PROJECT 2

NUMN12 BVP PROJECT

LUND
UNIVERSITY

Jose Mendez-Villanueva

Lund University
NUMN12: Numerical Analysis for Differential Equations
Professor Tony Stillfjord

Brief Theoretical & Methodical Foundations

For this project we will need to know how to use and implement numerical methods for 2-point BVP in which we have Dirichlet conditions. We focus on 2nd order linear cases thus we can avoid theory/explanations for other sorts of problems. The specific numerical method that we decided to implement was Finite Difference, to begin with here is the general form of the problem that we will be working with:

$$y'' = f(x,y), \qquad a < x < b, \quad y(0) = \alpha, \qquad y(b) = \beta$$

Then to begin, we will want to divide our domain $a \le x \le b$ by N intervals which will give us an equally spaced grid with each space being of width $h = \frac{b}{N+1}$. Then we are able to write $x_i = a + xi, \quad i = 0,1, \ldots, N$. Now for our finite difference method we will have our derivatives replaced by our finite difference approximations. We will be using a 2nd Order Central Difference method as we have a 2nd Order ODE. Thus, we denote $y'' = \frac{y_{i+1}-2y_i+y_{i-1}}{h^2}$, which leads to the general form for the equation to be:

$$\frac{y_{i+1}-2y_i+y_{i-1}}{h^2} = f(x_i, y_i), \;\; y_0 = \alpha, \;\; y_{N+1} = \beta$$

The above is for a system of N equations in which we will be solving for $y_1, y_2, \ldots, y_N$ which are our unknowns and we approximate $y(x_1), y(x_2), \ldots, y(x_N)$. Since we are considering the simplest case, we will denote $F(y)$=0 which gives us the following: $F_1(y) = \frac{\alpha - 2y_i + y_2}{h^2} - f(x_1, y_1)$,

$$F_i(y) = \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2} - f(x_i, y_i)$$

$$F_N(y) = \frac{y_{N-1}-2y_N + \beta}{h^2} - f(x_N, y_N)$$

Which will then be reduced to a linear system of equations:

$$\frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & \ddots & \ddots & 0 \\ 0 & \ddots & -2 & 1 \\ 0 & \cdots & 1 & -2 \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} -\frac{\alpha}{h^2} + f(x_1) \\ f(x_2) \\ \vdots \\ -\frac{\beta}{h^2} + f(x_N) \end{bmatrix}$$

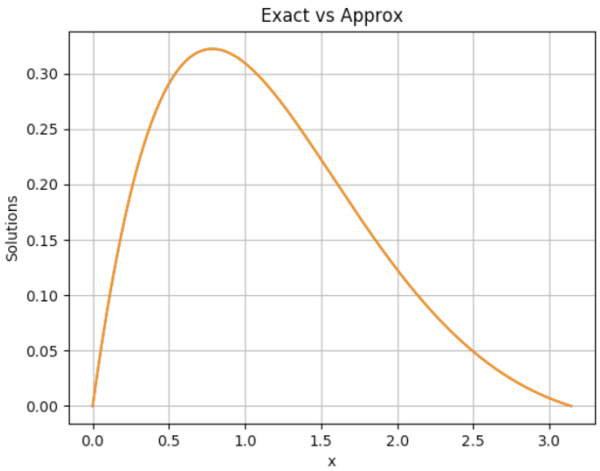
In which we then solve. Since this is the general form for this type of problem, we sometimes have to change the problem according to the system or if a more complex equation is given. For instance, The Beam equation involves using two finite difference matrices because of the fact that there is a system of two, 2nd Order Equations of the form:

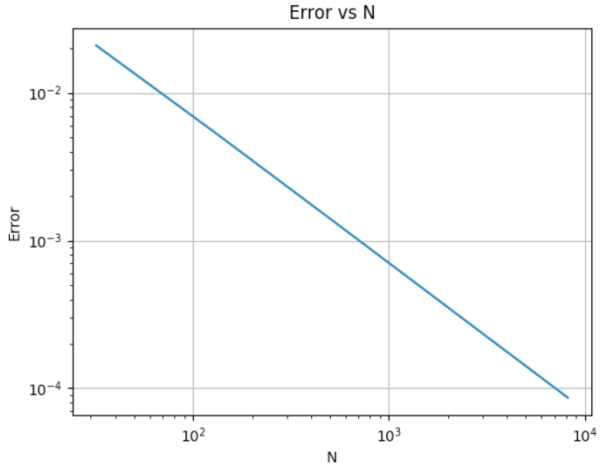
$$M'' = q(x),$$

$$u'' = \frac{M(x)}{(EI)}$$

Then you have to solve the first equation in the manner that is shown above and then solve the second equation as the second equation needs the solutions of the first equation. Following, the general theory for the Schrödinger Equation will be discussed later on.

Task 1

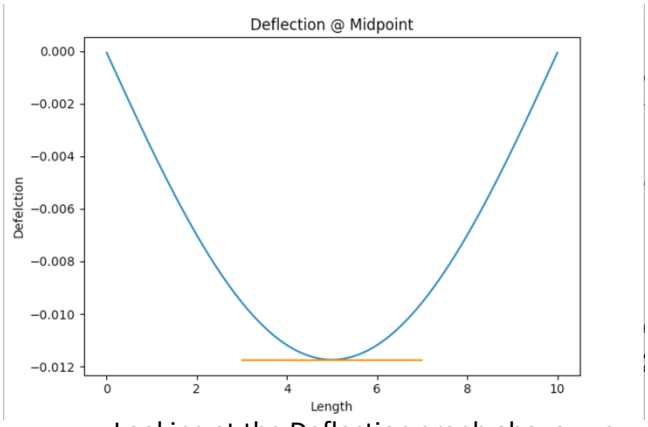




The graph above to the left is depicting the function $f(x) = \frac{2cosx}{e^x}$ and $y(x) = \frac{sinx}{e^x}$ where $f(x)$ is my approximated function and $y(x)$ is the exact function that I compare it to. Here we have the interval of $[0,\pi]$ in wich I use N = 2^{13}. From the graph we can see that using the 2nd Order Finite Difference method gives a fairly accurate result in comparison to the exact function as they are almost one curve. Now for the graph on the right, this graph is Error vs N in which I compare the Error for five different values of N. I use the values $2^5, 2^7, 2^9, 2^{11}, 2^{13}$.

Looking at the left hand picture of my main code. Here we can see how the interval and N give the equally spaced grid H values. Then continue to define B(column matrix) which contains the function that I am trying to approximate and cc which contains the 2nd Order Finite Difference values and then solving for these matrices we are able to get our solutions. Then using different values of N , I am able to compare the Exact and Approximate graphs and see how as N increases the curves become essentially one and we can also see how Error decreases as N increases as well.

```
 9 class FiniteDifference(object):
10     def __init__(self, N,fvec,vexact, alpha, beta):
11         self.b = np.pi
12         self.a = 0
13         self.N = N
14         self.H = (self.b -self.a) / (self.N+2)
15 
16         self.X = np.linspace(self.a, self.b, self.N+2).transpose()
17         self.Solution1 = np.array([])
18         self.Exact  = np.array([])
19         self.Error  = np.array([])
20         self.fvec   = fvec
21         self.vexact = vexact
22         self.alpha = alpha
23         self.beta = beta
24         print(fvec.shape)
25 
26     def Matrix(self):
27         B     = (self.H**2)*self.fvec
28         B[0]  = B[0] - self.alpha
29         B[-1] = B[-1]-self.beta
30         cc    = np.zeros(self.N)
31         cc[0] = -2
32         cc[1] = 1
33         u = la.solve_toeplitz(cc, B)
34         u = np.hstack((self.alpha,u,self.beta))
35 
36         self.Error     = np.sqrt(self.H)*la.norm(u - self.vexact)
37         self.Solution1 = u
38         self.Exact     = self.vexact
39         print(self.Error)
40 
41 
42 def ff(x):
43     return -2*np.cos(x)*np.exp(-x)
44 def exx(x):
45     return np.sin(x)*np.exp(-x)
46 
47 N=2**5
48 N2 = 2**7
49 N3 =2**9
50 N4 = 2**11
51 N5 = 2**13
52 
53 xx = np.linspace(0,np.pi,N+2)
```



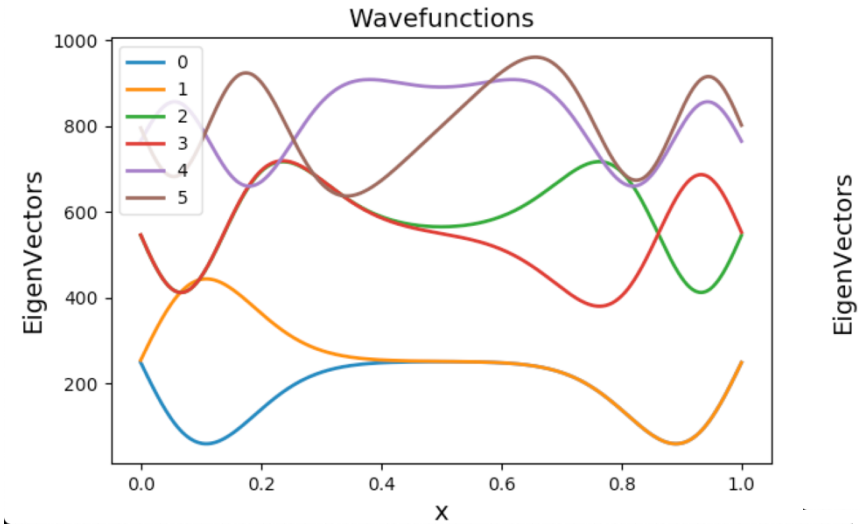

```python
import numpy as np
from scipy import sparse
import matplotlib.pyplot as plt


class FiniteDifference(object):
    def __init__(self):
        self.b = 10
        self.a = 0
        self.N = 999
        self.H = (self.b -self.a) / self.N
        self.X = np.linspace(self.a, self.b, self.N+1).transpose()
        self.Solution1 = np.array([])
        self.Solution2 = np.array([])
        self.E = 1.9e11
    def Matrix(self):
        B = np.zeros((self.N+1, 1)).ravel()
        B[1:self.N] = -50000*(self.H**2 )  #To convert kN/m to N/m: You multiply by 1000 as 1000N = 1kN
        Main = -2*np.ones((self.N + 1, 1)).ravel()
        UpperLower = 1*np.ones((self.N, 1)).ravel()
        UpperLower2 = 1*np.ones((self.N, 1)).ravel()
        A = Main.shape[0]
        DIAGONAL = [Main, UpperLower,UpperLower2]
        BigA = sparse.diags(DIAGONAL, [0,-1,1], shape = (A,A)).toarray()
        BigA[0,0] = -2
        BigA[1,0] = 1
        BigA[self.N,self.N - 1] = 1
        BigA[self.N,self.N - 1] = 1
        self.Solution1 = np.append(self.Solution1, np.linalg.solve(BigA, B))
        C = np.zeros((self.N+1, 1)).ravel()
        C[1:self.N] = (self.Solution1[1:self.N]/(.001*(self.E * (3-2*(np.cos((self.X[1:self.N]*np.pi/self.b))**12)/self.b)))) * (self.H**2 )
        self.Solution2 = np.append(self.Solution2, np.linalg.solve(BigA, C))
        print(min(self.Solution2) )

Testcase = FiniteDifference()
Testcase.Matrix()
plt.figure(1)
plt.plot(Testcase.X,Testcase.Solution2)
plt.plot([3,7], [min(Testcase.Solution2), min(Testcase.Solution2)])
plt.xlabel('Length')
plt.ylabel('Defelction')
plt.title('Deflection @ Midpoint')
plt.show()
#print(Testcase.Solution2)
```

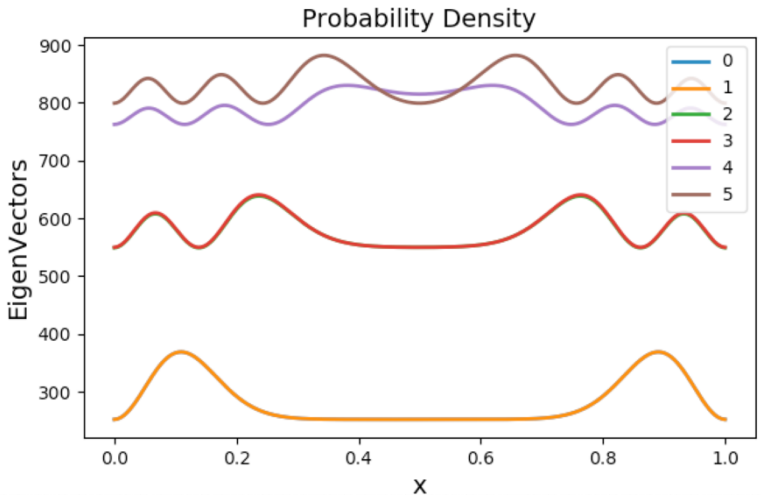
Looking at the Deflection graph above, we see how when the load is put at the midpoint we see a very small defelction as this is expected of a support beam. We expect a small amount of deflection and that is why we see the deflection coming very close to -.012 m exactly at the middle. The computed minimal value that I received from my code was then -0.01174105 m. The original code that I had provided did not give the correct value because I had not raised cosine to the 12^{th} power. Looking at the accuracy of the graph and at the computed value we can see how the approximated values are fairly accurate. Now looking at my code, we use the same basic concept of the last task to implement this but with a slight twist to it. Since we are solving for two equations 2^{nd} Order Differntial Equations, where we need to solve the first equation first as the second equation depends off of the first one. I then first computed the first equation with my B and BigA matrices and then incorporate that solution into my C matrix which contains the 2^{nd} equation that we need. Then after constructing that equation matrix we then use BigA matrix and the C matrix and compute the solution from those two matrices.

Task 2

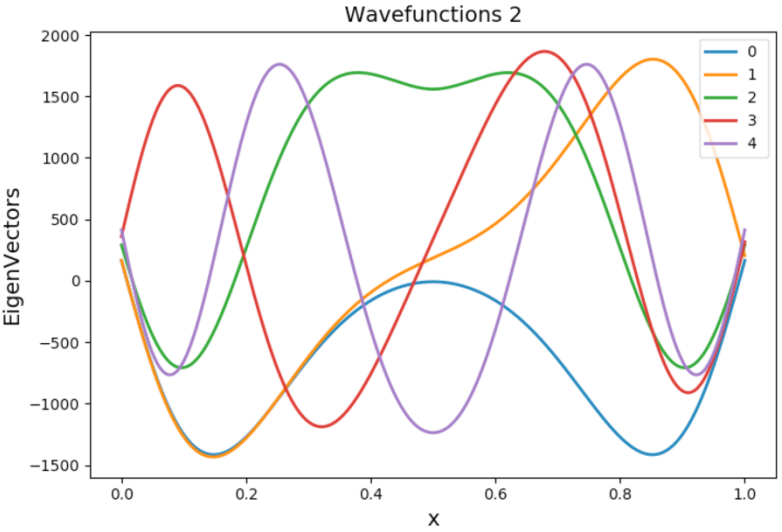
Exact	-9.8696	-39.4784	-88.8264
Approximated	-9.8696	-39.4783	-88.8258

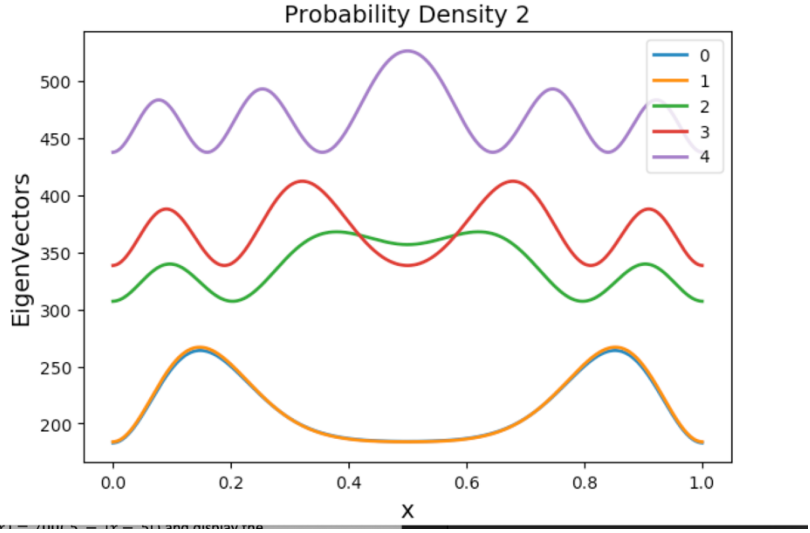
The aprroximated values for the table above are using N=2^{10} and thus we can see how the approximated eigenvalues and exact eigenvalues are very similar to each other. I use N=2^{10} to show the approximated values versus the exact and the graph and approximated values are for the first three values. I then create my BigA matrix which will be the 2^{nd} Order Finite Difference matric (Toeplitz) and solve for the eigenvalues and eigenvectors . Once that is obtained I can then obtain the aprroximations for the different values of N that I set up for and then obtain the error.



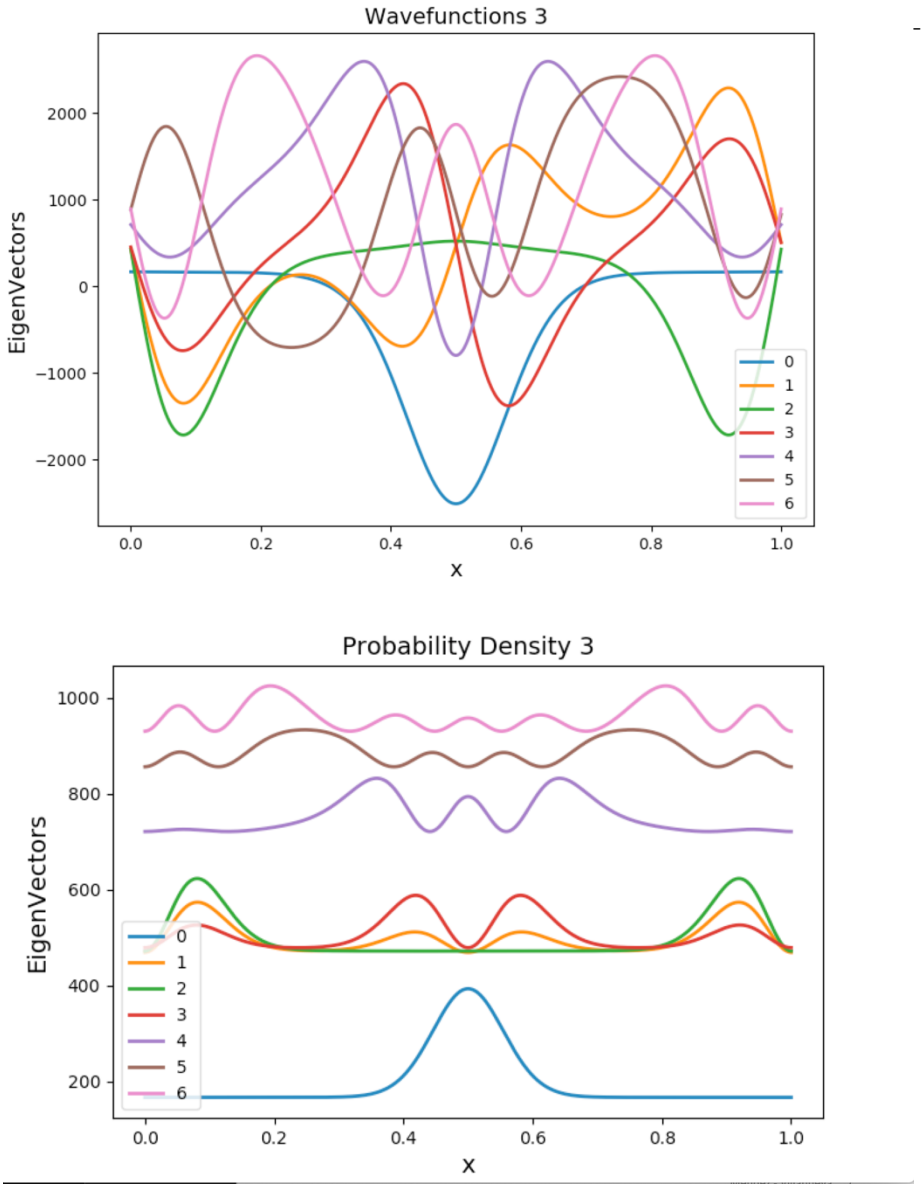


These two graphs are for the Schrodinger equation for potential $V(x) = 800\sin(\pi x)^2$ in the interval [0,1] with boundary conditions $\psi(0) = 0 = \psi(1)$. I deceided to graph the first 6 Wavefunctions and Probability Densities as to show more data for this. Looking at the legend, 0 corresponds to the first eigenvalue and ascends in order to the 5 being the highest eigenvalue for this graph. Both graphs are plotted at the energy levels but different scales for amplitude, hence why one of the graphs has less waves and the other more. For the wave function I graphed it at the energy levels and scaled it so it is also readable. We also see the difference between the graphs as the Probability Density graph will have all positive values as it is $|\psi|^2$, then looking at both we can see how the graphs are accuarte in regards to each other. Now at x=.5 the proability to find the particle becomes significant for eigenvalue 6 from the graph. This is because since there are more oscillations there is more liklihood of a particle being found anywhere in the well.





The above two graphs now correspond to the Potential $V(x) = 700(.5 - |x - .5|)$ and display the same amount of information as the two graphs before. You can notice some differenes in the oscillations bwetween this Potential function and the one from before. These oscialltions seem to be more evenly spread out compared to the other ones. If the oscillations are rapid and occur more often that implies that there is more probability for finding a particle anywhere in the well. Then when comparing both of the probability densities you can see that for values 4-6, that the graph for Probability Density 2 has more oscilations that are equally spaced out.



Now the above graph includes a triple state, the above graph below will show how there should be two local maxima in which Eigenvalue 4 demenstrate this the best. The potential function that I used to obtain this was $V(x) = 800\sin(2\pi x)^2$. If you multiply the inside by higher numbers you start to get interesting looking graphs as well.

```python
import numpy as np
from scipy import sparse
import matplotlib.pyplot as plt

class FiniteDifference(object):
    def __init__(self):
        self.b = 1
        self.a = 0
        self.N = 1000
        self.H = (self.b -self.a) / self.N
        self.X = np.linspace(self.a, self.b, self.N+1).transpose()
        self.Solution = np.array([])
    def Matrix(self):
        Main = -2*np.ones((self.N + 1 , 1)).ravel()
        UpperLower = 1*np.ones((self.N, 1)).ravel()
        UpperLower2 = 1*np.ones((self.N, 1)).ravel()
        A = Main.shape[0]
        DIAGONAL = [Main, UpperLower,UpperLower2]
        BigA = sparse.diags(DIAGONAL, [0,-1,1], shape = (A,A)).toarray()
        BigA[0,0] = -2
        BigA[1,0] = 1
        BigA[self.N,self.N - 1] = 1
        BigA[self.N,self.N - 1] = 1

        VFunction = np.diagflat((800*(np.sin(self.X*np.pi)**2)))
        HamMatrix = ((-BigA)/(self.H**2)) + VFunction
        evalue, evector = np.linalg.eig(HamMatrix )

        z = np.argsort(evalue)
        z = z[0:6]
        Energies=(evalue[z])

        plt.figure(1, figsize=(12,10))
        for i in range(len(z)):
            plt.plot(self.X, ( evector[:, z[i]] /  np.sqrt(self.H) * 100 ) + Energies[i]  , lw=2, label="{} ".format(i))
            plt.xlabel('x', size=14)
            plt.ylabel('EigenVectors',size=14)
        plt.legend()
        plt.title('Wavefunctions',size=14)
        plt.show()

        plt.figure(2, figsize=(12,10))
        for i in range(len(z)):
            plt.plot(self.X, (((abs(evector[:, z[i]] * evector[:, z[i]])  / np.sqrt(self.H))) * 1000) + Energies[i]   , lw=2, label="{} ".format(i))
            plt.xlabel('x', size=14)
            plt.ylabel('EigenVectors',size=14)
        plt.legend()
        plt.title('Probability Density',size=14)
        plt.show()

        VFunction2 = np.diagflat((700*(.5 - abs(self.X - .5))))
        HamMatrix2 = ((-BigA)/(self.H**2)) + VFunction2
        evalue2, evector2 = np.linalg.eig(HamMatrix2 )

        z2 = np.argsort(evalue2)
        z2 = z2[0:5]
        Energies2=(evalue2[z2])

        plt.figure(3, figsize=(12,10))
        for i in range(len(z2)):
            plt.plot(self.X, ((evector2[:, z2[i]]  / np.sqrt(self.H))*1000 ) + Energies2[i]  , lw=2, label="{} ".format(i))
            plt.xlabel('x', size=14)
            plt.ylabel('EigenVectors',size=14)
```

Looking at my code, it has the same fundamentals as all the other codes before but since we have a more complicated equation it changes some of the things that we need to do. Since the equation is of the form: $\psi'' - V(x)\psi = -E\psi$, we then have two matrices and 3 column vectors. We have the 2nd Order Finite Difference matrix for ψ'' and then we have a diagonal matrix for $V(x)$ with the diagonal consisting of $V(x_i)$ which is the Potential Function, and then our column matrices are ψ_i that take the place of the ψ.

$$\frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & \ddots & \ddots & 0 \\ 0 & \ddots & -2 & 1 \\ 0 & \cdots & 1 & -2 \end{bmatrix}\begin{bmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_N \end{bmatrix} - \begin{bmatrix} V_1 & 0 & 0 & 0 \\ 0 & V_2 & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & 0 & 0 & V_N \end{bmatrix}\begin{bmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_N \end{bmatrix} = -E\begin{bmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_N \end{bmatrix}$$

Then we combine our 2nd Order Finite DIfference matrix with our $V(x)$ matrix which gives us our Hamiltonian matrix

$$Hamiltonian = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & \ddots & \ddots & 0 \\ 0 & \ddots & -2 & 1 \\ 0 & \cdots & 1 & -2 \end{bmatrix} - \begin{bmatrix} V_1 & 0 & 0 & 0 \\ 0 & V_2 & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & 0 & 0 & V_N \end{bmatrix}$$

and then solve for the Eigenvalues and Eigenvectors .Then ordering the eigenvalues from lowest eigenvalues to highest eigenvalues by setting our z variable and then calling it into the eigenvector gives the plot needed in order from lowest to highest. This code includes the main portion of code , not including the code for the second graphs of wavefunctions and probability density which is included in the CODE SECTION of my report after references.

Problems

Some of the problems that I faced were implementing my loglog graph in a normal readable way in python, thus that is why I needed help on this. Other than that, another issue that I faced was being able to find a Potential function that includes a triple state. It took some testing but after reading the comment I believe that I was able to include it. Coding in general in python caused me much difficulty and need to work on coding better in python.

References

Burden, Richard L., et al. Numerical Analysis. Cengage Learning, 2016.

"Finite Difference Method." *Wikipedia*, Wikimedia Foundation, 8 Nov. 2019, en.wikipedia.org/wiki/Finite_difference_method.

Code

Task 1.1:

```
import math
import numpy as np
from scipy import sparse
import matplotlib.pyplot as plt


class FiniteDifference(object):
    def __init__(self, N):
        self.b = 10                                          #Final time
        self.a = 0                                           #Start time
        self.N = N                                          #Number of steps wanted to create the equally spaced grid
        self.H = (self.b -self.a) / self.N
        self.X = np.linspace(self.a, self.b, self.N+1).transpose()  #Transpose to get in correct format for when it runs code
        self.Solution1 = np.array([])
        self.Exact = np.array([])
        self.Error = np.array([])

    def Matrix(self):
        B = np.zeros((self.N + 1, 1)).ravel()
        B[1:self.N] = (2*np.cos(self.X[1:self.N])/(np.e**(self.X[1:self.N]))) * (self.H**2 )
        Main = -2*np.ones((self.N + 1, 1)).ravel()
        UpperLower = 1*np.ones((self.N, 1)).ravel()
        UpperLower2 = 1*np.ones((self.N, 1)).ravel()
        A = Main.shape[0]
        DIAGONAL = [Main, UpperLower,UpperLower2]
        BigA = sparse.diags(DIAGONAL, [0,-1,1], shape = (A,A)).toarray()
        BigA[0,0] = -2
        BigA[1,0] = 1
        BigA[self.N,self.N - 1] = 1
        BigA[self.N,self.N - 1] = 1
        self.Solution1 = np.append(self.Solution1, np.linalg.solve(-BigA, B))
        self.Exact = np.append(self.Exact, np.sin(self.X) / np.e**(self.X))
        self.Error = np.append(self.Error, abs(self.Exact - self.Solution1) )
        print(self.Error)

Testcase = FiniteDifference(2**4)
Testcase.Matrix()
Testcase4 = FiniteDifference(2**7)
Testcase4.Matrix()
Testcase6 = FiniteDifference(2**9)
Testcase6.Matrix()

x = [1/2**4]*17
xx = [1/2**7]*129
xxx = [1/2**9]*513

plt.figure(1)
plt.plot(Testcase6.X, Testcase6.Exact, label=r"$Exact$")
plt.plot(Testcase6.X, Testcase6.Solution1, label=r"$Approximation$")
plt.legend(loc='upper right')
plt.xlabel('Time')
plt.ylabel('Solution Values')
plt.title('Error vs Exact')
plt.show()

plt.figure(2)
plt.loglog(x,Testcase.Error,'.', lw=.1, label=r"$2^4$")
plt.loglog(xx,Testcase4.Error, '.', lw=.1, label=r"$2^7$")
plt.loglog(xxx,Testcase6.Error, '.', lw=.1, label=r"$2^9$")
plt.legend(loc='lower right')
plt.xlabel('N')
plt.ylabel('Error')
plt.title('Error vs N')
plt.show()
```

Task 1.2:

```

import numpy as np
from scipy import sparse
import matplotlib.pyplot as plt


class FiniteDifference(object):
    def __init__(self):
        self.b = 10                                          #Final time
        self.a = 0                                          #Start time
        self.N = 999                                         #Number of steps wanted to create the equally spaced grid
        self.H = (self.b -self.a) / self.N
        self.X = np.linspace(self.a, self.b, self.N+1).transpose()  #Transpose to get in correct format for when it runs code
        self.Solution1 = np.array([])
        self.Solution2 = np.array([])
        self.E = 1.9e11
    def Matrix(self):
        B = np.zeros((self.N+1, 1)).ravel()
        B[1:self.N] = -50000*(self.H**2 )  #To convert kN/m to N/m: You multiply by 1000 as 1000N = 1kN
        Main = -2*np.ones((self.N + 1, 1)).ravel()
        UpperLower = 1*np.ones((self.N, 1)).ravel()
        UpperLower2 = 1*np.ones((self.N, 1)).ravel()
        A = Main.shape[0]
        DIAGONAL = [Main, UpperLower,UpperLower2]
        BigA = sparse.diags(DIAGONAL, [0,-1,1], shape = (A,A)).toarray()
        BigA[0,0] = 1
        BigA[0,1] = 0
        BigA[self.N,self.N] = 1
        BigA[self.N,self.N - 1] = 0
        self.Solution1 = np.append(self.Solution1, np.linalg.solve(BigA, B))
        C = np.zeros((self.N+1, 1)).ravel()
        C[1:self.N] = self.Solution1[1:self.N]/(self.E * (3-2*np.cos((self.X[1:self.N]*np.pi))/self.b)) * (self.H**2 )
        self.Solution2 = np.append(self.Solution2, np.linalg.solve(BigA, C))
        print(min(self.Solution2))

Testcase = FiniteDifference()
Testcase.Matrix()
plt.figure(1)
plt.plot(Testcase.X,Testcase.Solution2)
plt.plot([3,7], [min(Testcase.Solution2), min(Testcase.Solution2)])
plt.xlabel('Length')
plt.ylabel('Defelction')
plt.title('Deflection @ Midpoint')
plt.show()
```

Task 2.1:

```python
import numpy as np
from scipy import sparse
import matplotlib.pyplot as plt

class FiniteDifference(object):
    def __init__(self,N):
        self.b = 1                                          #Final time
        self.a = 0                                          #Start time
        self.N = N                                         #Number of steps wanted to create the equally spaced grid
        self.H = (self.b -self.a) / self.N
        self.X = np.linspace(self.a, self.b, self.N+1).transpose()
        self.Solution = np.array([])
        self.Exact = np.array([])
        self.Error = np.array([])
        self.SORT = np.array([])
        self.deltaX = np.array([])
    def Matrix(self):
        Main = -2*np.ones((self.N + 1, 1)).ravel()
        UpperLower = 1*np.ones((self.N, 1)).ravel()
        UpperLower2 = 1*np.ones((self.N, 1)).ravel()
        A = Main.shape[0]
        DIAGONAL = [Main, UpperLower,UpperLower2]
        BigA = sparse.diags(DIAGONAL, [0,-1,1], shape = (A,A)).toarray()
        BigA[0,0] = -2
        BigA[1,0] = 1
        BigA[self.N,self.N-1] = 1
        BigA[self.N,self.N - 1] = 1
        evalue, evector = np.linalg.eig(BigA/self.H **2)
        Sortedevalue = sorted(evalue)
        self.SORT = np.append(self.SORT, np.flip(Sortedevalue, 0))
        self.SORT = np.insert(self.SORT, 0, 0)

        for i in range(4):
            self.Exact = np.append(self.Exact, ((i**2)*(np.pi**2)))
            self.Error = np.append(self.Error, self.SORT[i] + self.Exact[i])

        print(self.Exact[1:4])
        print(self.SORT[1:4])
        print(self.Error[1:4])




Testcase = FiniteDifference(2**4)
Testcase.Matrix()
Testcase2 = FiniteDifference(2**5)
Testcase2.Matrix()
Testcase3 = FiniteDifference(2**6)
Testcase3.Matrix()
Testcase4 = FiniteDifference(2**7)
Testcase4.Matrix()
Testcase5 = FiniteDifference(2**8)
Testcase5.Matrix()
Testcase6 = FiniteDifference(2**9)
Testcase6.Matrix()
Testcase7 = FiniteDifference(2**10)
Testcase7.Matrix()


plt.loglog([1/2**4, 1/2**4, 1/2**4],Testcase.Error[1:4], '.', label=r"$2^4$")
plt.loglog([1/2**5, 1/2**5, 1/2**5],Testcase2.Error[1:4], '.', label=r"$2^5$")
plt.loglog([1/2**6, 1/2**6, 1/2**6],Testcase3.Error[1:4], '.', label=r"$2^6$")
plt.loglog([1/2**7, 1/2**7, 1/2**7],Testcase4.Error[1:4], '.', label=r"$2^7$")
plt.loglog([1/2**8, 1/2**8, 1/2**8],Testcase5.Error[1:4], '.', label=r"$2^8$")
plt.loglog([1/2**9, 1/2**9, 1/2**9],Testcase6.Error[1:4], '.', label=r"$2^9$")
plt.loglog([1/2**10, 1/2**10, 1/2**10],Testcase7.Error[1:4], '.', label=r"$2^(10)$")
plt.xlabel('N')
plt.legend(loc='lower right')
plt.ylabel('Error')
plt.title('Error vs N')
plt.show()
```

Task 2.2:

```
import numpy as np
from scipy import sparse
import matplotlib.pyplot as plt

class FiniteDifference(object):
    def __init__(self):
        self.b = 1
        self.a = 0
        self.N = 1000
        self.H = (self.b -self.a) / self.N
        self.X = np.linspace(self.a, self.b, self.N+1).transpose()
        self.Solution = np.array([])
    def Matrix(self):
        Main = -2*np.ones((self.N + 1 , 1)).ravel()
        UpperLower = 1*np.ones((self.N, 1)).ravel()
        UpperLower2 = 1*np.ones((self.N, 1)).ravel()
        A = Main.shape[0]
        DIAGONAL = [Main, UpperLower,UpperLower2]
        BigA = sparse.diags(DIAGONAL, [0,-1,1], shape = (A,A)).toarray()
        BigA[0,0] = -2
        BigA[1,0] = 1
        BigA[self.N,self.N - 1] = 1
        BigA[self.N,self.N - 1] = 1


        VFunction = np.diagflat((800*(np.sin(np.pi*self.X ))**2))
        HamMatrix = ((-BigA)/(self.H**2)) + VFunction
        evalue, evector = np.linalg.eig(HamMatrix )

        z = np.argsort(evalue)
        z = z[0:7]

        Energies=(evalue[z]/evalue[z][0])

        plt.figure(1, figsize=(12,10))
        for i in range(len(z)):
            plt.plot(self.X, evector[:, z[i]]  / np.sqrt(self.H) + Energies[i] * 3, lw=2, label="{} ".format(i))
            plt.xlabel('x', size=14)
            plt.ylabel('EigenVectors',size=14)
        plt.legend()
        plt.title('Wavefunctions',size=14)
        plt.show()

        plt.figure(2, figsize=(12,10))
        for i in range(len(z)):
            plt.plot(self.X, (evector[:, z[i]] * evector[:, z[i]])  / np.sqrt(self.H) + Energies[i] / 5, lw=2, label="{} ".format(i))
            plt.xlabel('x', size=14)
            plt.ylabel('EigenVectors',size=14)
        plt.legend()
        plt.title('Probability Density',size=14)
        plt.show()


        VFunction2 = np.diagflat((700*(.5 - abs(self.X - .5))))
        HamMatrix2 = ((-BigA)/(self.H**2)) + VFunction2
        evalue2, evector2 = np.linalg.eig(HamMatrix2 )

        z2 = np.argsort(evalue2)
        z2 = z2[0:7]


        plt.figure(3, figsize=(12,10))
        for i in range(len(z2)):
            plt.plot(self.X, evector2[:, z2[i]]  / np.sqrt(self.H) , lw=2, label="{} ".format(i))
            plt.xlabel('x', size=14)
            plt.ylabel('EigenVectors',size=14)
        plt.legend()
        plt.title('Wavefunctions 2',size=14)
        plt.show()

        plt.figure(4, figsize=(12,10))
        for i in range(len(z2)):
            plt.plot(self.X, (evector2[:, z2[i]] * evector2[:, z2[i]])  / np.sqrt(self.H), lw=2, label="{} ".format(i))
            plt.xlabel('x', size=14)
            plt.ylabel('EigenVectors',size=14)
        plt.legend()
        plt.title('Probability Density 2',size=14)
        plt.show()


Testcase = FiniteDifference()
Testcase.Matrix()
```