DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION

UNIVERSITY OF MORATUWA

EN 3110: ELECTRONIC DEVICES

NUMERICAL SOLUTIONS FOR ONE-DIMENSIONAL TIME-INDEPENDENT SCHRÖDINGER'S WAVE EQUATION FOR A USER-DEFINED POTENTIAL ENERGY FUNCTION USING PYTHON

K.G. Abeywardena

160005C

31st of January 2020

CONTENT

TABLE OF FIGURES

1. INTRODUCTION

Quantum Physics, unlike the Classical Physics, describes the behavior of the matter and light (photons) on the atomic and subatomic level. It is heavily based on the mathematical models which attempt to describe and account for the properties of atoms and their constituents. Schrödinger Equation, also known as *Schrödinger's Wave Equation*, is a partial differential equation that describes the dynamics of a quantum mechanics system using the wave function.

The *Time-Dependent Schrödinger Equation* is as follows:

$$\frac{-\hbar^2}{2m}\nabla^2\Psi(\vec{r},t) + V(\vec{r})\Psi(\vec{r},t) = i\hbar\frac{\partial\Psi(\vec{r},t)}{\partial t}$$

Where,

$\hbar - reduced\ Planck\ constant\ (1.0546 \times 10^{-34} Js)$

$m - Effective\ mass\ of\ the\ quantum\ particle$

$\Psi(\vec{r},t) - Time\ Dependent\ wave\ function$

$\vec{r} - Position\ Vector\ of\ the\ particle$

$V(\vec{r}) - Potenstial\ Energy\ Function$

Since we are more interested in stationary quantum states rather than how they evolve with time, we usually consider the *Time-Independent* version of the Schrödinger's wave equation as below:

$$\frac{-\hbar^2}{2m}\nabla^2\Psi(\vec{r}) + V(\vec{r})\Psi(\vec{r}) = E\Psi(\vec{r})$$

Where,

$\Psi(\vec{r}) - Time\ Independent\ wave\ function$

$E - Total\ energy\ of\ the\ particle$

According to the *Heisenberg's Uncertainty Principle*, we cannot predict the exact position and momentum of a particle at the same time in quantum world. We can only predict a probability for the particle being at a specific location $\vec{r}$. By solving the Schrödinger equation, we can get the *Probability Density Function (PDF)* for a particle being at $\vec{r}$ as follows.

$$p(\vec{r}) = |\Psi(\vec{r})|^2 = \Psi(\vec{r}) \times \Psi(\vec{r})^*$$

In this report, we explore a method to numerically solve *1-Dimensional* wave equation for a user specified 1-Dimensional Potential Energy Function $V(x)$ using Python. The solution will calculate the user defined number of possible eigen values for Energy (E) along with the solutions for wave function, $\Psi(x)$, and corresponding Probability density Function $|\Psi(x)|^2$

2. METHOD

In this method, we first represent the *Hamiltonian Operator* in matrix form by converting the *Laplacian Operator* and the *Potential Energy Function* to matrices. We can rearrange the 1-Dimensional Schrödinger's wave equation as follows.

$$\underbrace{\left(\frac{-\hbar}{2m}\frac{d^2}{dx^2}+V(x)\right)}_{Hamiltonian\ Operator\ (H)}\quad \Psi(x)=E\Psi(x)$$

We then represent this as a *vector equation* as follows which yields to solving an *Eigen Vector Equation*.

$$H.|\psi\rangle = E.|\psi\rangle$$

Where,

$H - Hamiltonian\ Matrix\ (N \times N)$

$|\psi\rangle - Vectorized\ wave\ function\ (N \times 1)$

$E - Eigen\ Energy\ value$

I used the $eigh()$ function of the $Scipy\ library$ in Python to find the *Eigen Values* (E) and the corresponding *Eigen Vectors* $(|\psi\rangle)$ by solving $det(H - E.I) = 0$.

The range of *x co-ordinates* ($|x| \leq\ L$) and the number of discrete samples (N) within the range of x co-ordinates are to be specified by the user when executing the program.

2.1. Representing Laplacian Operator in Matrix form

Laplacian Operator for 1-Dimention is given by ${d^2}/{dx^2}$ which can be then approximated using the three-point finite difference method.

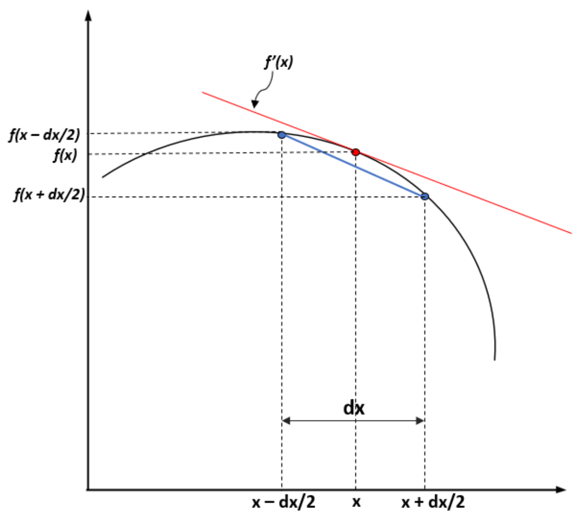

Figure 2.1: Approximating Derivative

Using the *Central Difference Equation*, we can approximate the first derivative of a function as,

$$f'(x) = \frac{f\left(x + \frac{dx}{2}\right) - f\left(x - \frac{dx}{2}\right)}{dx} + O(dx)$$

Where $O(dx) - the\ error\ of\ approximation$

Hence using the same principle, we can approximate the 1-Dimensioanl Laplacian as follows.

$$\left(\frac{d^2}{dx^2}\right) f(x) = \frac{f'\left(x+\frac{dx}{2}\right) - f'\left(x-\frac{dx}{2}\right)}{dx} + O(dx^2)$$

Using the two formulas,

$$\left(\frac{d^2}{dx^2}\right) f(x) = \frac{\left[\frac{f\left(x+\frac{dx}{2}+\frac{dx}{2}\right) - f\left(x+\frac{dx}{2}-\frac{dx}{2}\right)}{dx}\right] - \left[\frac{f\left(x-\frac{dx}{2}+\frac{dx}{2}\right) - f\left(x-\frac{dx}{2}-\frac{dx}{2}\right)}{dx}\right]}{dx} + O(dx^2)$$

$$\left(\frac{d^2}{dx^2}\right) f(x) = \frac{\left[\frac{f(x+dx) - f(x)}{dx}\right] - \left[\frac{f(x) - f(x-dx)}{dx}\right]}{dx} + O(dx^2)$$

$$\left(\frac{d^2}{dx^2}\right) f(x) = \frac{f(x+dx) + f(x-dx) - 2f(x)}{(dx)^2} + O(dx^2)$$

Assuming that dx is small enough such that the error $O(dx^2)$ approaches to 0, we can approximate the 1-Dimensioanl Laplacian operator as follows.

$$\left(\frac{d^2}{dx^2}\right) f(x) \approx \frac{f(x+dx) + f(x-dx) - 2f(x)}{(dx)^2}$$

I defined $dx = \frac{2L}{N}$ where L, N are defined as before and $|x| \leq L$

Now we can write the *Laplacian Equations* in the Schrödinger's Wave Equation as follows:

$$\Psi''(-L) = \frac{1}{(dx)^2}\{\Psi(-[L-dx]) + \Psi(-[L+dx]) - 2\Psi(-L)\}$$
$$\Psi''(-[L-dx]) = \frac{1}{(dx)^2}\{\Psi(-[L-2dx]) + \Psi(-L) - 2\Psi(-[L-dx])\}$$
$$\vdots$$
$$\vdots$$
$$\Psi''(L-dx) = \frac{1}{(dx)^2}\{\Psi(L) + \Psi([L-2dx]) - 2\Psi(L-dx)\}$$
$$\Psi''(L) = \frac{1}{(dx)^2}\{\Psi(L+dx) + \Psi(L-dx) - 2\Psi(L)\}$$

$$\underbrace{\begin{bmatrix} \Psi''(-L) \\ \Psi''(-[L-dx]) \\ \vdots \\ \vdots \\ \vdots \\ \Psi''(L-dx) \\ \Psi''(L) \end{bmatrix}_{N\times 1}}_{\nabla^2|\psi\rangle} = \underbrace{\frac{1}{(dx)^2}\begin{bmatrix} 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 & 0 \\ \vdots & & \ddots & & & & \\ \vdots & & & \ddots & & & \\ \vdots & & & & \ddots & & \\ 0 & & & & & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & -2 & 1 \end{bmatrix}_{N\times N+2}}_{Laplacian\ Matrix} \underbrace{\begin{bmatrix} \Psi(-[L+dx]) \\ \Psi(-L) \\ \Psi(-[L-dx]) \\ \vdots \\ \vdots \\ \Psi(L) \\ \Psi(L+dx) \end{bmatrix}_{N+2\times 1}}_{|\psi\rangle}$$

In this case, Dirichlet boundary conditions with $\Psi(-[L+dx]) = \Psi(L+dx) = 0$ will be used as this will ensure the integral of the *PDF* over the whole x-dimension becomes finite. (So that the wave functions can be normalized such that the *Total Probability* becomes 1)

Then,

$$\nabla^2 = \frac{1}{(dx)^2}\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & -2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix}_{N\times N} \quad , \quad |\psi\rangle = \begin{bmatrix} \Psi(-L) \\ \Psi(-[L-dx]) \\ \vdots \\ \vdots \\ \Psi(L) \end{bmatrix}_{N\times 1}$$

$$giving\ \ \nabla^2|\psi\rangle \approx \left(\frac{d^2}{dx^2}\right)\Psi(x)$$

We can see that the Laplacian matrix is a symmetric matrix which can be solved using the $eigh()$ function and decrease the memory needed as storing the upper triangle of the matrix is sufficient for the recreation of the matrix when calculating the solutions.

2.2. Representing Potential Energy Function as a Matrix

The *Potential Energy Function* $V(x)$ is a user defined input into the system. This is stated as a vector of N elements and it should be converted to $N \times N$ square matrix with diagonal being occupied by the N elements input to the system. All the other elements become 0, making this a *Diagonal Matrix.*

Below shows the method of how the potential energy function is represented as a matrix.

$$|V\rangle = \begin{bmatrix} V(-L) \\ V(-[L-dx]) \\ \vdots \\ \vdots \\ V(L) \end{bmatrix}_{N\times 1} \xRightarrow{Diagonalization} V = \begin{bmatrix} V(-L) & 0 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & V(0) & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 0 & V(L) \end{bmatrix}_{N\times N}$$

$$giving \quad V.|\psi\rangle \approx V(x)\,\Psi(x)$$

This can be easily achieved using the $numpy.diag()$, a built-in function in Python, once the Potential Energy Function is given.

2.3. Hamiltonian Matrix and Solution

Now that we have represented the Laplacian and Potential Energy Function as matrices, it is time to build the Hamiltonian matrix.

$$H = \frac{-\hbar^2}{2m} \times \nabla^2 + V$$

$$H = \frac{-\hbar^2}{2m(dx)^2}\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & -2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix}_{N\times N} + \begin{bmatrix} V(-L) & 0 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & V(0) & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 0 & V(L) \end{bmatrix}_{N\times N}$$

For the computation, I approximated $\frac{\hbar^2}{m} = 1$ as the values for $\hbar, m$ are of the order of $\sim 10^{-30}$ that leads to instability on the solution due to the small numbers.

Since the Hamiltonian is a symmetric matrix and that we have a eigen vector equation, we can use $eigh()$ function of the $Scipy\ library$ to find the *eigen values* and the corresponding *eigen vectors*. The *Eigen Values* corresponds to the *Eigen Energy Values* while the *Eigen Vectors* correspond to the *solutions for wave functions*. Resulting wave functions and the PDFs can be normalized as follows:

Let,

$$K = \sum_i \left| |\psi\rangle^{(i)} \right|^2 dx_i = dx \sum_i \left| |\psi\rangle^{(i)} \right|^2$$

$$Normalized\ wave\ vector: |\bar{\psi}\rangle = {|\psi\rangle}/{\sqrt{K}}$$

$$Normalized\ PDF: \overline{|f\rangle} = {|f\rangle}/{K}$$

Where $|f\rangle = \left||\psi\rangle\right|^2$ is the PDF which is not normalized

After the solutions are obtained, I plotted the wave functions and the PDFs with the corresponding Eigen Energy Values for *5 eigen states*.

3. Results

The wave functions and PDFs are plotted for several 1-Dimensional Potential energy functions considering 5 Eigen Energy values.

3.1. Free – Particle

Free – particle is a quantum particle that is *not subjected* to any changes in the potential energy as there are no interactions with other particles. For a Free Particle, the potential energy is usually a *constant*. In this report, I consider the potential energy of the particle as 0eV (Meaning the *Total Energy* = *Kinetic Energy*)

Based on the *probability distribution function*, we see that, the probability of finding a free particle anywhere in the space along x-axis is uniform.

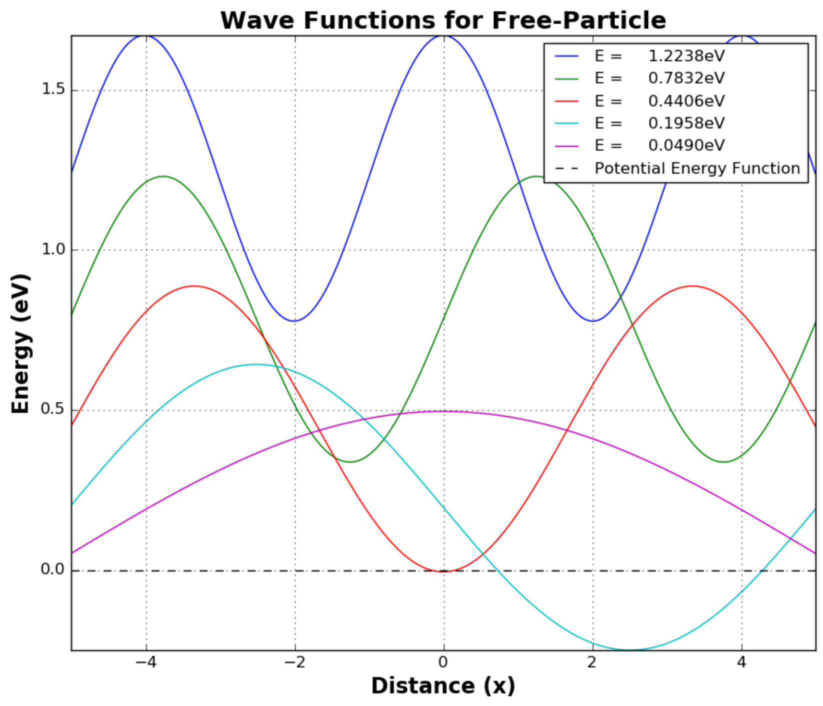

Figure 3.1: Wave Functions for Free-Particle

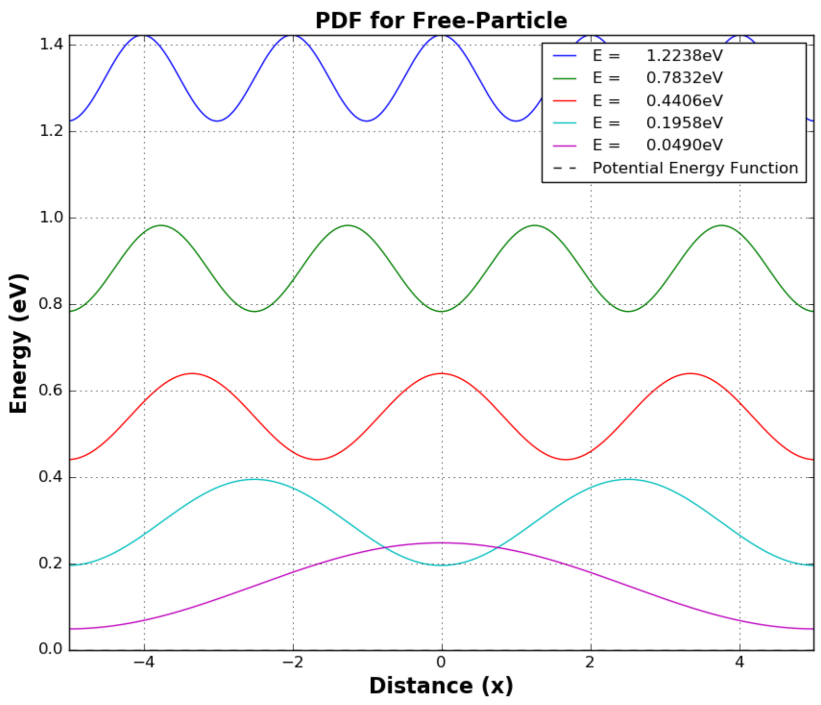

Figure 3.2: Probability Density Function for Free-Particle

3.2. Near Infinite Potential Well

In this scenario, I modeled an Infinite Potential Well by making an energy barrier of 2000eV.

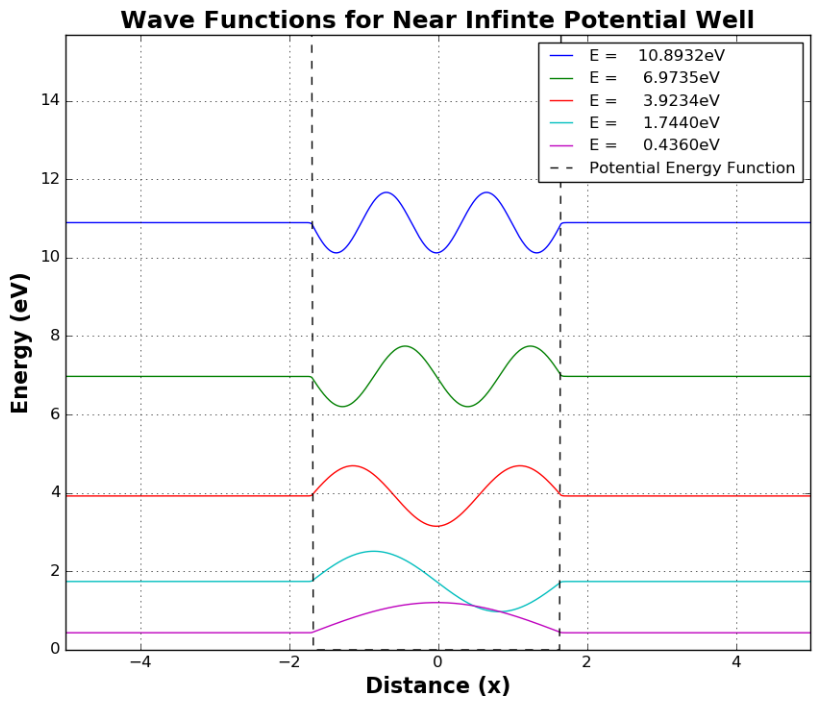

Figure 3.3: Wave Functions for a Particle in a Infinite Potential Well

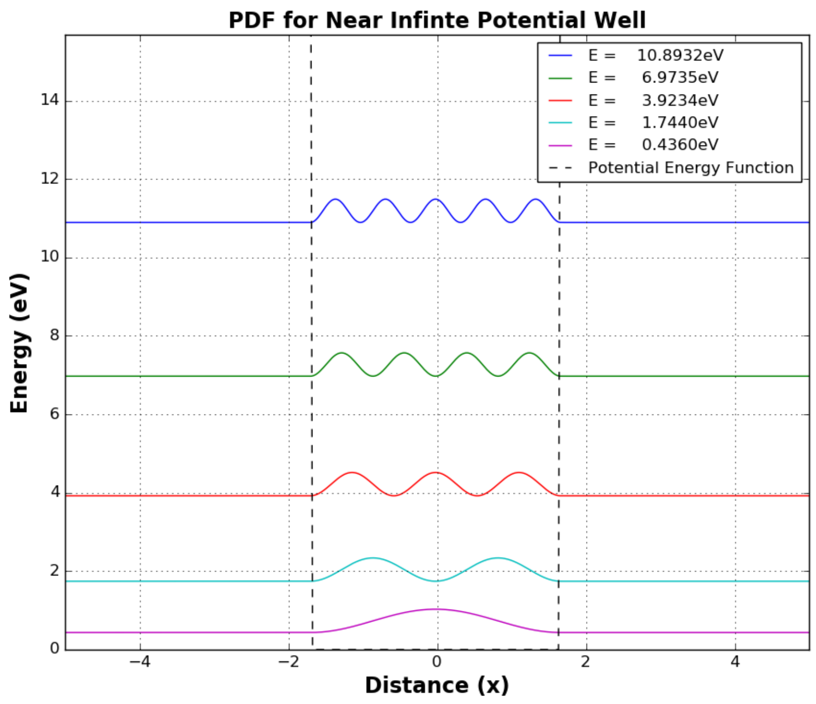

Figure 3.4: Probability Density Function for a Particle in Infinite Potential Well

3.3. Finite Potential Well

Here I modeled a *Finite Potential Well* by making the potential barrier to 10eV.

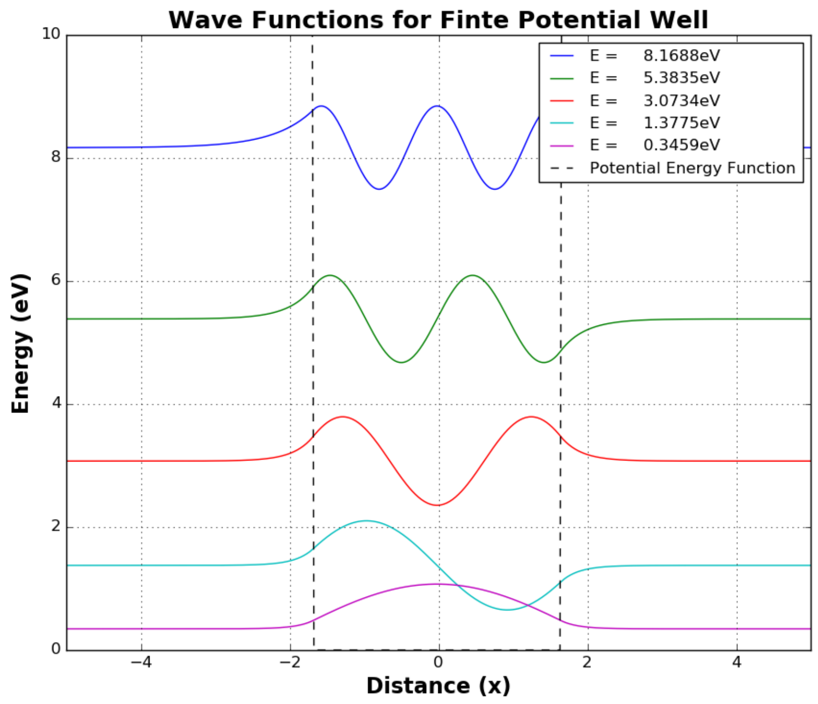

Figure 3.5: Wave Functions for a Particle in a Finite Potential Well

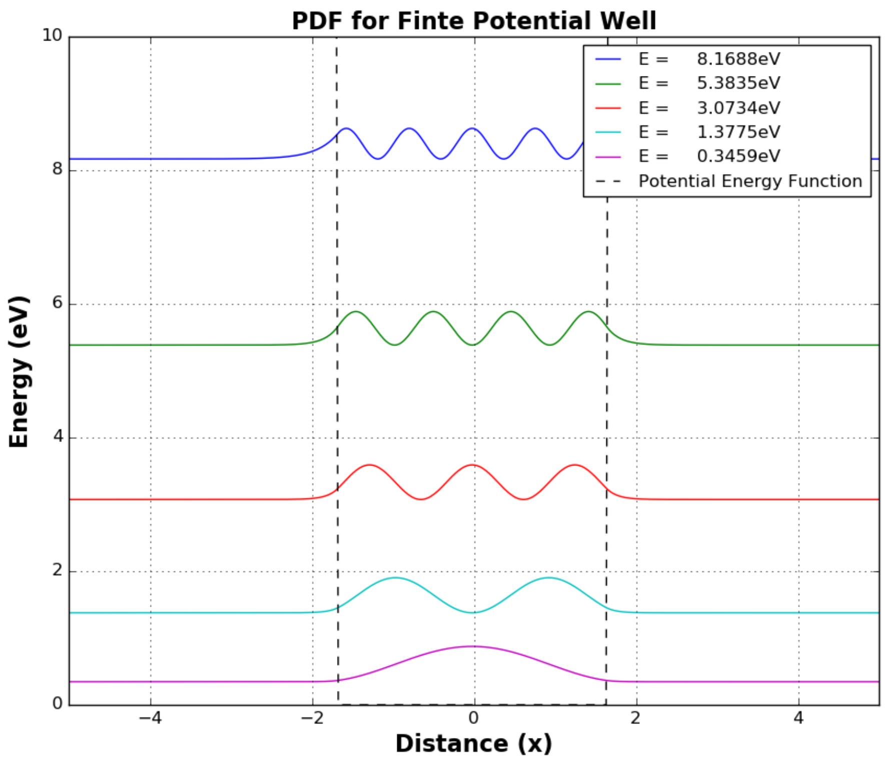

Figure 3.6: Probability Density Function for a Particle in Finite Potential Well

By comparing the two *PDFs* of a particle in Infinte and Finite Potential Wells, we can see that the particle in Finite Potential Well is capable of tunneling through the potential barrier but with lesser probability.

3.4. Step Potential Barrier

In this scenario, I have created a Heavy-Side Finite Potential Barrier where a particle travels from -x direction towards it. From the *PDFs* we can see that the particles can tunnel though to the potential barrier to some extend but ultimately the particles are bounced back towards -x direction.

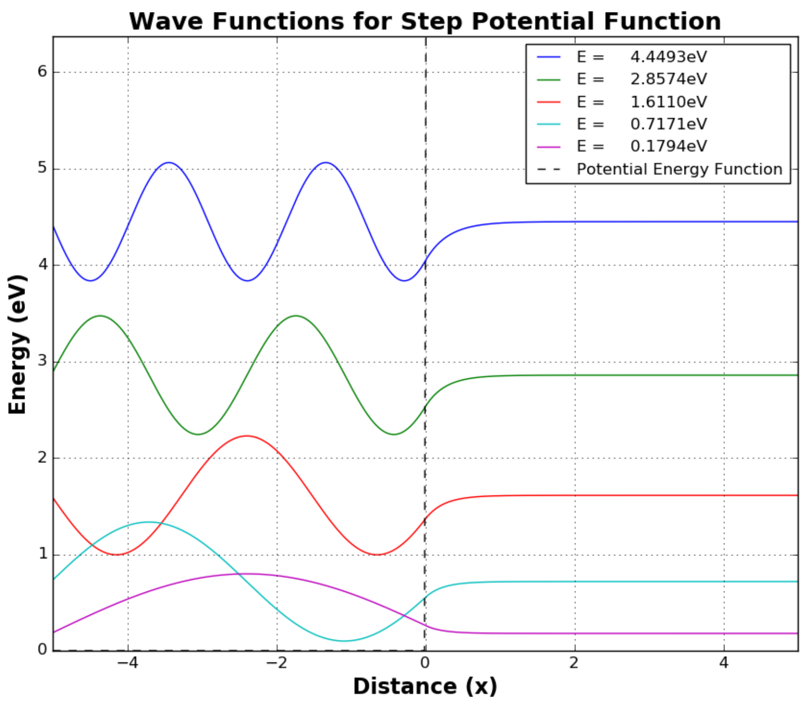

Figure 3.7: Wave Functions for a Particle traveling from -x direction towards a Finite Step Potential Barrier

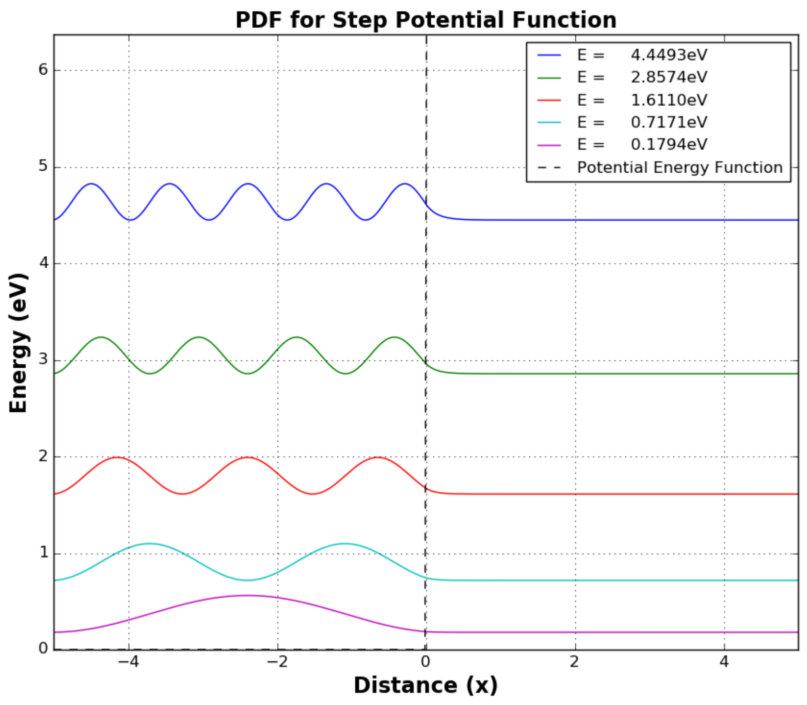

Figure 3.8: Probability Density Function for a Particle traveling from -x direction towards a Finite Step Potential Barrier

3.5. Linear Potential

I have modeled the linear potential energy function as $V(x) = x$.

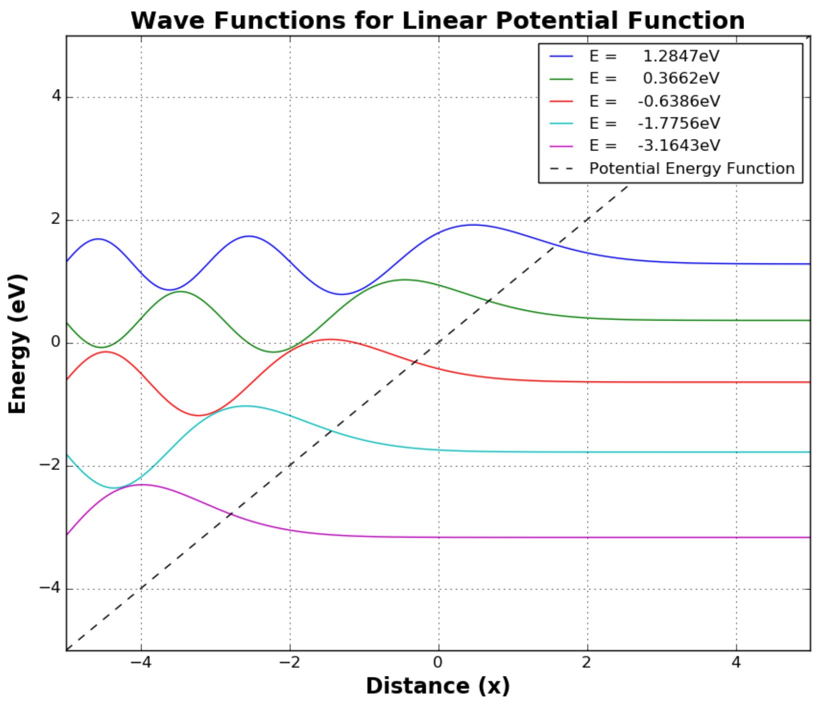

Figure 3.9: Wave Functions for a Particle traveling towards a Linear Potential Barrier

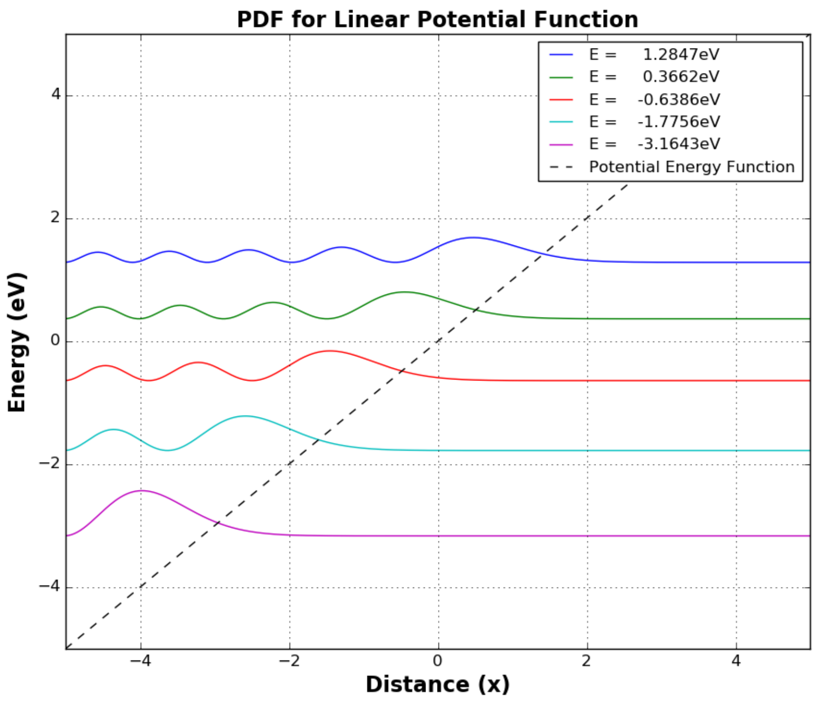

Figure 3.10: Probability Density Function for a Particle traveling towards a Linear Potential Barrier

3.6. Triangular Potential barrier

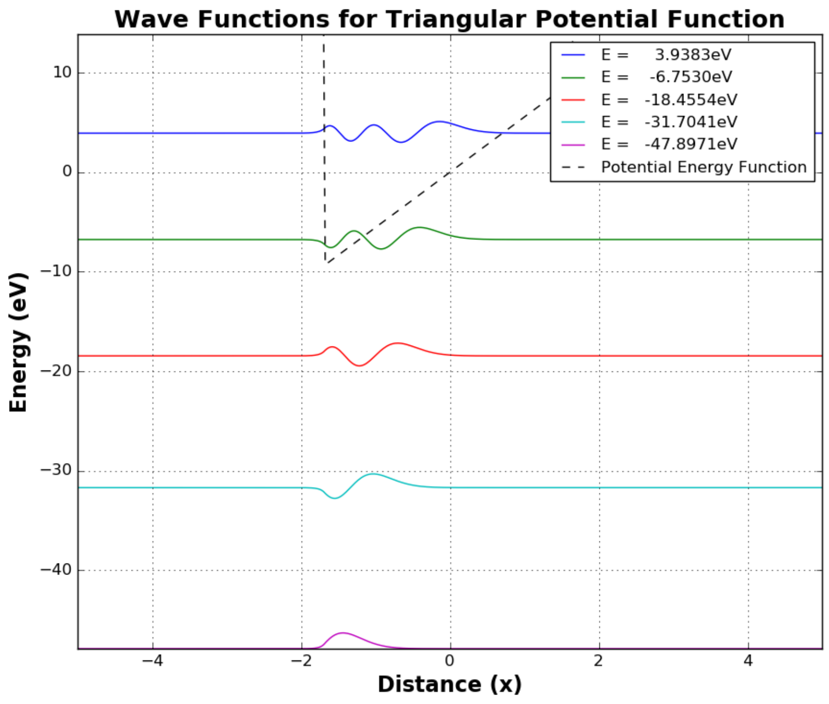

Figure 3.11: Wave Functions for a Particle traveling through a Triangular Potential Barrier

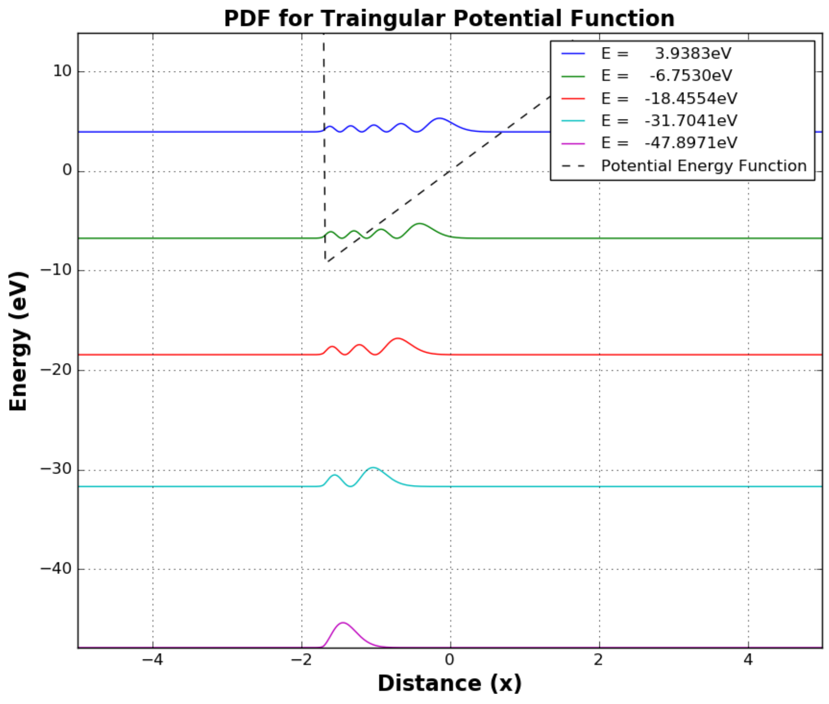

Figure 3.12: Probability Density Function for a Particle traveling through a Triangular Potential Barrier

3.7. Custom Potential barrier

Here, instead of a potential well, I have created a potential tower of 100eV.

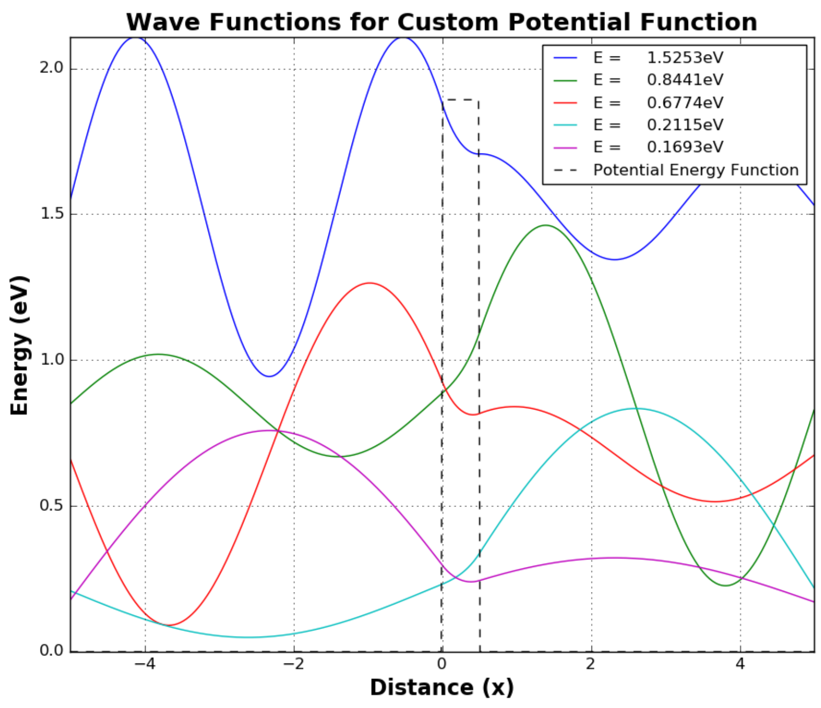

Figure 3.13: Wave Functions for a Particle traveling through a Custom Potential Barrier

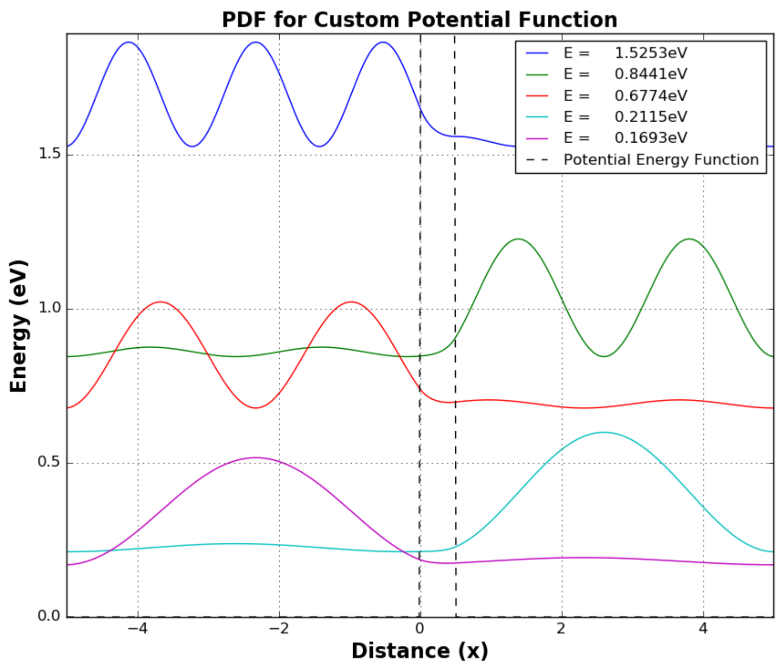

Figure 3.14: Probability Density Functions for a Particle traveling through a Custom Potential Barrier

3.8. Harmonic Oscillator

In this scenario, I have created a Harmonic Oscillator Potential Energy Barrier in the form of $V(x) = x^2$.

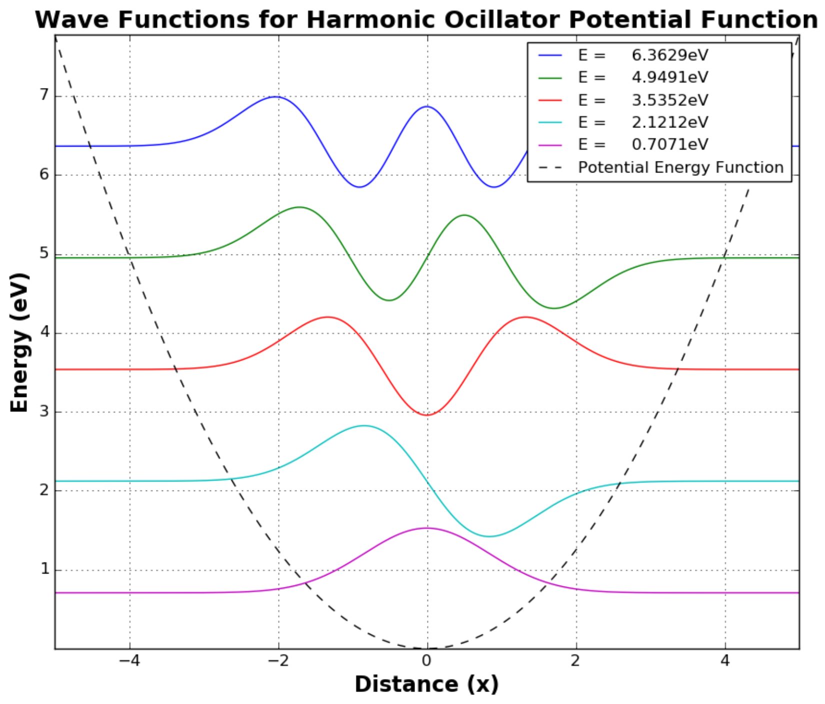

Figure 3.16: Wave Functions for a Particle in Harmonic Oscillator Potential Function

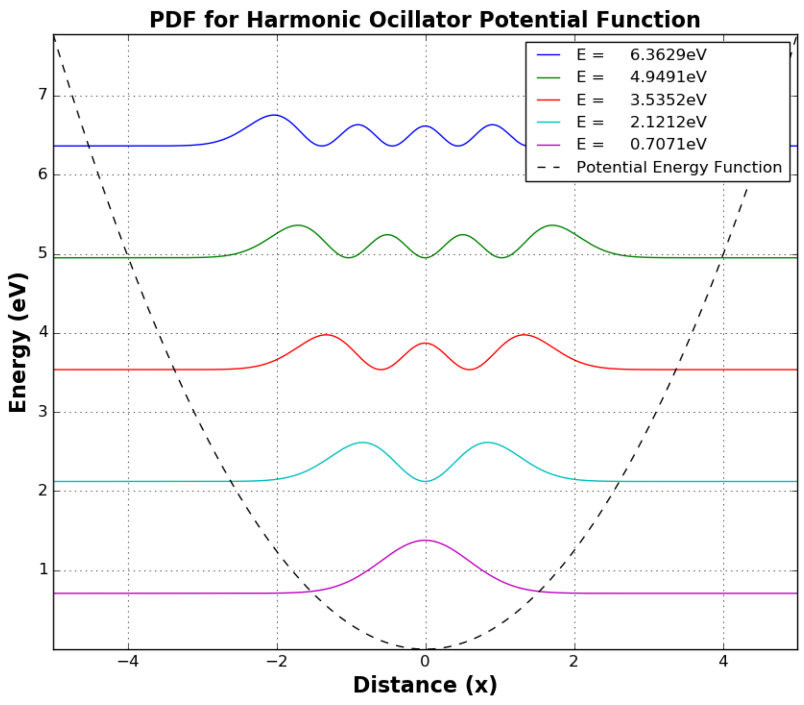

Figure 3.15: Probability Density Functions for a Particle in Harmonic Oscillator Potential Function

4. DISCUSSION

In this report, I have shown how the Schrödinger's wave equation can be modeled as a vector equation using the Finite difference method and how the eigen vectors and eigen energy values looks like for each of the potential energy functions for 1-Dimension.

4.1. Possibility of extending to higher dimensions

This method can be extended to 2-Dimensional or 3-Dimensional by approximating the Laplacian for each dimension using the three-point finite difference method. When representing the *Potential Energy Functions* and *Total energy of a particle*, as they are scalar functions, we can represent them as a summation of energies for each dimension and solve the equation.

Example: **2-Dimensional**

The 2-Dimensional Schrödinger's Wave Equation can be written as follows.

$$\frac{-\hbar^2}{2m}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)\Psi(x,y)+V(x,y)\Psi(x,y)=E\Psi(x,y)$$

Using Separation of variables,

$$\Psi(x,y)=\Psi_X(x)\Psi_Y(y)$$

$$\frac{-\hbar^2}{2m}\left(\Psi_Y(y)\left(\frac{\partial^2}{\partial x^2}\right)\Psi_X(x)+\Psi_X(x)\left(\frac{\partial^2}{\partial y^2}\right)\Psi_Y(y)\right)+(V_X(x)+V_Y(y))\Psi_X(x)\Psi_Y(y)=(\varepsilon_X+\varepsilon_Y)\Psi_X(x)\Psi_Y(y)$$

Leading to

$$\frac{-\hbar^2}{2m}\left(\frac{\partial^2}{\partial x^2}\right)\Psi_X(x)+V_X(x)\Psi_X(x)=\varepsilon_X\Psi_X(x)$$

$$\frac{-\hbar^2}{2m}\left(\frac{\partial^2}{\partial y^2}\right)\Psi_Y(y)+V_Y(y)\Psi_Y(y)=\varepsilon_Y\Psi_Y(y)$$

And

$$\left(\frac{\partial^2}{\partial x^2}\right)\Psi_X(x)\approx\frac{\Psi_X(x+dx)+\Psi_X(x-dx)-2\Psi_X(x)}{(dx)^2}$$

$$\left(\frac{\partial^2}{\partial y^2}\right)\Psi_Y(y)\approx\frac{\Psi_Y(y+dy)+\Psi_Y(y-dy)-2\Psi_Y(y)}{(dy)^2}$$

Now, we can see that this has simplified to *two* 1-Dimenstional problems which we can solve using the method described in this report.

The eigen vector corresponding to the n^{th} eigen energy value would be:

$$|\psi\rangle_n = |\psi_X\rangle_n . (|\psi_Y\rangle_n)^T$$

And the n^{th} Eigen Energy Value would be

$$E_{(nx,ny)} = (\varepsilon_X)_{nx} + (\varepsilon_Y)_{ny}$$

Following shows the results I obtained using the above discussed method for three *Potential Energy Functions.* (Here I have plotted for $E_{(1,1)}$, $E_{(2,2)}$, $E_{(3,3)}$, $E_{(4,4)}$)

A. <u>Near Infinite 2D Potential Well</u>

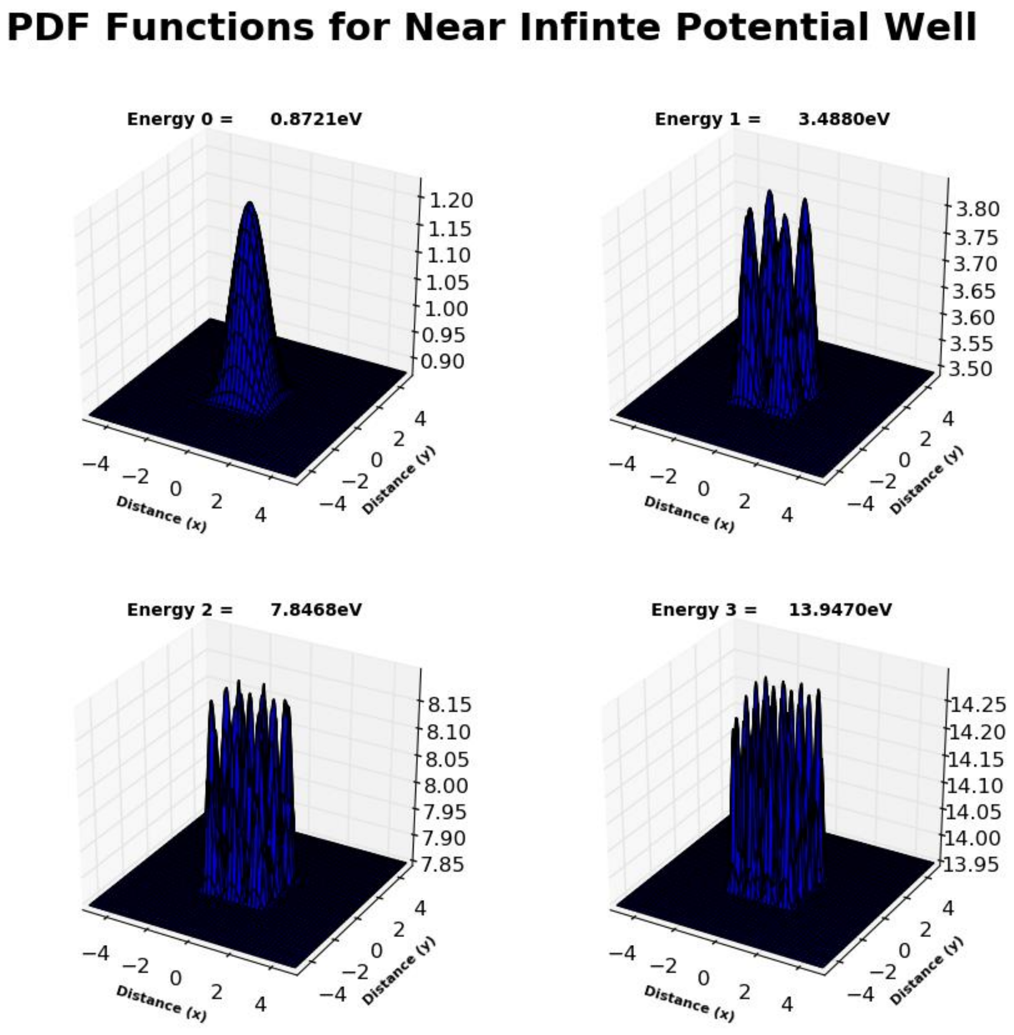

Figure 4.1: Probability Density Functions for Near Infinite 2D Potential Well

B. Finite 2D Potential Well

We can observe that the probability density curves for Infinite Potential Well are strictly limited to the area with 0eV but the curves for Finite Potential Well penetrates the finite potential barrier as well.

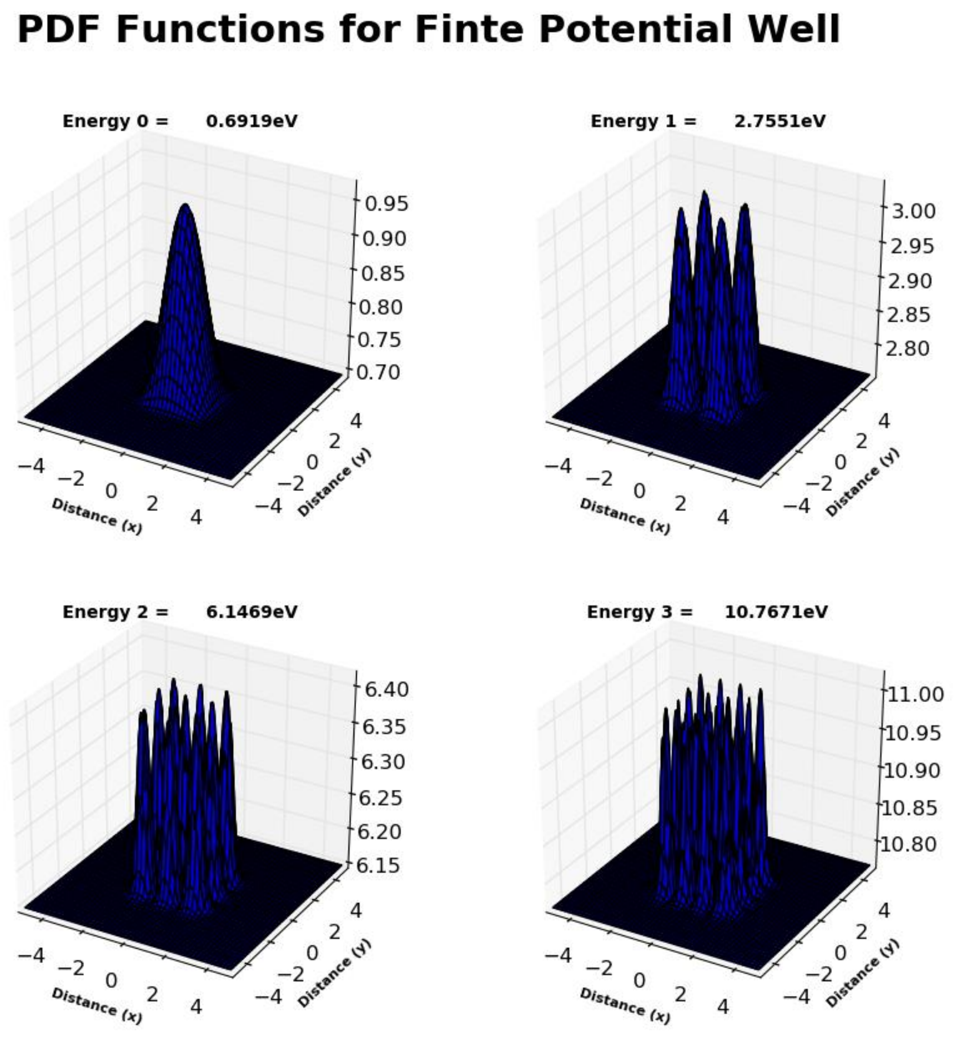

Figure 4.2: Probability Density Functions for Finite 2D Potential Well

C. <u>Harmonic Oscillator</u>

In this I have modeled a paraboloid potential energy function.

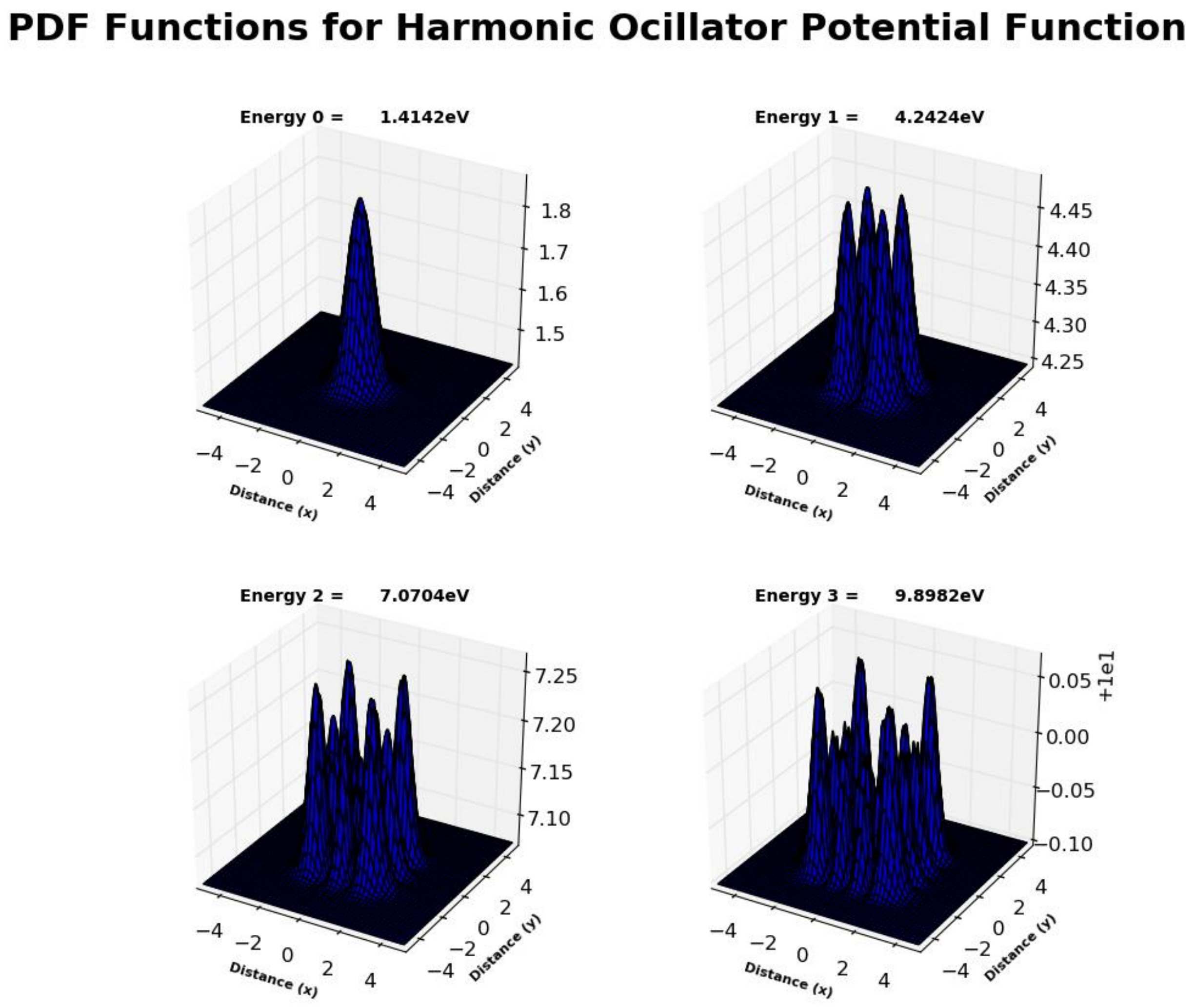

Figure 4.3: Probability Density Function for Paraboloid Potential Function

Even though this method gives good approximations for the above scenarios, it cannot be generalized to every scenario. This is because the assumption that we made on the separation of variables is not true for most of the cases.

Similar analysis can be done for 3-Dimentional scenarios as well but since the visualization cannot be done, I did not implemented it in here.

4.2. Effect of the number of samples on accuracy and time complexity

Further, I used $N = 500$ to obtain the above results. Higher the number of samples we get, lower the approximation error hence the results are more accurate. But when we take more sample points, since we need to create a matrix of the size $N \times N$, it leads to more space in the memory and increases the time complexity of the process as well.

A comparison of the $PDFs$ I obtained for finite well problem for $N = 500, N = 2000$ (Keeping the x-coordinate range the same) are shown in. We can see that even though the We can see that the calculated result by the program with $N = 2000$ are marginally close to the values we obtained by solving the ODE than $N = 500$ but it not worthy with the time complexity it causes.

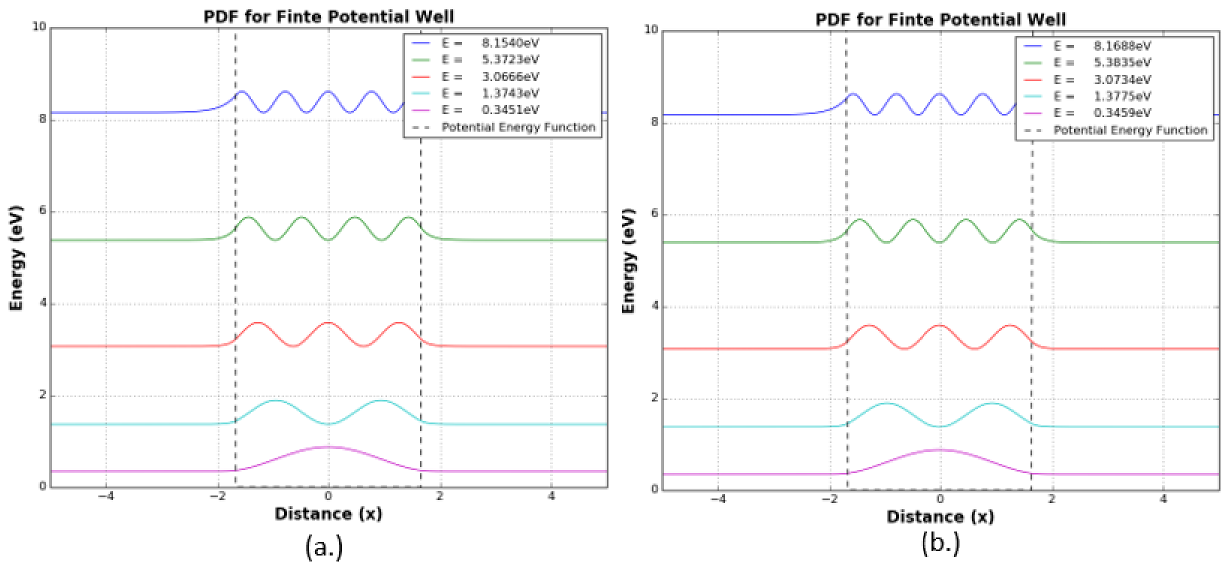

Figure 4.4: Comparison between different sample sizes: (a.) N=500 (b.) N=2000

5. CONCLUSION

The eigenvector method gives solutions which are much closer to the solutions we obtained by solving the ODEs manually. Further the $\hbar$ *and* m are approximated to 1 which leads to more stable solutions as the program will not have to deal with more small numbers in that case. Higher the samples between a given range of x co-ordinates, the results tend to be more accurate with an increase in the time complexity. Further the accuracy can be increased by creating more accurate *Laplacian Matrix* using higher number of finite difference equations than the three-point method (like a seven-point finite difference method).

6. APPENDIX

6.1. PYTHON CODE-1D

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eigh
import matplotlib

plt.style.use('classic')

#Function for creating the Hamiltonian matrix
def Hamiltonian(N, dx, potential, h_bar, mass):
    Laplacian = (-2*np.diag(np.ones((N)),0) + np.diag(np.ones((N-1)),1) + np.diag(np.ones((N-1)),-1))/(dx**2)
    H = -(h_bar**2/(2*mass))*Laplacian + np.diag(potential)
    eigen_energy, eigen_vectors = eigh(H)

    probability = np.abs(eigen_vectors)**2
    K = np.sum(probability, axis=0)*dx
    norm_vectors = eigen_vectors/np.sqrt(K)          #Normalizing the eigen vectors
    probability /= K                                 #Normalizing the PDFs
    return norm_vectors, eigen_energy, probability

#Plotting Functions
def plotWavefunc(x, potential, num_states, wave_vects, energy, func_name):
    plt.figure('WaveFunc')
    Labels = []

    for i in range(num_states-1, -1, -1):
        plt.plot(x, wave_vects[:,i] + energy[i])
        Labels.append('E = {:10.4f}eV'.format(energy[i]))

    Labels.append('Potential Energy Function')

    if 'Linear' in func_name:
        plt.plot(x, np.transpose(potential)*max(max(energy[:num_states+1]), max(potential))/max(potential+10*
*-40), 'k--')
    else:
        plt.plot(x, np.transpose(potential)*min(max(energy[:num_states+1]), max(potential))/max(potential+10**
-40), 'k--')

    plt.xlabel('Distance (x)', fontdict={'weight': 'bold', 'size': 16, 'color': 'black'})
    plt.ylabel('Energy (eV)', fontdict={'weight': 'bold', 'size': 16, 'color': 'black'})
    plt.legend(Labels, loc='upper right', fontsize = 'medium')
    plt.title(f'Wave Functions for {func_name}', fontdict={'weight': 'bold', 'size': 18, 'color': 'black'})
    plt.grid(b=True)
    plt.autoscale(tight=True, axis='both')
    plt.savefig(f'wavePlot-{func_name}.png')
    plt.autoscale(tight=True)
    plt.close('all')
    return

def plotProbfunc(x, potential, num_states, prob_vects, energy, func_name):
    plt.figure('PDF')
    Labels = []

    for i in range(num_states-1, -1, -1):
        plt.plot(x, prob_vects[:,i] + energy[i])
        Labels.append('E = {:10.4f}eV'.format(energy[i]))

    Labels.append('Potential Energy Function')

    if 'Linear' in func_name:
        plt.plot(x, np.transpose(potential)*max(max(energy[:num_states+1]), max(potential))/max(potential+10*
*-40), 'k--')
    else:
        plt.plot(x, np.transpose(potential)*min(max(energy[:num_states+1]), max(potential))/max(potential+10**
-40), 'k--')

    plt.xlabel('Distance (x)', fontdict={'weight': 'bold', 'size': 16, 'color': 'black'})
    plt.ylabel('Energy (eV)', fontdict={'weight': 'bold', 'size': 16, 'color': 'black'})
    plt.legend(Labels, loc='upper right', fontsize = 'medium')
    plt.title(f'PDF for {func_name}', fontdict={'weight': 'bold', 'size': 16, 'color': 'black'})
    plt.grid(b=True)
    plt.autoscale(tight=True, axis='both')
    plt.savefig(f'probPlot-{func_name}.png')
    plt.autoscale(tight=True)
    plt.close('all')
    return

#Potential Enegery Functions
def FreeParticle(N, dx, h_bar, mass, x, n_states):
    V_x = np.zeros((N))
    wave_vec, energy, prob = Hamiltonian(N,dx, V_x, h_bar, mass)
    plotWavefunc(x, V_x, n_states, wave_vec, energy, 'Free-Particle')
    plotProbfunc(x, V_x, n_states, prob, energy, 'Free-Particle')
    return
```

```
def infiniteWell(N, dx, h_bar, mass, x, n_states):
    n = np.floor(N/3).astype('int32')
    V_x = np.ones((N)) * 2000
    V_x[n:2*n] = 0
    wave_vec, energy, prob = Hamiltonian(N,dx, V_x, h_bar, mass)
    plotWavefunc(x, V_x, n_states, wave_vec, energy, 'Near Infinte Potential Well')
    plotProbfunc(x, V_x, n_states, prob, energy, 'Near Infinte Potential Well')
    return

def finiteWell(N, dx, h_bar, mass, x, n_states):
    n = np.floor(N/3).astype('int32')
    V_x = np.ones((N)) * 10
    V_x[n:2*n] = 0
    wave_vec, energy, prob = Hamiltonian(N,dx, V_x, h_bar, mass)
    plotWavefunc(x, V_x, n_states, wave_vec, energy, 'Finte Potential Well')
    plotProbfunc(x, V_x, n_states, prob, energy, 'Finte Potential Well')
    return

def linearfunc(N, dx, h_bar, mass, x, n_states):
    V_x = x
    wave_vec, energy, prob = Hamiltonian(N,dx, V_x, h_bar, mass)
    plotWavefunc(x, V_x, n_states, wave_vec, energy, 'Linear Potential Function')
    plotProbfunc(x, V_x, n_states, prob, energy, 'Linear Potential Function')
    return

def HarmonicOcilator(N, dx, h_bar, mass, x, n_states):
    V_x = x**2
    wave_vec, energy, prob = Hamiltonian(N,dx, V_x, h_bar, mass)
    plotWavefunc(x, V_x, n_states, wave_vec, energy, 'Harmonic Ocillator Potential Function')
    plotProbfunc(x, V_x, n_states, prob, energy, 'Harmonic Ocillator Potential Function')
    return

def StepBarrier(N, dx, h_bar, mass, x, n_states):
    V_x = np.zeros((N))
    V_x[N//2:] += 10
    wave_vec, energy, prob = Hamiltonian(N,dx, V_x, h_bar, mass)
    plotWavefunc(x, V_x, n_states, wave_vec, energy, 'Step Potential Function')
    plotProbfunc(x, V_x, n_states, prob, energy, 'Step Potential Function')
    return

def Triangular(N, dx, h_bar, mass, x, n_states):
    n = np.floor(N/3).astype('int32')
    V_x = np.ones((N)) * 100
    V_x[n:2*n] = x[n:2*n]*40
    wave_vec, energy, prob = Hamiltonian(N,dx, V_x, h_bar, mass)
    plotWavefunc(x, V_x, n_states, wave_vec, energy, 'Triangular Potential Function')
    plotProbfunc(x, V_x, n_states, prob, energy, 'Traingular Potential Function')
    return

def CustomBarrier(N, dx, h_bar, mass, x, n_states):
    n = np.floor(N/20).astype('int32')
    V_x = np.zeros((N))
    V_x[n*10:n*11] = 5
    wave_vec, energy, prob = Hamiltonian(N,dx, V_x, h_bar, mass)
    plotWavefunc(x, V_x, n_states, wave_vec, energy, 'Custom Potential Function')
    plotProbfunc(x, V_x, n_states, prob, energy, 'Custom Potential Function')
    return

if __name__ == '__main__':
    if len(sys.argv) < 4:
        print('| python file | x limit | number of points | number of eigen states |')
    else:
        xLimit = int(sys.argv[1])                    #user input for x-co-ordinate limit
        N = int(sys.argv[2])                         #user input for the number of samples
        eigen_states = int(sys.argv[3])              #user input for the number of eigen states to look at

        print('Enter- 0:Free particle | 1: Near infinite well | 2: Finite well | 3: Linear | 4: Harmonic Oscillator | 5: S
tep Barrier | 6: Triangular | 7: Custom Barrier')

        try:
            potential_func = int(input('Enter the potential function number: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 :'))        #user in
put for type of potential energy
        except:
            print('Enter- 0:Free particle | 1: Near infinite well | 2: Finite well | 3: Linear | 4: Harmonic Oscillator | 5:
 Step Barrier | 6: Triangular | 7: Custom Barrier')

        x = np.linspace(-xLimit, xLimit, N)
        dx = x[1] - x[0]
        h_bar = 1
        mass = 1

        plt.rcParams['figure.figsize'] = (10,8)
        plt.rc('legend', fontsize=12)
        plt.rcParams['legend.frameon'] = True

        if potential_func == 0:
            FreeParticle(N, dx, h_bar, mass, x, eigen_states )

        elif potential_func == 1:
```

```
        infiniteWell(N, dx, h_bar, mass, x, eigen_states )

    elif potential_func == 2:
        finiteWell(N, dx, h_bar, mass, x, eigen_states )

    elif potential_func == 3:
        linearfunc(N, dx, h_bar, mass, x, eigen_states )

    elif potential_func == 4:
        HarmonicOcilator(N, dx, h_bar, mass, x, eigen_states )

    elif potential_func == 5:
        StepBarrier(N, dx, h_bar, mass, x, eigen_states )

    elif potential_func == 6:
        Triangular(N, dx, h_bar, mass, x, eigen_states )

    elif potential_func == 7:
        CustomBarrier(N, dx, h_bar, mass, x, eigen_states)

    else:
        print('Invalida input')
```

6.2. PYTHON CODE – 2D

```
mport sys
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eigh
import matplotlib
from mpl_toolkits.mplot3d import Axes3D


plt.style.use('classic')


#Function for creating the Hamiltonian matrix
def Hamiltonian(N, dx, potential, h_bar, mass):

    Laplacian = (-2*np.diag(np.ones((N)),0) + np.diag(np.ones((N-
1)),1) + np.diag(np.ones((N-1)),-1))/(dx**2)
    H = -(h_bar**2/(2*mass))*Laplacian + np.diag(potential)
    eigen_energy, eigen_vectors = eigh(H); print(eigen_energy.shape)
    probability = np.abs(eigen_vectors)**2
    K = np.sum(probability, axis=0)*dx
    norm_vectors = eigen_vectors/np.sqrt(K)             #Normalizing the eigen ve
ctors
    probability /= K                                    #Normalizing the PDFs
    return norm_vectors, eigen_energy, probability

#Plotting Functions
def plotWavefunc(x, potential, num_states, wave_vectsX, wave_vectsY, energy, func
_name):
    fig = plt.figure('WaveFunc', facecolor='w', edgecolor='k')

    for i in range(num_states-1, -1, -1):

        ax = fig.add_subplot(num_states//2, num_states//2, i+1,  projection='3d')

        X, Y = np.meshgrid(x, x)

        wave_vect = np.reshape(wave_vectsX[:,i], (len(wave_vectsX),1))*np.transpo
se(np.reshape(wave_vectsY[:,i], (len(wave_vectsX),1)))
        ax.plot_surface(X, Y, wave_vect + energy[i])

        plt.xlabel('Distance (x)', fontdict={'weight': 'bold', 'size': 8, 'color'
: 'black'})
        plt.ylabel('Distance (y)', fontdict={'weight': 'bold', 'size': 8, 'color'
: 'black'})
        ax.set_title('Energy {} =  {:10.4f}eV'.format(i,energy[i]), fontdict={'fo
ntsize': 10, 'fontweight': 'bold'})
        plt.grid(b=True)
        plt.autoscale(tight=True)

    plt.suptitle(f'Wave Functions for {func_name}', fontsize = 22, fontweight = '
bold')
    plt.savefig(f'wavePlot-{func_name}.png')
    plt.close('all')
    return

def plotProbfunc(x, potential, num_states, wave_vectsX, wave_vectsY, energy, func
_name):
    fig = plt.figure('PDF', facecolor='w', edgecolor='k')

    for i in range(num_states-1, -1, -1):

        ax = fig.add_subplot(num_states//2, num_states//2, i+1,  projection='3d')
        X, Y = np.meshgrid(x, x)

        wave_vect = np.reshape(wave_vectsX[:,i], (len(wave_vectsX),1))*np.transpo
se(np.reshape(wave_vectsY[:,i], (len(wave_vectsX),1)))
        probability = np.abs(wave_vect)**2

        ax.plot_surface(X, Y, probability + energy[i])

        plt.xlabel('Distance (x)', fontdict={'weight': 'bold', 'size': 8, 'color'
: 'black'})
        plt.ylabel('Distance (y)', fontdict={'weight': 'bold', 'size': 8, 'color'
: 'black'})
        ax.set_title('Energy {} =  {:10.4f}eV'.format(i,energy[i]), fontdict={'fo
ntsize': 10, 'fontweight': 'bold'})
        plt.grid(b=True)
        plt.autoscale(tight=True)

    plt.suptitle(f'PDF Functions for {func_name}', fontsize = 22, fontweight = 'b
old')
    plt.savefig(f'PDFPlot-{func_name}.png')
    plt.close('all')
    return

def infiniteWell(N, dx, h_bar, mass, x, n_states):
    n = np.floor(N/3).astype('int32')
    V_x = np.ones((N)) * 2000
    V_x[n:2*n] = 0
    wave_vec_x, energy_x, _ = Hamiltonian(N,dx, V_x, h_bar, mass)   # Solution fo
r x-direction
    wave_vec_y, energy_y, _ = Hamiltonian(N,dx, V_x, h_bar, mass)   # Solution fo
r y-direction

    energy = energy_x + energy_y; print(energy.shape)               # Total Eigen
 Energy values
    plotWavefunc(x, V_x, n_states, wave_vec_x, wave_vec_y, energy, 'Near Infinte
Potential Well')
    plotProbfunc(x, V_x, n_states, wave_vec_x, wave_vec_y, energy, 'Near Infinte
Potential Well')
    return

def finiteWell(N, dx, h_bar, mass, x, n_states):
    n = np.floor(N/3).astype('int32')
    V_x = np.ones((N)) * 10
    V_x[n:2*n] = 0
    wave_vec_x, energy_x, _ = Hamiltonian(N,dx, V_x, h_bar, mass)   # Solution fo
r x-direction
    wave_vec_y, energy_y, _ = Hamiltonian(N,dx, V_x, h_bar, mass)   # Solution fo
r y-direction

    energy = energy_x + energy_y; print(energy.shape)               # Total Eigen
 Energy values
    plotWavefunc(x, V_x, n_states, wave_vec_x, wave_vec_y, energy, 'Finte Potenti
al Well')
    plotProbfunc(x, V_x, n_states, wave_vec_x, wave_vec_y, energy, 'Finte Potenti
al Well')
    return

def HarmonicOcilator(N, dx, h_bar, mass, x, n_states):
    V_x = x**2
    wave_vec_x, energy_x, _ = Hamiltonian(N,dx, V_x, h_bar, mass)
    wave_vec_y, energy_y, _ = Hamiltonian(N,dx, V_x, h_bar, mass)

    energy = energy_x + energy_y
    plotWavefunc(x, V_x, n_states, wave_vec_x, wave_vec_y, energy, 'Harmonic Ocil
lator Potential Function')
    plotProbfunc(x, V_x, n_states, wave_vec_x, wave_vec_y, energy, 'Harmonic Ocil
lator Potential Function')


    return

def StepBarrier(N, dx, h_bar, mass, x, n_states):
    V_x = np.zeros((N))
    V_x[N//2:] += 10
    wave_vec_x, energy_x, _ = Hamiltonian(N,dx, V_x, h_bar, mass)
    wave_vec_y, energy_y, _ = Hamiltonian(N,dx, V_x, h_bar, mass)

    energy = energy_x + energy_y
    plotWavefunc(x, V_x, n_states, wave_vec_x, wave_vec_y, energy, 'Step Potentia
l Function')
    plotProbfunc(x, V_x, n_states, wave_vec_x, wave_vec_y, energy, 'Step Potentia
l Function')
```