PyLab - SpringMass

PHY224 Lab 3

Fredrik Dahl Bråten, Pankaj Patil

October 18, 2021

1 Abstract

In this exercise, we studied the equation of motion of a mass-spring system in damped and undamped cases. In both the cases the we verified the experimental data with that obtained using qualitative fit of equation of motion. This analysis was done in Python by use of the numpy, scipy and matplotlib modules.

2 Introduction

The undamped spring mass equation is given by the Hook's law,

$$F_{spring} = -ky$$

Where y is the vertical displacement of the mass and k is the spring constant. Above equation is used to derive the equation of motion of the spring mass system,

$$\frac{d^2y}{dt^2} + \Omega_0^2 y = 0 \implies y = y_0 + A\sin(\Omega_0 t) \tag{1}$$

Where $\Omega_0 = \sqrt{\frac{k}{m}}$, $A =$ Amplitude of Oscillations, $y_0 =$ Initial Position of the mass

In case of damping, we need to add damping force to the equation which depends on the velocity of the mass relative to surrounding medium. In our case, where we have small Reynolds numbers, the drag force is directly proportional to the velocity of the mass,

$$\vec{F_d} = -\gamma\vec{v}$$

Where γ is damping coefficient. The equation of motion is then given by

$$\frac{d^2y}{dt^2} + \gamma\frac{dy}{dt} + \Omega_0^2 y = 0 \implies y = y_0 + Ae^{-\gamma t}\sin(\Omega_0 t) \tag{2}$$

In both the cases, we qualitatively fit the data to the above displacement equations.

3 Methods, Materials and Experimental Procedure

We successfully followed the procedures as described by the TAs and lab manual [1] for this experiment.

4 Results

4.1 Undamped Spring Mass

In Appendix Figure 1, we see the displacement data is plotted against time for the undamped case. The graph is a qualitative fit of the equation describing the undamped oscillatory motion of spring-mass system.

Using the qualitative fit we obtained following values,

y_0 = Initial Position = 20.71 cm

A = Amplitude = 0.70 cm

Ω_0 = Frequency of Oscillations = 9.07 rad/s

$T = \frac{2\pi}{\Omega_0}$ = Period of Oscillations = 0.693 s

Using the Ω_0 value we obtain the spring constant

$$k = m\Omega_0^2 = 16.44\ kg/s^2$$

The simulated data for the undamped case is plotted in Figure 5, 6.

4.2 Damped Spring Mass

In Appendix Figure 7, we see the displacement data is plotted against time for the damped case. The graph is a qualitative fit of the equation describing the damped oscillatory motion of spring-mass system.

Using the qualitative fit we obtained following values,

y_0 = Initial Position = 19.94 cm

A = Amplitude = 1.64 cm

γ = Damping Coefficient = 0.01

Ω_0 = Frequency of Oscillations = 8.69 rad/s

$T = \frac{2\pi}{\Omega_0}$ = Period of Oscillations = 0.723 s

Using the Ω_0 value we obtain the spring constant

$$k = m\Omega_0^2 = 16.24\ kg/s^2$$

The simulated data for the damped case is plotted in Figure 8, 9, 10.

5 Discussion

We first derive the equation of motion for undamped spring-mass system as follows,

$F_{spring} = -ky$ Hook's Law

$F = ma$ Newton's Second Law of Motion

$\implies ma = -ky$

$\implies m\frac{d^2y}{dt^2} = -ky$

$\implies m\frac{d^2y}{dt^2} + ky = 0$

Here we approximated that the motion of the system is purely one dimensional in only y-direction.

Above equation of motion can be written as,

$$\begin{aligned}
&\frac{d^2y}{dt^2} = -\Omega_0^2 y \quad \text{where } \Omega_0 = \sqrt{\frac{k}{m}} \\
\implies &\frac{dv}{dy} = -\Omega_0^2 y \quad \text{where } \frac{d^2y}{dt^2} = \frac{dv}{dy}, \quad v = \frac{dy}{dt} \\
\implies &\frac{1}{\Delta t}[v(t+\Delta t) - v(t)] = -\Omega_0^2 y \\
\implies &[v(t+\Delta t) - v(t)] = -\Delta t \Omega_0^2 y \\
\implies &v(t+\Delta t) = v(t) - \Delta t \Omega_0^2 y
\end{aligned}$$

And

$$\begin{aligned}
&v = \frac{dy}{dt} \\
\implies &\frac{1}{\Delta t}[y(t+\Delta t) - y(t)] = v(t) \\
\implies &y(t+\Delta t) = y(t) + \Delta t v(t)
\end{aligned}$$

Using above equations for derivative approximations, for the Forward Euler method, we get following equations,

$$y_{i+1} = y_i + \Delta t v_i$$
$$v_{i+1} = v_i - \Delta t \Omega_0^2 y_i$$

for $i = 0, 1, 2, \ldots$

Above equations were used to compute the simulated oscillations which are plotted in Figure 5, 6.

The oscillatory motion of spring mass system is a sinusoidal graph, which is expected as it is a periodic motion.

Our specified parameters produce a curve which fit our data very well.

In the Distance and Velocity vs. Time plots, we see that the Euler-Cromer simulation fit our data over time much better than the Forward Euler simulation, which amplitude grows significantly in time. The amplitude of the plots should however be constant, if not be weakly decreasing, due to the small unavoidable damping of our system of experiment, as seen in the plot of measured data.

As expected, we get elliptical phase plots for our measured and simulated Distance and Velocity, however, as we will see in the energy plot, the Forward Euler simulation grows in energy, which causes the phase space plot of the Forward Euler simulation ellipse to increase in radius over time. This also corresponds to the increase in amplitude in the position and velocity plot over time.

Furthermore, as we can see from the total Energy plots, the total energy of the system is approximately conserved for the Euler-Cromer simulation, grows in time and is not conserved in the Forward Euler simulation (which is unphysical for our system), and is approximately conserved for our measured data when accounting for uncertainties in our measurements of the distance over time. In theory, total energy should be conserved, but the energy will oscillate to be in the form of potential and kinetic energy.

The radius of the elliptical phase plots, correspond to the total energy of the system. When considering our total energy plots over time, it makes sense that the Euler Cromer and measured data plots give approximately stable phase ellipses, though the Forward Euler phase plot has increasing radius with time, as its energy is increasing over time. The reason the radius should be constant, is that the total energy should be constant. The y and v component of the phase plot radius oscillate in their contribution to the radius length, corresponding to how the total energy is conserved, but oscillates in the form of kinetic and potential energy.

For the undamped spring-mass system, the mechanical energy is conserved and is given by,

$$E_{tot} = \frac{1}{2}mv^2 + \frac{1}{2}ky^2$$

Rearranging above gives,

$$\frac{v^2}{k} + \frac{y^2}{m} = \frac{2E_{tot}}{mk}$$

Which is an equation of an ellipse. Hence the phase plot of system is an ellipse.

For damped oscillations, the equation of motion is given by Eqn. (2) in the Introduction. Using this equation, we get Euler-Cromer approximations for distance and velocity values. These approximations were used to plot simulated data for damped oscillations in Figure 8, 9, 10.

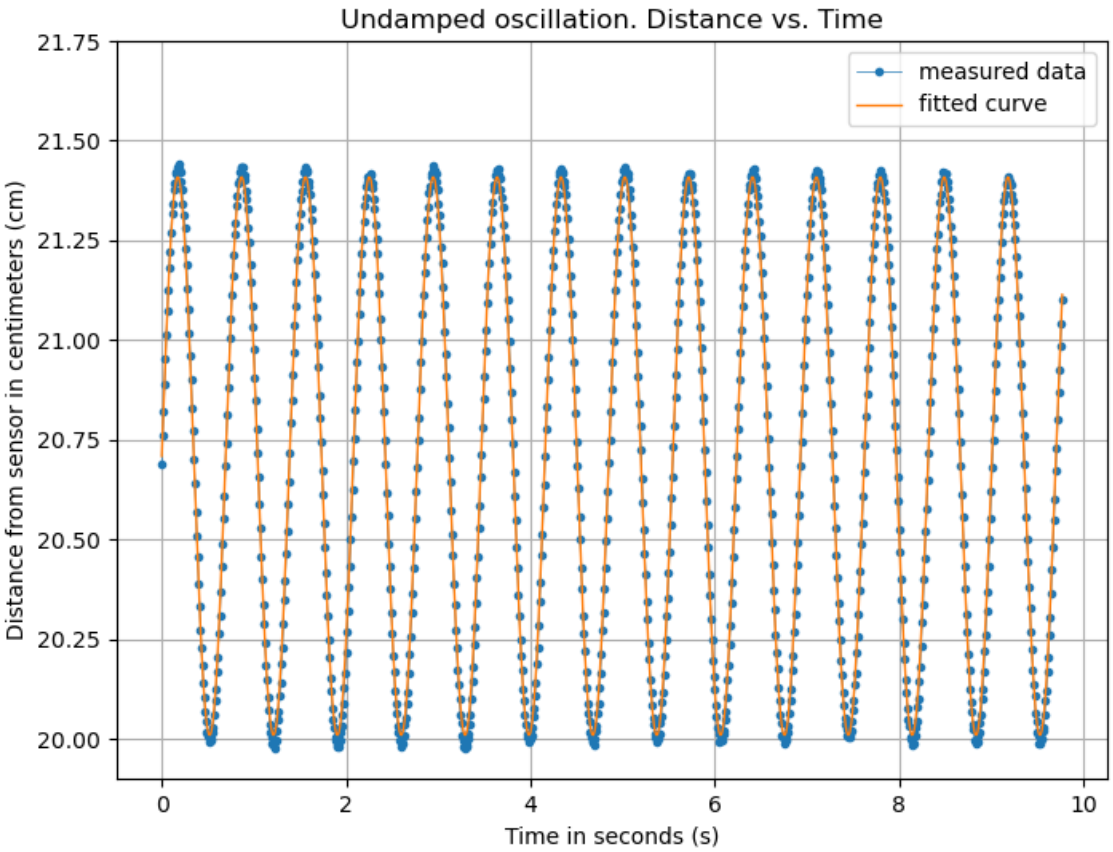
In damped oscillations, as expected the amplitude of oscillation decays exponentially as is evident from Figure 7. We also notice that the energy of system also decreases with time as expected, and is shown in Figure 10.

6 Conclusions

By approximating the motion in one dimension, we established that the equation of motion of spring-mass system in undamped case is given by Eqn (1). And that for damped system is give by Eqn. (2). In undamped system the energy of the system is conserved. Qualitative fit of the solution to equation of motion enables us to compute the spring constant, in both the cases and it is found to be in agreement within experimental errors. The qualitative fit establishes that the motion of spring-mass system is sinusoidal with constant period. In case of damping the amplitude of the motion decays exponentially. Furthermore, our two simulations using the Forward Euler and the Euler-Cromer methods, show that the Forward Euler method does not conserve energy, as the Euler-Cromer method approximately does. Therefore, the Euler-Cromer method is a better method for solving differential equations such as those we have worked with here.

A Appendix

A.1 Plots For Undamped Spring-Mass System



Model Function : $y = y_0 + A\sin(\Omega_0 t)$
y_0 = Initial Position = 20.71 cm
A = Amplitude = 0.70 cm
Ω_0 = Frequency of Oscillations = $2\pi/0.693$ rad/s

Figure 1: Undamped Oscillations: Distance vs. Time

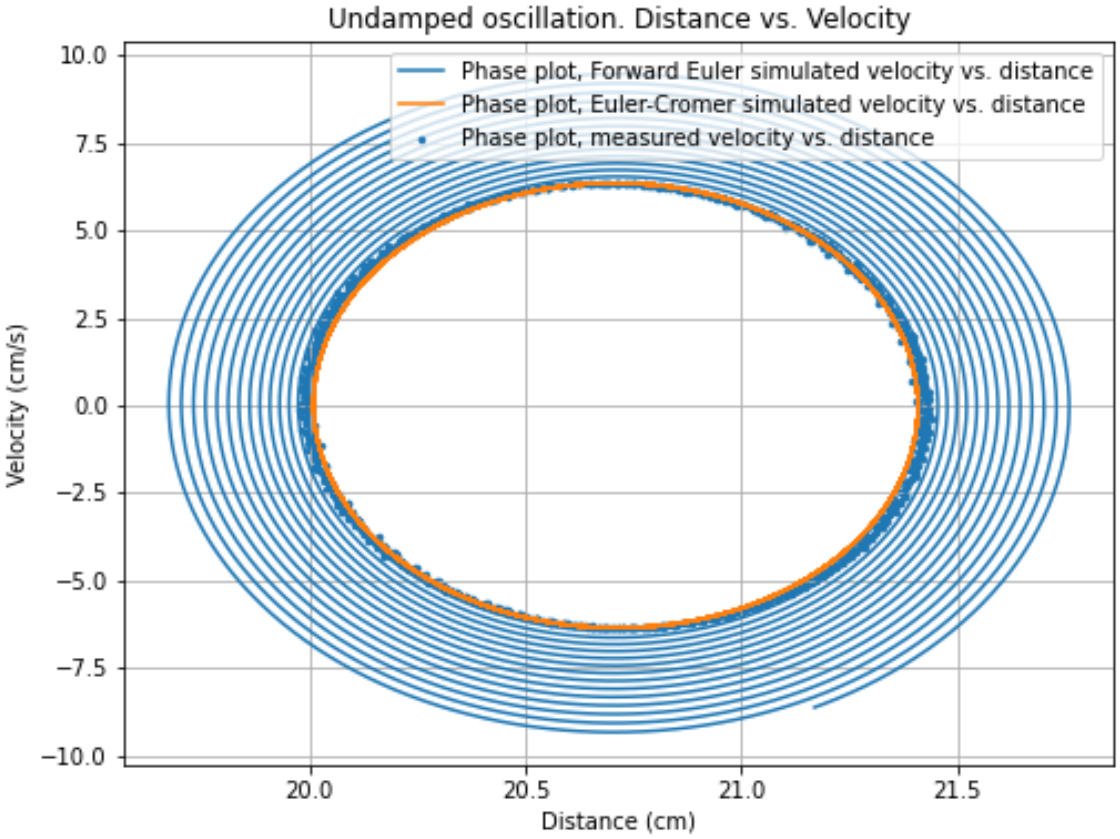


Figure 2: Undamped Oscillations: Velocity vs. Distance

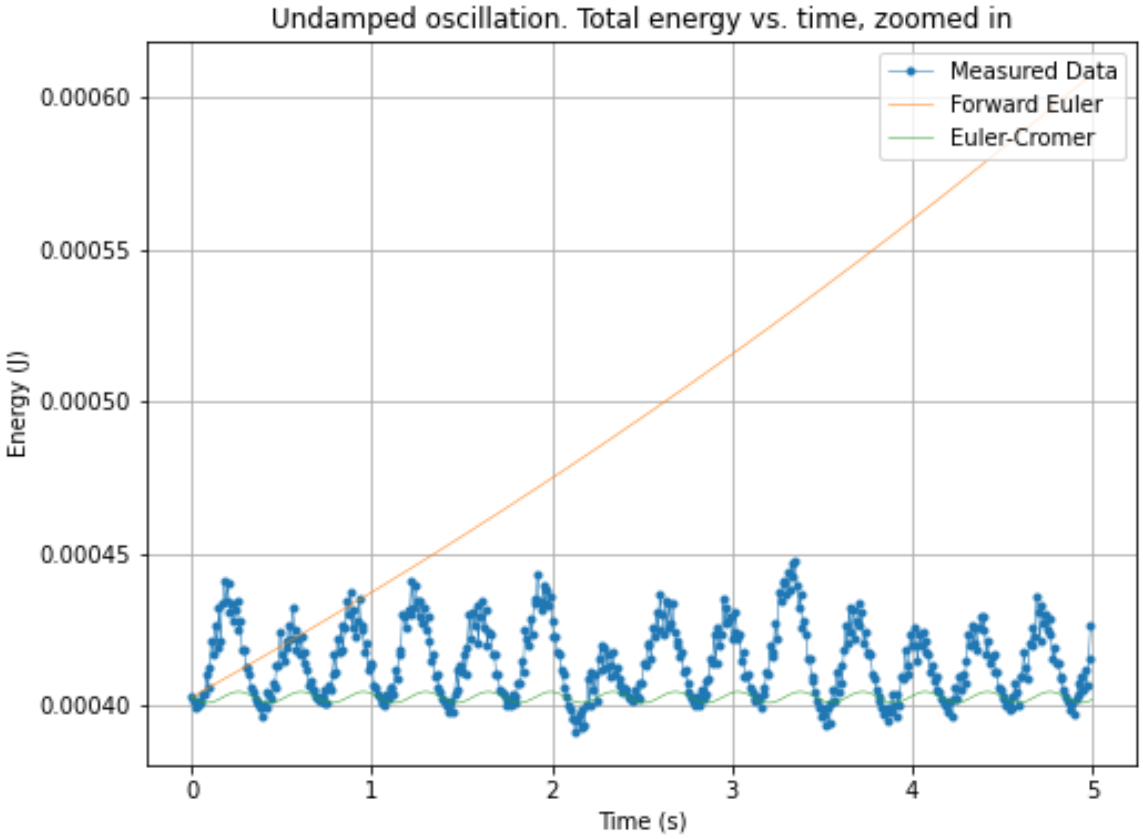


Figure 3: Undamped Oscillations: Total Energy vs. Time (zoomed in)

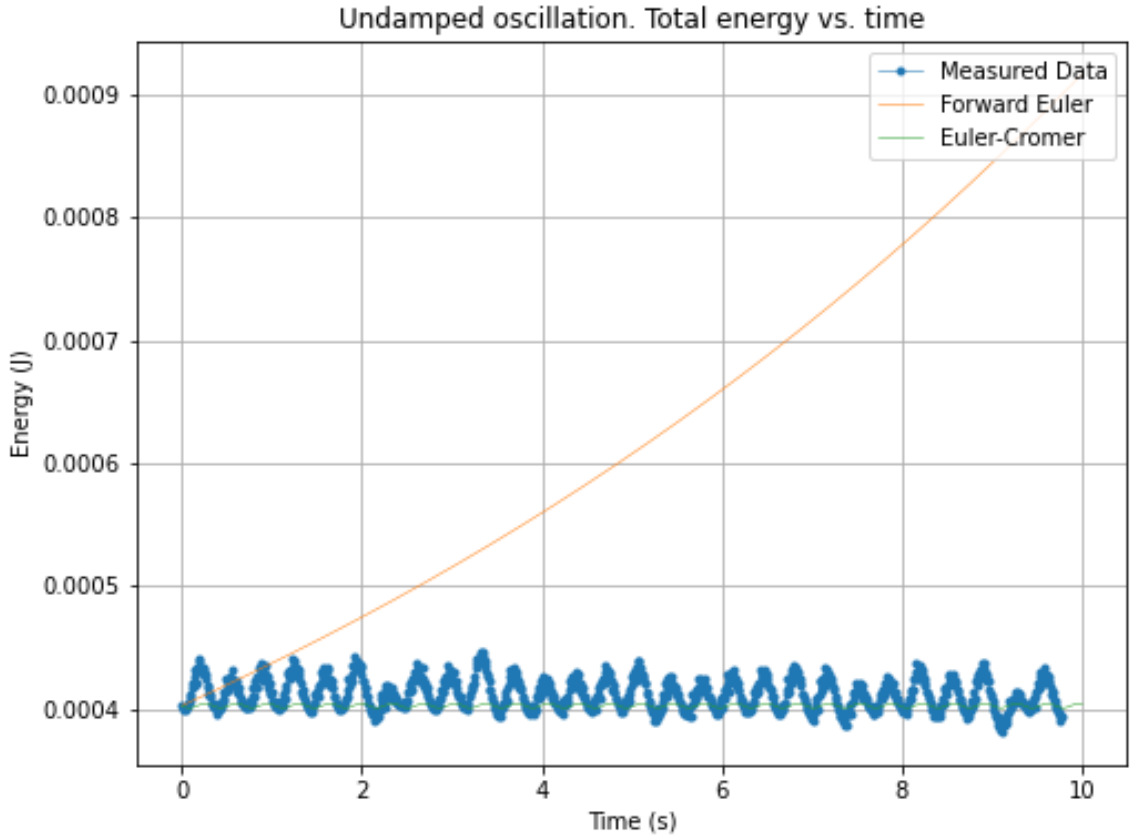


Figure 4: Undamped Oscillations: Total Energy vs. Time

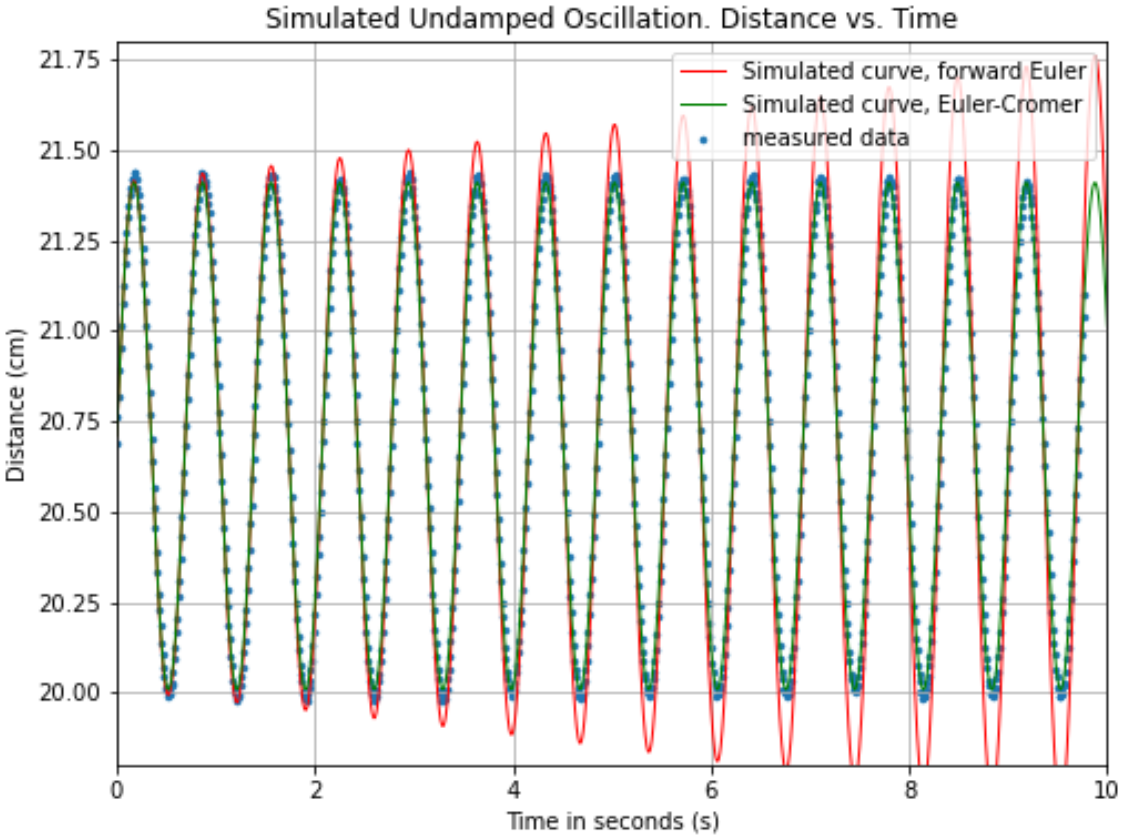


Figure 5: Undamped Oscillations: Simulated Undamped Oscillation. Distance vs. Time

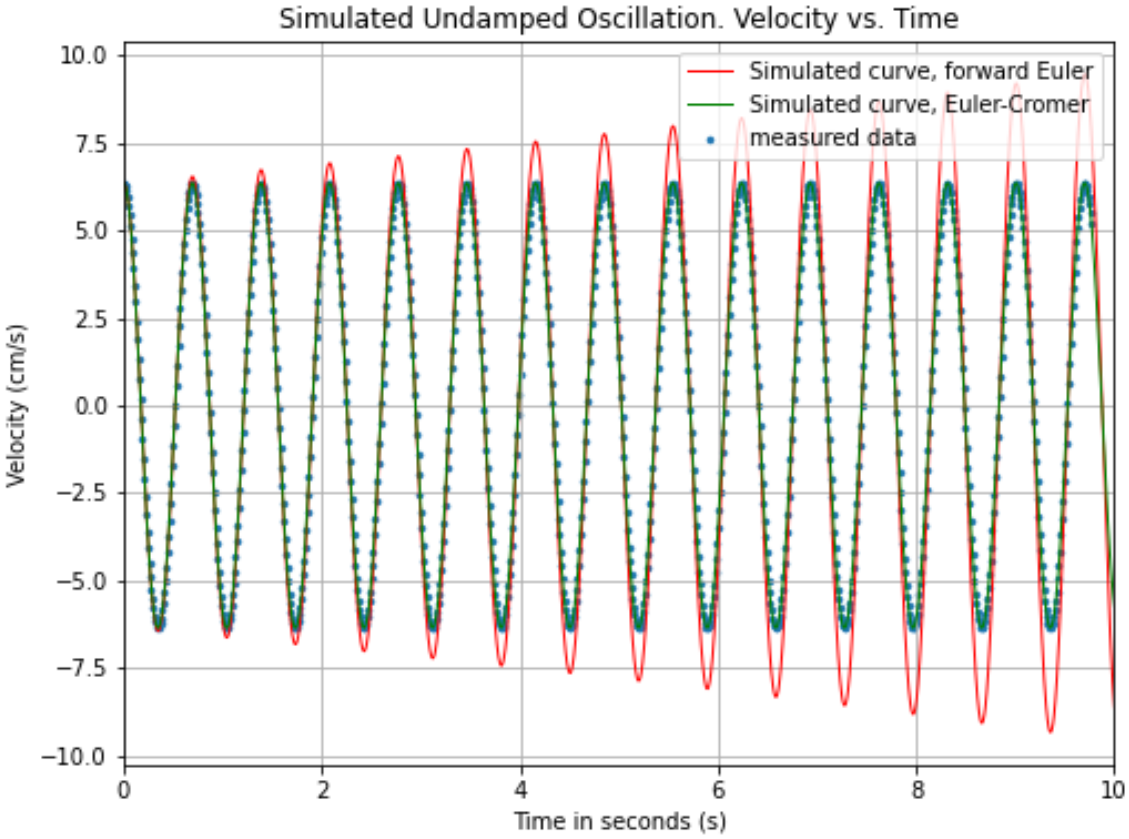
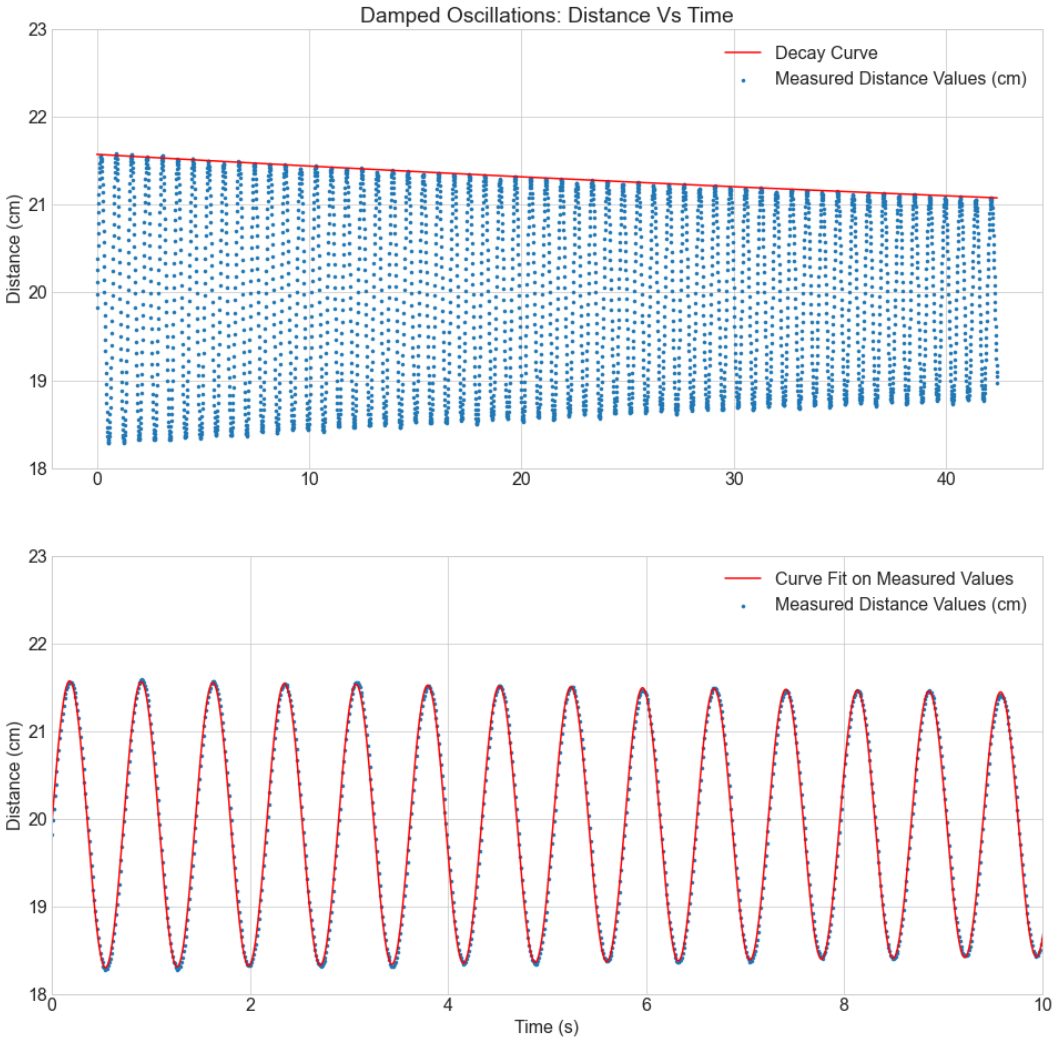


Figure 6: Undamped Oscillations: Simulated Undamped Oscillation. Velocity vs. Time

A.2 Plots For Damped Spring-Mass System



Model Function : $y = y_0 + Ae^{-\gamma t}\sin(\Omega_0 t)$
y_0 = Initial Position = 19.94 cm
A = Amplitude = 1.64 cm
γ = Damping Coefficient = 0.0085
Ω_0 = Frequency of Oscillations = $2\pi/0.723$ rad/s

Figure 7: Damped Oscillations: Distance vs. Time

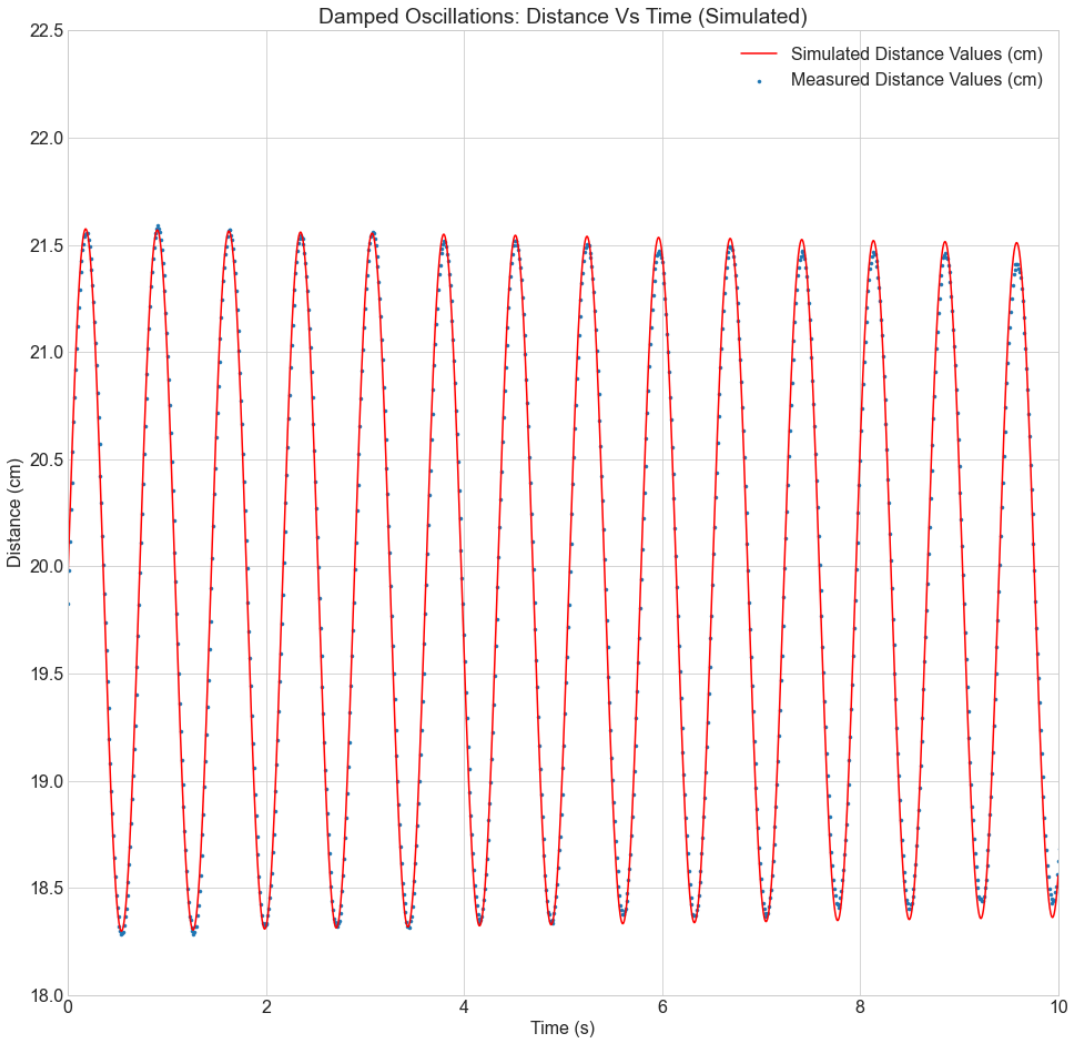


Figure 8: Damped Oscillations: Distance vs. Time (Simulated)

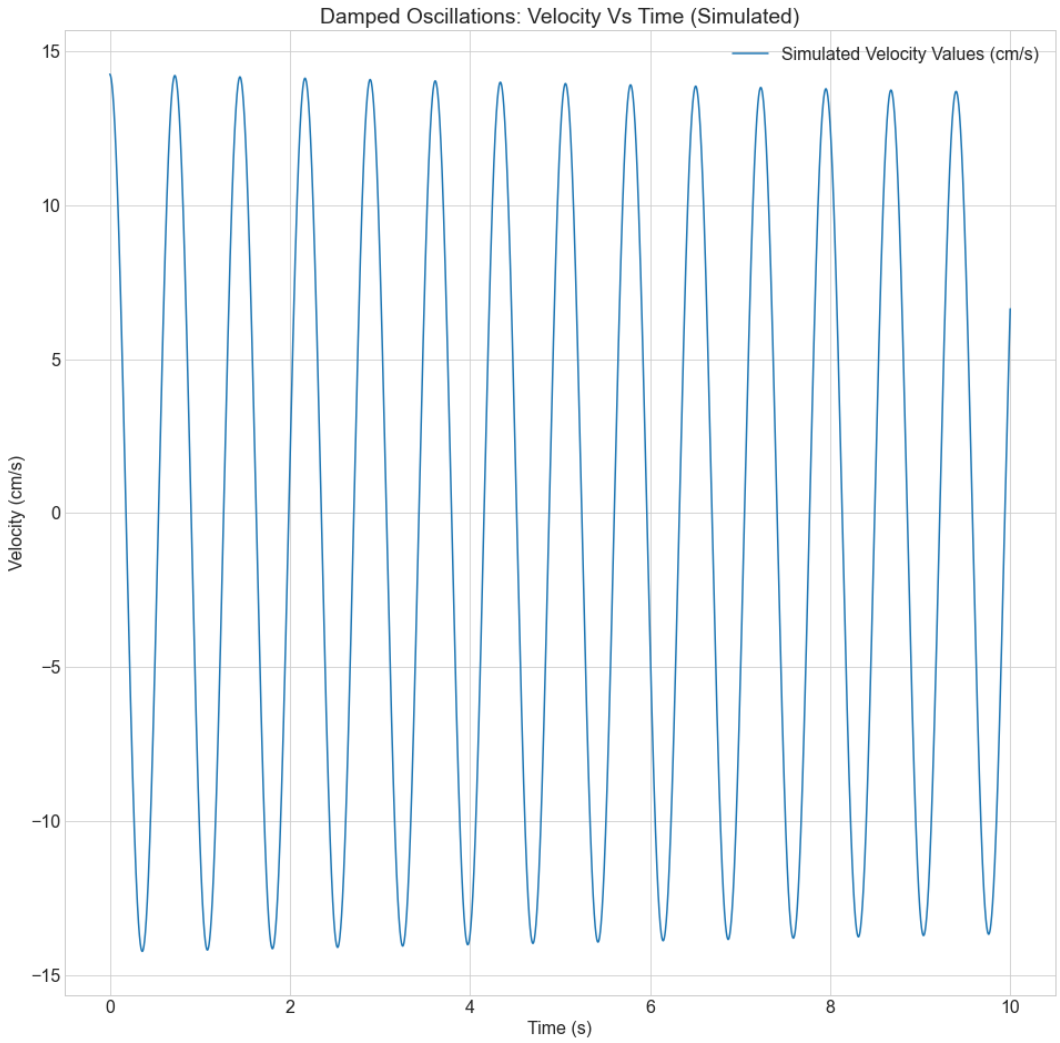


Figure 9: Damped Oscillations: Velocity vs. Time (Simulated)

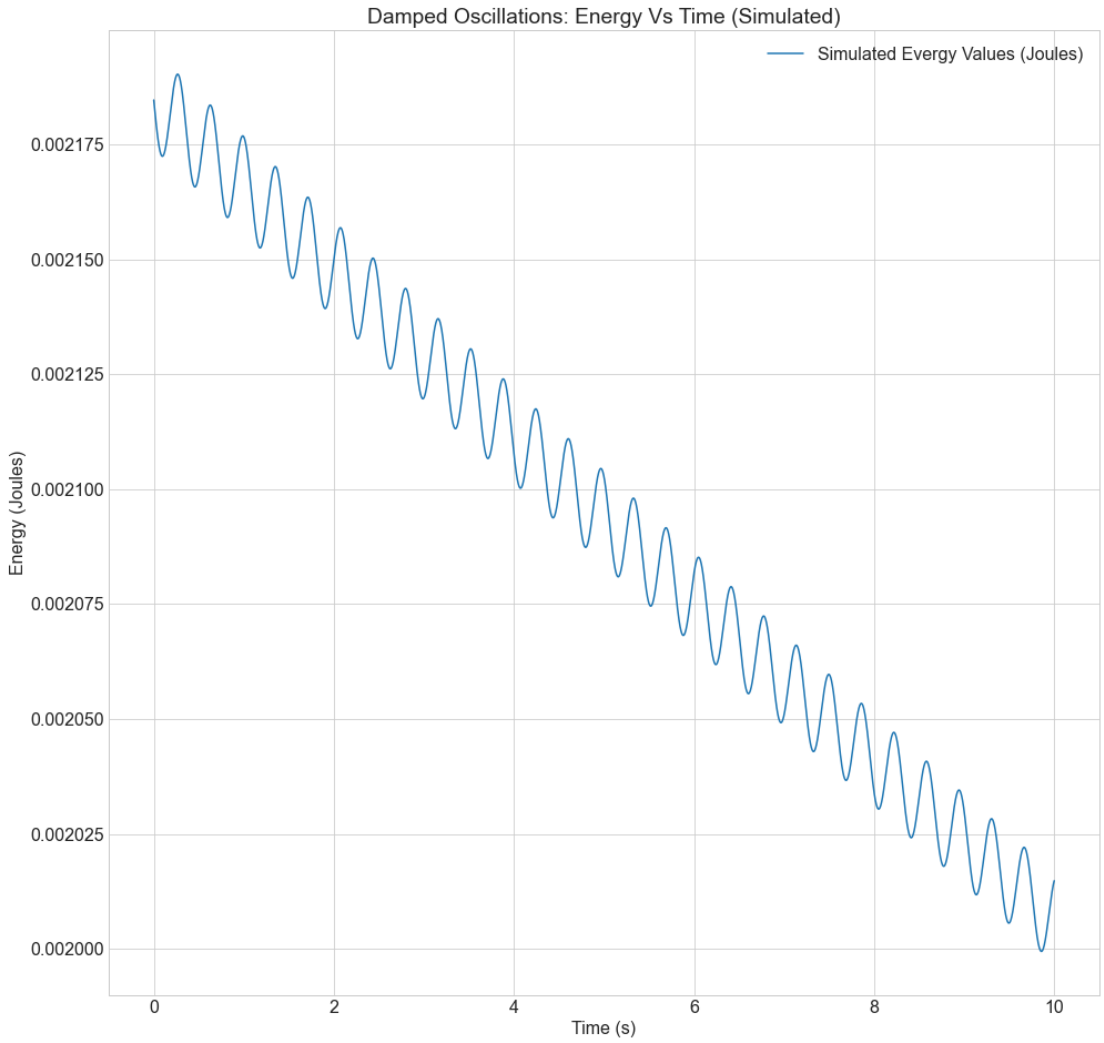


Figure 10: Damped Oscillations: Energy vs. Time (Simulated)

A.3 Python Code: Undamped Oscillations

The Python code for this exercise is divided into two files. Functions.py file contains utility methods which we will be frequently using in this course. Undamped.py file contains the code which analyzes the data.

A.3.1 Functions.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Sep 30 09:46:36 2021

@author: Fredrik
"""
import numpy as np
import scipy.optimize as optim
import matplotlib.pyplot as plt

#Defining the function for curve fitting and plotting
def curve_fit_and_plot(model,initial_guess,xdata,ydata,y_uncer,xunit,yunit,
                       plot_title):
    """
    This function uses the scipy curve_fit function to estimate the parameters
    of the model which will minimize the euclidian distance between our data
    points, and the model curve.
    We print these optimal model parameters along with their uncertainty, and
    plot the original data with error bars, along with the curve fit model.

    Parameters
    ----------
    model : function to be used as model
        model(x,a,b,c,...), where we are estimating a,b,c, etc.
    initial_guess : list of guesses for the parameters a,b,c, etc. eg. [2,4,254]
    xdata : list of input points for the model, eg. [2,4,5,7,9,28]
    ydata : list of output points for the model, eg [23,25,26,85,95,104]
    y_uncer : list of uncertaintes associated with the ydata, which the model
        shall output
    xunit : String describing the unit along the x-axis for label when plotting.
        eg. 'Voltage (V)'
    yunit : String describing the unit along the y-axis for label when plotting.
        eg. 'Current (A)'
    plot_title : String describing the title of the plot.
        eg. 'Current vs. Voltage'
```

```
    Returns None
    """
    #Using the scipy curve fit function to find our model parameters
    p_opt , p_cov = optim.curve_fit(model , xdata , ydata, p0 = initial_guess,
                                    sigma = y_uncer, absolute_sigma = True )
    p_std = np.sqrt( np.diag ( p_cov ))

    print("The optimal values for our curve fit model parameters, are:",np.round(p_opt,2))
    print("Their associated uncertainties  are:", np.round(p_std,2))

    #Now we create some data points on the model curve for plotting
    xvalues_for_plot = np.linspace(xdata[0],xdata[-1],1000)
    yvalues_for_plot = []
    for i in xvalues_for_plot:
        yvalues_for_plot.append(model(i,p_opt[0],p_opt[1]))

    #Now we plot the original data with error bars, along with the curve fit model
    plt.figure(figsize=(10,5))
    plt.errorbar(xdata,ydata,y_uncer,c='r', ls='', marker='o',lw=1,capsize=2,
                 label = 'Points of measurement with uncertainty')
    plt.plot(xvalues_for_plot,yvalues_for_plot, c='b',
             label = 'Scipy curve fit')
    plt.title(plot_title)
    plt.xlabel(xunit)
    plt.ylabel(yunit)
    plt.legend()
    plt.grid()
    plt.savefig(plot_title+'.png')
    plt.show()

    return None

def error_plot(model,p_opt,xdata,ydata,y_uncer,xunit,yunit,
                        plot_title):
    #Now we create some data points on the model curve for plotting
    xvalues_for_plot = np.linspace(xdata[0],xdata[-1],1000)
    yvalues_for_plot = []
    for i in xvalues_for_plot:
        yvalues_for_plot.append(model(i,p_opt[0],p_opt[1]))

    #Now we plot the original data with error bars, along with the curve fit model
    plt.figure(figsize=(10,5))
    plt.errorbar(xdata,ydata,y_uncer,c='r', ls='', marker='o',lw=1,capsize=2,
                 label = 'Points of measurement with uncertainty')
    plt.plot(xvalues_for_plot,yvalues_for_plot, c='b',
             label = 'Scipy curve fit')
```

```
    plt.title(plot_title)
    plt.xlabel(xunit)
    plt.ylabel(yunit)
    plt.legend()
    plt.grid()
    plt.savefig(plot_title+'.png')
    plt.show()
    return None

def chi2(y_measure,y_predict,errors):
    """Calculate the chi squared value given a measurement with errors and
    prediction"""
    return np.sum( np.power(y_measure - y_predict, 2) / np.power(errors, 2) )

def chi2reduced(y_measure, y_predict, errors, number_of_parameters):
    """Calculate the reduced chi squared value given a measurement with errors
    and prediction, and knowing the number of parameters in the model."""
    return chi2(y_measure, y_predict, errors)/ \
            (y_measure.size - number_of_parameters)

def read_data(filename, Del, skiprows, usecols=(0,1)):
    """Load give\n file as csv with given parameters,
    returns the unpacked values"""
    return np.loadtxt(filename,
                      skiprows=skiprows,
                      usecols=usecols,
                      delimiter=Del,
                      unpack=True)

def fit_data(model_func, xdata, ydata, yerrors, guess):
    """Utility function to call curve_fit given x and y data with errors"""
    popt, pcov = optim.curve_fit(model_func,
                                 xdata,
                                 ydata,
                                 absolute_sigma=True,
                                 sigma=yerrors,
                                 p0=guess)

    pstd = np.sqrt(np.diag(pcov))
    return popt, pstd
```

A.3.2 Undamped.py

```
#Importing modules
import numpy as np
import matplotlib.pyplot as plt
import Functions as F


##Undamped Oscillation:

#Specifying modelling function
def model(t,a,b,c):
    return a+b*np.sin(c*t)

#Importing data
Time, Distance = F.read_data('undamped_point_data_set2.txt', None,2)

#Defining Constants
Sample_time = 0.01 #seconds
m = 200.0/1000 #Kilograms
m_uncertainty = 0.1/1000 #kilograms

#Specifying parameters for the model function
a = np.mean(Distance)
b = 0.7
c = (2*np.pi)/0.693

#Calculating the spring constant based on these parameters
spring_constant = m * c**2
print("The spring constant of the string estimated in the undamped system",
      "exercise, is:", spring_constant, "kg/s^2")

#Offsetting time array
Time = np.array([i-0.49 for i in Time])

#Plotting data points and model curve
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(1,1,1)
ax.plot(Time, Distance, marker='.',lw=0.5,label='measured data')
ax.plot(Time, model(Time,a,b,c),lw=1,label='fitted curve')
ax.set_ylim((19.9, 21.75))
ax.legend(loc=1)
ax.set_xlabel("Time in seconds (s)")
ax.set_ylabel("Distance from sensor in centimeters (cm)")
ax.set_title("Undamped oscillation. Distance vs. Time")
```

```
ax.grid()
ax.figure.savefig("Undamped oscillation. Distance vs. Time"+".png")
plt.show()

print("Our specified parameters produce a curve which fit our data vert well.")

## With the parameters we found, we do a simulation and compare it to our data:
#We use a Euler Forward timestep approach
dt = 1/1000 #our timestep will be dt.
t_sim = np.linspace(0,10,int(1/dt*10)) #array of time points
y_sim = np.zeros(len(t_sim)) #so far empty array of relative distances
v_sim = np.zeros(len(t_sim)) #The same for velocities
v_sim[0] = b*c#cm/s. I adjusted this parameter such that
                #our simulation fit our data
y_sim[0] = np.mean(Distance)
#Now we perform the simulation, looping forward in time:
for i in range(len(t_sim)-1):
    v_sim[i+1] = v_sim[i]-dt*spring_constant/m*(y_sim[i]-np.mean(Distance))
    #*100 for m->cm in sping constant
    y_sim[i+1] = y_sim[i]+dt*v_sim[i]

#Euler-Cromer method:
y_sim_2 = np.zeros(len(t_sim)) #so far empty array of relative distances
v_sim_2 = np.zeros(len(t_sim)) #The same for velocities
v_sim_2[0] = b*c #cm/s. I adjusted this parameter such that
               #our simulation fit our data
y_sim_2[0] = np.mean(Distance)
#Now we perform the simulation, looping forward in time:
for i in range(len(t_sim)-1):
    y_sim_2[i+1] = y_sim_2[i]+dt*v_sim_2[i]
    v_sim_2[i+1] = v_sim_2[i]-dt*spring_constant/m*(y_sim_2[i+1]-np.mean(
        Distance))
    #*100 for m/s->cm/s


#we then plot our simulated curves along with our data.
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(1,1,1)
ax.scatter(Time, Distance, marker='.',lw=0.5,label='measured data')
ax.legend(loc=1)
ax.set_title("Simulated Undamped Oscillation. Distance vs. Time")
ax.plot(t_sim, y_sim,lw=1,label='Simulated curve, forward Euler',c='r')
ax.plot(t_sim, y_sim_2,lw=1,label='Simulated curve, Euler-Cromer',c='g')
ax.set_ylim((19.8, 21.8))
ax.legend(loc=1)
ax.set_xlabel("Time in seconds (s)")
ax.set_ylabel("Distance (cm)")
```

```
ax.set_xlim(0,10)
ax.grid()
ax.figure.savefig("Simulated Undamped Oscillation. Distance vs. Time"+".png")
plt.show()

print("We see that the Euler-Cromer simulation fit our data over time much",
      " better than the Forward Euler simulation, which amplitude grows",
      " significantly in time. (Depending on our time step size ofc.).",
      " The amplitude should however be constant, if not be weakly decreasing,",
      " due to small unavoidable damping in our system of experiment.")



#Phase and Velocity plot
Velocity = b*c*np.cos(c*Time) #Using the time derivative of our curve fit model

#Velocity plot
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(1,1,1)
ax.scatter(Time, Velocity, marker='.',lw=0.5,label='measured data')
ax.legend(loc=1)
ax.set_title("Simulated Undamped Oscillation. Velocity vs. Time")
ax.plot(t_sim, v_sim,lw=1,label='Simulated curve, forward Euler',c='r')
ax.plot(t_sim, v_sim_2,lw=1,label='Simulated curve, Euler-Cromer',c='g')
ax.legend(loc=1)
ax.set_xlabel("Time in seconds (s)")
ax.set_ylabel("Velocity (cm/s)")
ax.set_xlim(0,10)
ax.grid()
ax.figure.savefig("Simulated Undamped Oscillation. Velocity vs. Time"+".png")
plt.show()

#Phase plot
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(1,1,1)
ax.scatter(Distance,Velocity,marker='.',lw=0.5,
           label='Phase plot, measured velocity vs. distance')
ax.plot(y_sim,v_sim,
        label='Phase plot, Forward Euler simulated velocity vs. distance')
ax.plot(y_sim_2,v_sim_2,
        label='Phase plot, Euler-Cromer simulated velocity vs. distance')
ax.legend(loc=1)
ax.set_xlabel("Distance (cm)")
ax.set_ylabel("Velocity (cm/s)")
ax.set_title("Undamped oscillation. Distance vs. Velocity")
ax.grid()
ax.figure.savefig("Undamped oscillation. Distance vs. Velocity"+".png")
```

```
plt.show()

print("As expected, we get elliptical phase plots for our measured and",
      " simulated Distance and Velocity, however, as we will see in the",
      " energy plot, the Forward Euler simulation grows in energy, which",
      "causes the phase space plot of the Forward Euler simulation ellipse to",
      " increase in radius over time. This also corresponds to the increase",
      " in amplitude in the position and velocity plot over time.")



#Energy vs time plot
#Dividing by 100 to get from cm and cm/s to m and m/s. Etot=Ekin+Epot
Etot = m*(Velocity/100)**2/2 + spring_constant*(
    (Distance-np.mean(Distance))/100)**2/2
Etot_sim = m*(v_sim/100)**2/2 + spring_constant*(
    (y_sim-np.mean(Distance))/100)**2/2
Etot_sim_2 = m*(v_sim_2/100)**2/2 + spring_constant*(
    (y_sim_2-np.mean(Distance))/100)**2/2

fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(1,1,1)
ax.plot(Time,Etot,lw=0.5,marker='.', label='Measured Data')
ax.plot(t_sim,Etot_sim,lw=0.5,label='Forward Euler')
ax.plot(t_sim,Etot_sim_2,lw=0.5,label='Euler-Cromer')
ax.legend(loc=1)
ax.set_xlabel("Time (s)")
ax.set_ylabel("Energy (J)")
ax.set_title("Undamped oscillation. Total energy vs. time")
ax.grid()
ax.figure.savefig("Undamped oscillation. Total energy vs. time"+".png")
plt.show()

#Zoomed in
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(1,1,1)
ax.plot(Time[0:500],Etot[0:500],lw=0.5,marker='.', label='Measured Data')
ax.plot(t_sim[0:5000],Etot_sim[0:5000],lw=0.5,label='Forward Euler')
ax.plot(t_sim[0:5000],Etot_sim_2[0:5000],lw=0.5,label='Euler-Cromer')
ax.legend(loc=1)
ax.set_xlabel("Time (s)")
ax.set_ylabel("Energy (J)")
ax.set_title("Undamped oscillation. Total energy vs. time, zoomed in")
ax.grid()
ax.figure.savefig(
    "Undamped oscillation. Total energy vs. time, zoomed in"+".png")
plt.show()
```

```
print("As we can see from the total Energy plots, the total energy of the",
      "system, is approximately conserved for the Euler-Cromer simulation,",
      " grows in time (unphysical) and is not conserved in the Forward Euler",
      " simulation, and is approximately conserved for our measured data,",
      " when accounting for uncertainties in our measurements of the distance",
      " over time. In theory, total energy should be conserved, but the",
      " energy will oscillate to be in the form of potential and kinetic",
      " energy.")

print("The radius of the elliptical phase plots, correspond to the total",
      " energy of the system. When considering our total energy plots over",
      " time, it makes sense that the Euler Cromer and measured data plots",
      " give approximately stable phase ellipses, though the Forward Euler",
      " phase plot has increasing radius with time, as its energy is",
      " increasing with time. The reason the radius should be constant, is",
      " because the total energy should be constant. The x and y component of",
      " the radius oscillate in their contribution to the radius length,",
      " corresponding to how the total energy is conserved, but oscillates",
      " in being in the form of kinetic and potential energy.")
```

A.4 Python Code: Damped Oscillations

The Python code for this exercise is divided into two files. statslab.py file contains utility methods which we will be frequently using in this course. lab_3_damped_code.py file contains the code which analyzes the data.

A.4.1 statslab.py

```
import numpy as np
import scipy.optimize as optim
import matplotlib.pyplot as plt


##############################################################################
# Utility Methods Library
#
# This file contains some utility method which are common to our data analysis.
# This library also contains customized plotting methods.
##############################################################################

# use bigger font size for plots
plt.rcParams.update({'font.size': 16})

def chi2(y_measure,y_predict,errors):
    """Calculate the chi squared value given a measurement with errors and
    prediction"""
    return np.sum( np.power(y_measure - y_predict, 2) / np.power(errors, 2) )

def chi2reduced(y_measure, y_predict, errors, number_of_parameters):
    """Calculate the reduced chi squared value given a measurement with errors
    and prediction, and knowing the number of parameters in the model."""
    return chi2(y_measure, y_predict, errors)/ \
            (y_measure.size - number_of_parameters)

def read_data(filename, skiprows=1, usecols=(0,1), delimiter=","):
    """Load give\n file as csv with given parameters,
    returns the unpacked values"""
    return np.loadtxt(filename,
                      skiprows=skiprows,
                      usecols=usecols,
                      delimiter=delimiter,
                      unpack=True)
```

```
def fit_data(model_func, xdata, ydata, yerrors, guess=None):
    """Utility function to call curve_fit given x and y data with errors"""
    popt, pcov = optim.curve_fit(model_func,
                                 xdata,
                                 ydata,
                                 absolute_sigma=True,
                                 sigma=yerrors,
                                 p0=guess)

    pstd = np.sqrt(np.diag(pcov))
    return popt, pstd

# y = ax+b
def linear_regression(xdata, ydata):
    """Simple linear regression model"""
    x_bar = np.average(xdata)
    y_bar = np.average(ydata)
    a_hat = np.sum( (xdata - x_bar) * (ydata - y_bar) ) / \
            np.sum( np.power((xdata - x_bar), 2) )
    b_hat = y_bar - a_hat  * x_bar
    return a_hat, b_hat

class plot_details:
    """Utility class to store information about plots"""
    def __init__(self, title):
        self.title = title
        self.x_log_scale = False
        self.y_log_scale = False

    def errorbar_legend(self, v):
        self.errorbar_legend = v
    def fitted_curve_legend(self, v):
        self.fitted_curve_legend = v
    def x_axis_label(self, v):
        self.x_axis_label = v
    def y_axis_label(self, v):
        self.y_axis_label = v
    def xdata(self, x):
        self.xdata = x
    def ydata(self, y):
        self.ydata = y
    def yerrors(self, y):
        self.yerrors = y
    def xdata_for_prediction(self, x):
        self.xdata_for_prediction = x
    def ydata_predicted(self, y):
        self.ydata_predicted = y
```

```
    def legend_position(self, p):
        self.legend_loc = p
    def chi2_reduced(self, c):
        self.chi2_reduced = c
    def set_x_log_scale(self, c):
        self.x_log_scale = c
    def set_y_log_scale(self, c):
        self.y_log_scale = c

def plot(plot_details, new_figure=True, error_plot=True):
    """Utility method to plot errorbar and line chart together,
    with given arguments"""
    if new_figure:
        fig  = plt.figure(figsize=(16, 10))
        fig.tight_layout()
        plt.style.use("seaborn-whitegrid")

    # plot the error bar chart
    if error_plot:
        plt.errorbar(plot_details.xdata,
                     plot_details.ydata,
                     yerr=plot_details.yerrors,
                     marker="o",
                     label=plot_details.errorbar_legend,
                     capsize=2,
                     ls="")

    # plot the fitted curve
    plt.plot(plot_details.xdata_for_prediction,
             plot_details.ydata_predicted,
             label=plot_details.fitted_curve_legend)

    # legend and title
    plt.title(plot_details.title)
    plt.xlabel(plot_details.x_axis_label)
    plt.ylabel(plot_details.y_axis_label)

    if plot_details.x_log_scale:
        plt.xscale("log")

    if plot_details.y_log_scale:
        plt.yscale("log")

    legend_pos = "upper left"
    if hasattr(plot_details, "legend_loc"):
        legend_pos = plot_details.legend_loc
```

```
    plt.legend(loc=legend_pos)
```

A.4.2 lab_3_damped_code.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@author: Pankaj Patil
"""

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

damped_mass = 215.1 # in grams
damped_mass_uncertainty = 0.1 # in grams

# model function for damped displacement
def displacement_damped(time, gamma):
    return y_0 + amplitude * np.power(math.e, -gamma*time) * np.sin(omega*time)

# model function for decay in dispacement
def displacement_decay(time, gamma):
    return y_0 + amplitude * np.power(math.e, -gamma*time)

# load the data file
damped_filename = "../data/damped_point_data.txt"
measured_time, measured_distance = utils.read_data(damped_filename,
                                                   usecols=(0,1),
                                                   skiprows=2,
                                                   delimiter=None)

# offset the  time values to start at t=0
measured_time = measured_time - measured_time[0]

# the initial position is mean  of  the measured  distances
y_0 = np.mean(measured_distance)

# model parameter values
amplitude = 1.64
gamma = 0.0085
omega = 2*math.pi / 0.723

# compute the spring constant
spring_constant = damped_mass * omega ** 2
```

```
print("Amplitude of Oscillations = %.4f cm"
      % amplitude)
print("Initial Position = %.2f cm"  % y_0)
print("Damping Coefficient = %.4f"  % gamma)
print("Frequency of Oscillations = %.4f rad/s"  % omega)
print("Estimated Spring Constant (Damped Oscillations) = %.4f g/s^2"
      % spring_constant)

def plot_measured_data():
    # compute the predicted displacement  values using our model function
    predicted_displacement = displacement_damped(measured_time, gamma)

    # create  a figure  for  out subplots
    plt.figure(figsize=(16, 16))
    plt.style.use("seaborn-whitegrid")

    # plot the measured data for Distance vs Time
    plt.subplot(2, 1, 1)
    plt.scatter(measured_time, measured_distance,
                marker='.',lw=0.5,
                label="Measured Distance Values (cm)")

    # plot the envelop using decay curve with gamma as decay coefficient
    plt.plot(measured_time, displacement_decay(measured_time, gamma),
             label="Decay Curve", color="r")

    plt.title("Damped Oscillations: Distance Vs Time")
    axes = plt.gca()
    axes.set_ylim(18, 23)

    plt.ylabel("Distance (cm)")
    plt.legend(loc="upper right")

    # plot the measured data for Distance vs Time
    plt.subplot(2, 1, 2)
    plt.scatter(measured_time, measured_distance,
                marker='.',lw=0.5,
                label="Measured Distance Values (cm)")

    plt.plot(measured_time, predicted_displacement,
             label="Curve Fit on Measured Values", color="r")

    plt.xlabel("Time (s)")
    plt.ylabel("Distance (cm)")
    plt.legend(loc="upper right")

    axes = plt.gca()
```

```
    axes.set_xlim(0, 10)
    axes.set_ylim(18, 23)

    # save the plot
    plt.savefig("lab3_damped_distance_vs_time.png")


def plot_simulated_data():
    # time spte of 0.001 seconds
    dt = 0.001
    time = np.linspace(0, 10, int(1/dt*10))
    y = np.zeros_like(time)
    v = np.zeros_like(time)
    energy = np.zeros_like(time)

    # initialize the first elements of our velocity and distance arrays
    v[0] = omega * amplitude
    y[0] = y_0

    # use Euler-Cromer method for simulation
    for i in range(len(time)-1):
        y[i+1] = y[i] + dt * v[i]
        v[i+1] = v[i] - dt * (omega ** 2) *  (y[i+1]-y_0) - gamma * v[i] * dt
        energy[i] = 0.5 * damped_mass * (v[i] ** 2) + \
            0.5 * spring_constant * (y[i]-y_0) ** 2

    # create a figure for our plots
    plt.figure(figsize=(16, 16))
    plt.style.use("seaborn-whitegrid")

    # plot the simulated data for Distance vs Time
    plt.scatter(measured_time, measured_distance,
                marker='.',lw=0.5,
                label="Measured Distance Values (cm)")

    # plot distance vs time
    plt.plot(time, y,
             label="Simulated Distance Values (cm)", color="r")

    plt.xlabel("Time (s)")
    plt.ylabel("Distance (cm)")
    plt.legend(loc="upper right")
    plt.title("Damped Oscillations: Distance Vs Time (Simulated)")

    axes = plt.gca()
    axes.set_xlim(0, 10)
    axes.set_ylim(18, 22.5)
```

```
    # save the plot
    plt.savefig("lab3_damped_sim_distance_vs_time.png")

    # create  a figure for our plots
    plt.figure(figsize=(16, 16))
    plt.style.use("seaborn-whitegrid")

    # plot velocity vs time
    plt.plot(time, v,
             label="Simulated Velocity Values (cm/s)")

    plt.xlabel("Time (s)")
    plt.ylabel("Velocity (cm/s)")
    plt.legend(loc="upper right")
    plt.title("Damped Oscillations: Velocity Vs Time (Simulated)")

    # save the plot
    plt.savefig("lab3_damped_sim_velocity_vs_time.png")

    # create a figure for our plots
    plt.figure(figsize=(16, 16))
    plt.style.use("seaborn-whitegrid")

    # plot energy vs time
    plt.plot(time[:-1], energy[:-1] / np.power(10, 7),
             label="Simulated Evergy Values (Joules)")

    plt.xlabel("Time (s)")
    plt.ylabel("Energy (Joules)")
    plt.legend(loc="upper right")
    plt.title("Damped Oscillations: Energy Vs Time (Simulated)")

    # save the plot
    plt.savefig("lab3_damped_sim_energy_vs_time.png")


# plot the measured data
plot_measured_data()

# plot simulated  data
plot_simulated_data()
```

References

[1] Lab Manual - Spring Mass - exercise4_NI.pdf