Karl Hipius
8/6/19
PHY 495, Summer Research

Modeling Quantum Mechanical systems with a Coupled Pendula

I. Introduction

The research conducted over the summer served to model a coupled pendula system with computer simulation. The program used was Python and within this program many useful programming techniques were utilized to help model the couple pendulum system. Once this system was modeled, a comparison could be made between the coupled pendula and a quantum mechanical system using similar mathematical equations. In the end, a comparison of amplitude versus frequency was made and will be provided in a subsequent section. In the sections that follow I will be discussing the physical system being modeled, the overall structure of my Python script, making a comparison between integrator techniques including reasons behind a time varying damping parameter, and will provide a resonance curve displaying amplitude versus frequency of a mass.

II. The Physical System

This section will serve to provide an explanation of the physical system that is being modelled as well as provide a discussion of Newton's laws and how the force equation comes about. The physical system that we are using to model quantum mechanical phenomena is a coupled pendulum. Our system currently consists of 22 masses of mass m(with 2 additional dummy masses), connected in series via hooks with springs of spring constant κ_{sp}, and are attached also by hook and fishing line to a pivot point P. Plastic blocks were constructed for the purpose of having the capability to change the effective pendulum length L as well as have all pendula oscillate in the same plane. Each mass in the system was separated equally in distance. The system is represented, in part, in figure 1.

Figure 1: The Physical System

The oscillator that we have chosen to drive the system has a power of 0.5hp. The driver effectively drives the 0_{th} pendulum at various frequencies. Once this pendulum is displaced, the force provided by the driven is then transferred down the line of masses until it reaches a potential well.

To describe the force acting on the system, Newton's second law is used at various points in time. The equation that was used to analyze the system is shown below in (1).

$$m_i \frac{\partial^2}{\partial t^2} \Delta_i = m_i \frac{g}{L_i} \Delta_i + k_{sp}(\Delta_{i+1} - 2\Delta_i + \Delta_{i-1}) \qquad (1)$$

This equation is just a different form of Newton's second law with the ΣF term on the left-hand side of the equation and the sum of the forces acting on the system on the right side. The left-hand side is looking at the ith mass where Δ is the displacement of the mass in the horizontal direction. Taking the second derivative of Δ provides us with an acceleration of the mass (ma). The first term on the right-hand side of the equation is caused by the force of gravity. It uses the small angle approximation for sin(θ) to be θ. Then, θ is equal to Δ_i / L_i, where L_i is the pendulum length. The other term on the right-hand side of the equation comes about because of the presence of the spring. The term is basically saying that the force on the system from the spring depends on the position of the observed mass, Δ_i, as well as the masses before and after that mass, Δ_{i+1} and Δ_{i-1}. The variable k_{sp} is the spring constant for each of the springs attached to the masses.

III. Discussion of Python Script Structure/Functions

This section will serve to provide information regarding the structure as well as the reasoning behind the structure of my Python script. The script consisted of seven functions, each with the purpose of simplifying the complex script to specific functions and only calling them when needed. In other words, to run the simulation, not every line of code was necessary which ultimately decreased the amount of time needed to run the simulation. Calling specific functions within the script made running the simulations quicker and easier organize runs. The first function in the script, after proper imports were completed, was calc(). Within calc, proper parameters were defined as well as the dimensional arrays that are being used to observe the coupled pendula system. Once these conditions for the pendula were set, there is a loop that defines the damping constant that will be present in our system. This damping constant was set to change over time (which will be discussed formally in IV.) and therefore looping this variable over time was necessary. Once the time varying damping coefficient was determined, conditions for initial position and frequency of the oscillator were defined. Position as well as velocity were defined as time varying quantities. Once position and velocity were defined, two functions of coupled differential equations were defined, and later used for the Runge-Kutta integration (this will be discussed in IV.). After the two functions were defined, the Runge-Kutta integration was run, effectively able to track the movements of all the masses through the course of the simulation. Within the Runge-Kutta integrator, there were two loops, the first to track time and the second to track each of the masses. Once the integration was completed, time, position, velocity, damping, mass, and effective pendulum length were returned.

The second function in the script was called write(). This function had inputs of time, position, velocity, damping coefficient, mass and effective pendulum length. Also, a variable called file was input into the function for the purpose of easily writing simulations into data files, which would later be read from the file. The reason behind this function was to be able to write simulations done in calc() to data files and not have to re-run the same simulation multiple times. This function made it easy to manage each simulation conducted and to be able to access the file whenever necessary.

The third function of the script was called read(). The data file in which one of the simulations was stored was an input to this function. Basically, the write() function wrote the simulation into a file, which is then accessed by read(). This function takes the data that was stored in write() and provides it to the user for the purpose of using it to determine the conditions

of the simulation. Once the data is accessed from the file, it is returned at the end of the function to be used for other functions, which in this case was to determine the maximum amplitude.

To find the maximum amplitude of a mass during the simulation, a separate function called amp() was created. This function had inputs of time, position, ts(starting time), and te(ending time). The purpose of having inputs ts and te was to have a very general function which could easily be edited to find the maximum amplitude found in the simulation at different times. The maximum amplitude function was made separate from calc() because it is much less time consuming and it keeps the script very organized. The plots provided in V. were taken with the start time at 550 seconds and end time of 600 seconds. Within this function there were if statements that allowed the function to run if no initial or final conditions were provided.

The fifth function in this program was called plot(). Similar to amp(), inputs to this function included time, position, ts, and te. However, two additional variables also had to be included in this function, which were velocity and mass number. This function used the same concept as amp() with no specifications needed for ts and te. This function provided plots of position versus time as well as velocity versus time. In doing so, the mass being plotted had to be specified when running the function. The results of running this function for position versus time as well as velocity versus time of the 5^{th} mass is provided in V.

The sixth function in this program is called calc1(). This function has inputs of position, velocity, mass number, and effective pendulum length. This function provides an energy calculation for the system as a function of time. This function contains a loop that passes through time and an additional loop that passes though mass number. As the function runs, sequential values for kinetic and potential energy are found which is why the inputs mentioned previously (position, velocity) are provided. Once this function has run, the energy of a specific mass is returned. The last function in the script is called plot(). This function takes inputs of time and energy (found in calc1) and plots the energy of a mass versus time. Within this function time can be varied to plot the energy of a mass at any time throughout the simulation.

IV. A comparison of Integrator Techniques

In analyzing the pendula system different integrator techniques were investigated. An integrator is used for the simulation because it finds the best analytic solution for sines and cosines present in a system that contains wave motion. All integrators have error associated with them, so it is very important to use one with the smallest error. There is not one integration technique that is effective for all problems, so multiple different techniques need to be compared to find the most accurate representation. Initially, implicit and explicit integrator techniques were used in the simulation to model the system. However, both techniques were not found to be an effective representation of the system because of the large propagation of error associated with them. Each simulation that was run would get less and less accurate to the actual system that was being analyzed as time went on. So, a Runge-Kutta integrator eventually was used for the simulation because of its ability to "cancel" the error associated with each timestep. In contrast to the implicit and explicit techniques, the 4^{th} order Runge-Kutta integrator effectively takes an additional timestep in between the defined timesteps in the program. In doing so, the error is "averaged" and a good approximation for what is happening in between timesteps is found. Within the integrator, a time dependent damping constant was defined.

A. Damping Parameter

The damping parameter used in the simulation was analytically determined and was defined to change over time. It was necessary to have a time dependent damping parameter

because at low enough damping, it was found that the system did not reach equilibrium. The system that we are using has a natural damping constant of 0.001. When the simulation was run with an initial damping parameter equal to that naturally found within the system, equilibrium was not reached for up to 1000 seconds. So, for the system to display harmonic motion, the damping parameter was set to initially be 0.01 and eventually decay to 0.001. Over the course of approximately 400 seconds, the system would reach simple harmonic motion with a damping coefficient of 0.001(the systems natural damping coefficient) and the analysis can be conducted.

V. Results

The main purpose of running the simulation was to acquire a graph that compares the amplitude of the masses versus the frequency of oscillation. However, this section also serves to provide a graph of displacement as well as velocity versus time of a mass(5^{th}). The plot for displacement versus time of the fifth mass at a frequency of 1 Hz is provided below in figure 2.

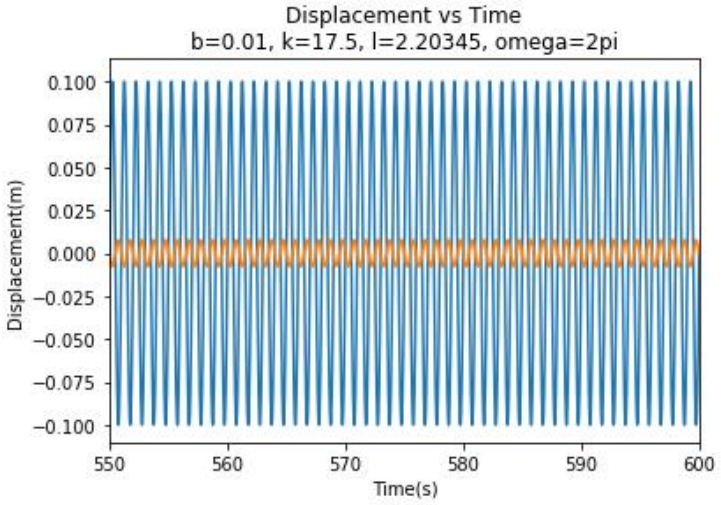

Figure 2: Plot of Amplitude vs Time of the 5^{th} mass

Figure 2 provides a graph of driven amplitude versus time(blue) as well as mass amplitude versus time(orange) of the 5^{th} mass from 550 to 600 seconds. As evidenced by the figure, the driver amplitude was set at 0.1 meters and the amplitude of oscillation of the mass is shown to be approximately 0.01 meters. A time varying damping parameter, a spring constant k_{sp}, and effective pendulum length L were included in the calculation and are provided at the top of the graph. Figure 3 displays a graph of velocity versus time of the same mass at the same driving frequency.

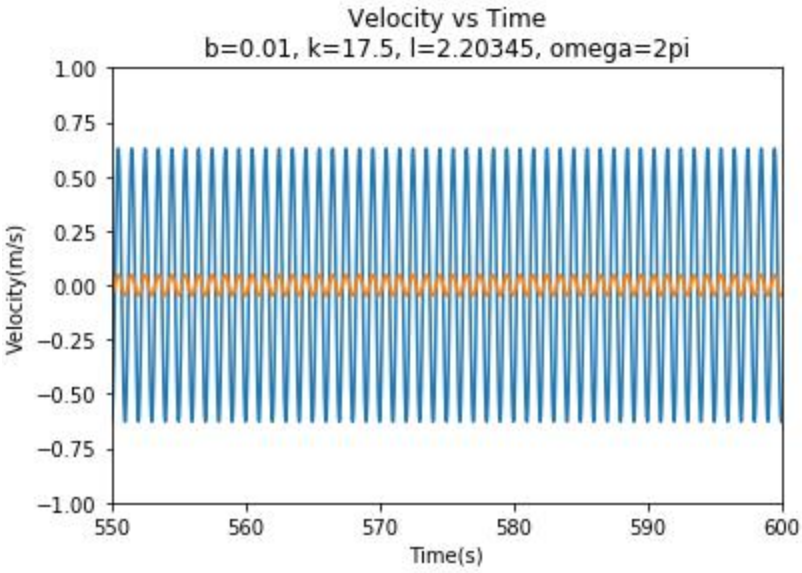

Figure 3: Plot of Velocity vs Time of the 5th mass

As shown in figure 3, the drivers' velocity(blue) is approximately 0.63 m/s, with a corresponding mass velocity(orange) of 0.05 m/s. This graph provides evidence of damping in the system because the mass amplitude and velocity are much less than that of the driver. A graph of amplitude versus frequency was also obtained from the simulations.

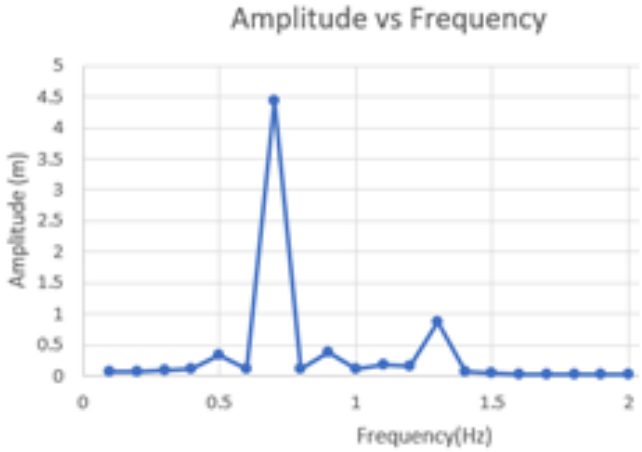

Figure 4: Plot of Amplitude vs Frequency

In figure 4, the maximum amplitude of the coupled pendula system is compared to the frequency at which the driver was oscillating. This plot is representative of a resonance curve with spikes in amplitude at 0.7Hz and 1.3Hz. The spikes in the system appear to go to about 4.5m and 0.9m which is much higher than that of the driver (0.1m). Therefore, this graph doesn't represent what is present in the system, but instead shows the frequencies at which the system naturally oscillates(resonance). Figure 4 was then magnified to see how the system acts when it is not at resonant frequencies.

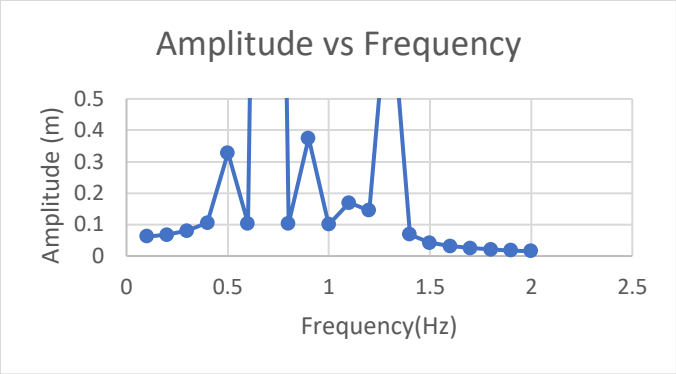

Figure 5: Plot of Amplitude vs Frequency Magnified

The figure shown above serves to provide information on the system before and after resonant frequencies. It is evident from the plot that the system naturally progresses in amplitude to larger values until resonance is reached. However, right before resonance is reached, amplitude oddly declines approximately 0.1m. This phenomenon is not typical for a resonance curve and should be investigated further by taking more timesteps in between 0.5 Hz and 0.7 Hz.

VI. Conclusion and Future Work

We have been able to build an apparatus of coupled pendula that can display wave motion which will eventually be used to model quantum mechanical phenomena. A comparison between this system and Newton's second law was completed. A discussion of the structure of the script as well as each of the functions within the script was described. Also, an analysis of integrator techniques was completed and a final graph showing the relationship between the amplitude and frequency was provided in the results. Future research opportunities involve a further analysis and simulation of the resonance curve provided in Section 5. Specifically, how the system is not acting in the traditional sense, moreover, why the system decays in amplitude just before resonance is found. Further research can hope to more closely analyze the system and how it relates to quantum mechanical phenomena. Also, finding the wavelength in the system as it relates to frequency as well as the phase difference between the driver and the masses can be determined in future analysis. Hopefully, a paper describing the research conducted with this system can be completed in the upcoming year.

Appendix

```
import numpy as np
import matplotlib.pyplot as plt
#import sys
import h5py

#import pandas as pd

location ="dataFolder/data.csv"

def calc():       #This function serves to run the entire simulation. It contains many of the
                  #arrays we are looking at to model our system. The simulation itself uses
                  #a Runge-Kutta integrator to approximate how the system is acting while
                  #minimizing propagation of error.

    #data = pd.read_csv("dataFolder/data.csv")

    #print(data.x)

    #data.x=[1,2,3]

    #print(data.x)

    #data.to_csv("dataFolder/data.csv")
    # system parameters
    g = 9.81
    k = 17.5
    pi = 3.14159
    v0 = 600
    v0y = v0 * np.sin(pi/180)

    # More parameters
    t0 = 0
    tf = 1000
    h = 0.001

    # steps
    n = 600000
    nmass = 22

    # variables
    t = np.arange(n) * h
    x = np.zeros((n,nmass+2))
    v = np.zeros((n,nmass+2))
    m = np.full(nmass+2,0.995)
    l = np.full(nmass+2,2.20345)
```

```
b = np.empty((n,nmass+2))
#b = np.full((n,nmass+2), 0.1)
k1 = np.zeros(nmass+2)
r1 = np.zeros(nmass+2)
k2 = np.zeros(nmass+2)
r2 = np.zeros(nmass+2)
k3 = np.zeros(nmass+2)
r3 = np.zeros(nmass+2)
k4 = np.zeros(nmass+2)
r4 = np.zeros(nmass+2)

alpha = -g/l


#b decrease over time
for i in range(0,n):
    beta = (-0.099 * (np.exp((0.04 * t[i]) - 10))) / (np.exp((0.04 * t[i]) - 10) + 1) + 0.100


    b[i,:] = beta
print(b)

#sys.exit()
#b.fill(beta)


# initial conditions
x[0,1:nmass-1] = 0



omega0 = 1.4 * np.pi
x[:,0] = 0.1 * np.sin(omega0 * t)
v[:,0] = - 0.1 * omega0 * np.cos(omega0 * t)
print(x[:,0])
exit

v[0,1:nmass-1] = 0



# functions from the 2 coupled DE's
def f1(t , x , v, b, j):
    return v[j]
def f2(t , x , v, b, j):
    return -g / l[j] * x[j] + (k/m[j]) * (x[j + 1] + x[j - 1] - 2 * x[j]) - b[j] * v[j]
```

```
    # Runge Kutta
    for i in range(0,n-1):

        if i % 1000 == 0:
            print(i)
        for j in range (1,nmass+1):
            k1[j] = f1(t[i], x[i,:], v[i,:], b[i,:], j)
            r1[j] = f2(t[i], x[i,:], v[i,:], b[i,:], j)
        for j in range (1,nmass+1):
            k2[j] = f1(t[i] + h/2, x[i,:] + (k1/2) * h, v[i,:] + (r1/2) * h, b[i,:], j)
            r2[j] = f2(t[i] + h/2, x[i,:] + (k1/2) * h, v[i,:] + (r1/2) * h, b[i,:], j)
        for j in range (1,nmass+1):
            k3[j] = f1(t[i] + h/2, x[i,:] + (k2/2) * h, v[i,:] + (r2/2) * h, b[i,:], j)
            r3[j] = f2(t[i] + h/2, x[i,:] + (k2/2) * h, v[i,:] + (r2/2) * h, b[i,:], j)
        for j in range (1,nmass+1):
            k4[j] = f1(t[i] + h, x[i,:] + k3 * h, v[i,:] + r3 * h, b[i,:], j)
            r4[j] = f2(t[i] + h, x[i,:] + k3 * h, v[i,:] + r3 * h, b[i,:], j)

    #iterative solution
        for j in range (1,nmass+1):
            x[i+1,j] = x[i,j] + (k1[j] + 2 * k2[j] + 2 * k3[j] + k4[j]) * h/6
            v[i+1,j] = v[i,j] + (r1[j] + 2 * r2[j] + 2 * r3[j] + r4[j]) * h/6
    return t, x, v, b, m, l

def write(t, x, v, b, m, l, file):    #This function uses the information calculated in def calc
                                      #and writes it to a file. This makes it easier to keep
                                      #track of each of the
                                      #simulations because they are now stored somewhere.
    hf = h5py.File(file, 'w')
    hf.create_dataset('time1', data=t)
    hf.create_dataset('displacement1', data=x)
    hf.create_dataset('velocity1', data=v)
    hf.create_dataset('damping1', data=b)
    hf.create_dataset('mass1', data=m)
    hf.create_dataset('length1', data=l)
    hf.close()
    #t,x,v,b,m,l = calc()
    #T=list(t)
    #X=list(x)


    #dataStored= pd.DataFrame( {'t': T,'x' : X})

    #print(dataStored)
```

```
    #dataStored.to_csv(location)


def read(file):                 #This function now takes the information that was stored in
                                #read and returns it to be used in calculations to be done
                                #with that information.
    hf = h5py.File(file, 'r')

    t1 = hf.get('time1')
    t1=np.array(t1)

    x1 = hf.get('displacement1')
    x1=np.array(x1)

    v1 = hf.get('velocity1')
    v1=np.array(v1)

    b1 = hf.get('damping1')
    b1=np.array(b1)

    m1 = hf.get('mass1')
    m1=np.array(m1)

    l1 = hf.get('length1')
    l1=np.array(l1)

    hf.close()
    return t1,x1,v1,b1,m1,l1

def amp(t, x, ts=400, te=None): #This function finds the maximum amplitude of the
                                #oscillation at any time we want to look at.
    if ts is None:
        ts = t[0]
    if te is None:
        te = t[-1]
    nmass = np.shape(x)[1] - 2
    #amp = np.zeros(nmass)

    m1 = 1
    m2 = nmass + 1
    ind1 = np.where(t == ts)[0][0]
    ind2 = np.where(t == te)[0][0]
    z = np.max(np.abs(x[ind1:ind2,m1:m2]))

    #for i in range (ind1,ind2):
        #amp[i] = np.max(x[i])
```

```
    return z


def plot(t, x, v, j, ts=550, te=None):     #This provides a plot of both the displacement
                                           # versus time as well as the velocity versus time.
                                          #This function also has the ability to look
                                           #at whatever times one would want.
    #for i in range(0,n-1)
    if ts is None:
        ts = t[0]
    if te is None:
        te = t[-1]

    plt.figure()
    plt.plot(t,x[:,0])
    plt.plot(t,x[:,j])
    plt.xlim(ts,te)
    plt.xlabel('Time(s)')
    plt.ylabel('Displacement(m)')
    #plt.ylim(-.1,.1)
    plt.title('Displacement vs Time\nb=0.01, k=17.5, l=2.20345, omega=2pi')

    plt.figure()
    plt.plot(t,v[:,0])
    plt.plot(t,v[:,j])
    plt.xlim(ts,te)
    plt.ylim(-1,1)
    plt.xlabel('Time(s)')
    plt.ylabel('Velocity(m/s)')
    plt.title('Velocity vs Time\nb=0.01, k=17.5, l=2.20345, omega=2pi')


#Energy
def calc1(x,v,m,l):  #This function is able to do an energy calculation for the system.
                     #This keeps track of kinetic and potential energy and returns
                     #their respective values to be used later.

    g = 9.81

    n = np.shape(x)[0]
    nmass = np.shape(x)[1] - 2
    #Arrays
    e = np.zeros(n)

    # Energy Calculation
```

```
    for i in range(0,n):
        u = 0
        ke = 0
        for j in range (1,nmass+1):
            ke += (1/2) * m[j] * v[i,j]**2
            u +=  m[j] * g * l[j] * (1 - np.sqrt(1-(x[i,j]/l[j])**2))
        e[i] = ke + u

    return e


def plot1(t, e):          #This function plots the enegy of the system as a function of time.

    tstart = 0
    tend = 20
    plt.figure()
    plt.plot(t,e)
    plt.plot(t,e)
    plt.xlim(tstart,tend)
    plt.ylim(0,0.05)
```