print(**"Enter the name of the file or type 'none' if you want to use default"**)  
 name = input()  
 **if** name == **"none"**:  
 os = SHO(0.01, 100)  
 os.load\_data()  
 **else**:  
 os = SHO(0.01, 100, fileNameLoad=name)  
 check = os.load\_data()  
 **if not** check:  
 print(**"Goodbye!"**)  
 **return** 0  
  
 print(**"3. Run critical damping simulation"**)  
 print(**"4. Run simulation with the force appplied"**)  
 print(**"5. Plot all of it"**)  
 print(**"6. Save the simulation"**)  
 print(**"7. Plot Resonance Curves"**)  
 print(**"8. Run default"**)  
 print(**"9. Leave"**)  
 option = input(**"Select an option:"**)  
 **if** option == **"3"**:  
 os.Critical()  
 **elif** option == **"4"**:  
 min = float(input(**"Minimum time: "**))  
 max = float(input(**"Maximum time: "**))  
 force = float(input(**"Force magnitude: "**))  
 Amp = float(input(**"Sinusoidal force amplitude: "**))  
 freq = float(input(**"Sinusoidal force frequency: "**))  
 os.push\_testing(min, max, force, Amp, freq)  
 **elif** option == **"5"**:  
 os.plot\_data()  
 os.plot\_single()  
 **elif** option == **"6"**:  
 os.save\_data()  
 print(**"File was saved as data.txt"**)  
 **elif** option == **"7"**:  
 os.complete\_Resonance()  
 **elif** option == **"8"**:  
 print(**"Running default simulation"**)  
 os.runSimulation()  
 os.push\_testing(45.6, 100, 2, 2, 0.062832) *# at zero amplitude* os.push\_testing(57, 100, 2, 2, 6.2832) *# at 3/4 of a cycle* os.push\_testing(53.2, 100, 2, 2, 0.41)  
 os.find\_accuracy\_complete()  
 os.Critical()  
 os.complete\_Resonance()  
 os.plot\_single()  
 os.plot\_data()  
 plt.show()  
  
 **else**:  
 print(**"Goodbye!"**)  
  
  
main()  
plt.show()