A Detailed Study of 1D Quantum Linear Harmonic Oscillator Using Python

A Dissertation Project

Submitted for the ***DSE 4 paper (6th Semester)*** *in*

B.Sc. Physics Honours

by

Suman Kumar Pal

Registration no.: 041702 of 2021-2022
Roll no.: 2116134-2139168

Under the guidance of

Dr. Anita Gangopadhyay
Associate Professor, Santipur College

Dr. Dipankar Bhattacharyya
Associate Professor, Santipur College

Department of Physics
Santipur College (affiliated to University of Kalyani)
Santipur, Nadia, West Bengal, PIN - 741404

July 13, 2024

Acknowledgements

This project is based on Quantum Mechanics, numerical solutions and Python implementation of a problem. I am grateful to Dr. Anita Gangopadhyay ma'am for her lectures on numerical solutions and Python implementation and Dr. Dipankar Bhattacharyya sir for his lectures on Quantum Mechanics. All these lectures helped me a lot to develop concepts on those topics. After studying these topics I got the idea to apply Python for some Quantum Mechanical problems and have tried that in this project.

A Detailed Study of 1D Quantum Linear Harmonic Oscillator Using Python

Suman Kumar Pal

July 13, 2024

Abstract

In this project, I have discussed the Python implementation of Schrödinger equation in detail.

The new works by me in this project are,

- Calculation of position and momentum uncertainties for a 1D linear harmonic oscillator and comparison of results with the exact numerical values (obtained from analytical solutions) (Section 4.5).

- Time evolution of bound state wavefunctions and explaining the concept of stationary states (Section 4.6 and Section 4.7).

- Frequency of oscillation of a superposed state (Section 4.8).

- Some possible future works (Chapter 6).

Contents

Chapter 1

Introduction

Here in this project, I am going to demonstrate the concepts of basic quantum mechanics by solving Schrödinger equation numerically and then compute that using Python.

In Chapter 2, the time dependent Schrödinger equation is solved numerically (Section 2.2) using Crank-Nicolson method. As an application of the time dependent Schrödinger equation, the evolution of Gaussian wavepackets is discussed in Chapter 5. For solving matrix equations, the LU factorization method (Section 2.3.4) is used. In Chapter 3, a quick method for solving the time independent Schrödinger equation is used.

In Chapter 4, the quantum harmonic oscillator problem is solved in detail. The calculated values of energies and uncertainties are compared with the exact values.

All the codes and animated plots can be found at `https://github.com/suman122003/BSc_Physics_Codes/tree/main/DSE_4_Computational_QM_project`.

Chapter 2

Solution of Time Dependent Schrödinger Equation

2.1 Introduction

The time dependent Schrödinger equation in 1D is,

$$-\frac{\hbar^2}{2m}\frac{\partial\Psi}{\partial x}+V(x)\Psi=i\hbar\frac{\partial\Psi}{\partial t} \tag{2.1}$$

This equation can be solved numerically by discrtization of space and time and then using the finite difference formulae to replace the derivatives.

Discretization of Space and Time

For discretization of space, I am going to use index $\mathrm{i}\in[0,N_x-1]$. Thus for position x, we have the upper bound x_{N_x-1} and lower bound x_0. An intermediate value can be expressed by,

$$x_{\mathrm{i}}=x_0+\mathrm{i}\delta x \tag{2.2}$$

For, Discretization of Time, I am going to use index $\mathrm{n}\geq 0$. So, for time, the lower bound is $t=0$ or $t=t_0$ (i.e. initial time). A later time can be expressed as,

$$t_{\mathrm{n+1}}=t_{\mathrm{n}}+\delta t \tag{2.3}$$

Here, I am going to write the position index i as a subscript and time index n as a superscript, i.e. I will represent $\Psi(x=x_{\mathrm{i}},t=t_{\mathrm{n}})$ by $\Psi_{\mathrm{i}}^{\mathrm{n}}$ throughout the project.

2.2 Numerical Solution using Crank-Nicolson method

In Eq. (2.1) we have 1st order differentiation in time and 2nd second-order differentiation in space. This kind of differential equations are called *parabolic partial differential equations* (as this equation resembles the equation of parabola). One of the most useful ways to solve the parabolic partial differential equation is the *Crank-Nicolson method* [2].

The numerical solution is given in [2] [4] (page 439) and [6].

The Taylor series expansion of $\Psi_{\mathrm{i}}^{\mathrm{n+1}}$ about the point $(\mathrm{i}, \mathrm{n}+\frac{1}{2})$ when $\frac{\delta t}{2}$ step in time is considered,

$$\Psi_{\mathrm{i}}^{\mathrm{n+1}} = \Psi_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} + \frac{\delta t}{2}\left(\frac{\partial \Psi}{\partial t}\right)_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} + \frac{1}{2}\left(\frac{\delta t}{2}\right)^2\left(\frac{\partial^2 \Psi}{\partial t^2}\right)_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} + \mathcal{O}(\delta t^3)$$

Similarly, the expansion of $\Psi_{\mathrm{i}}^{\mathrm{n}}$ will be,

$$\Psi_{\mathrm{i}}^{\mathrm{n}} = \Psi_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} - \frac{\delta t}{2}\left(\frac{\partial \Psi}{\partial t}\right)_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} + \frac{1}{2}\left(\frac{\delta t}{2}\right)^2\left(\frac{\partial^2 \Psi}{\partial t^2}\right)_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} + \mathcal{O}(\delta t^3)$$

By simplifying these equations,

$$\left(\frac{\partial \Psi}{\partial t}\right)_{\mathrm{i}}^{\mathrm{n+1}} = \frac{\Psi_{\mathrm{i}}^{\mathrm{n+1}} - \Psi_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}}}{\frac{\delta t}{2}} - \frac{1}{2}\frac{(\delta t/2)^2}{\delta t/2}\left(\frac{\partial^2 \Psi}{\partial t^2}\right)_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} + \frac{\mathcal{O}(\delta t^3)}{\delta t/2}$$

$$\left(\frac{\partial \Psi}{\partial t}\right)_{\mathrm{i}}^{\mathrm{n}} = \frac{\Psi_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} - \Psi_{\mathrm{i}}^{\mathrm{n}}}{\frac{\delta t}{2}} + \frac{1}{2}\frac{(\delta t/2)^2}{\delta t/2}\left(\frac{\partial^2 \Psi}{\partial t^2}\right)_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} + \frac{\mathcal{O}(\delta t^3)}{\delta t/2}$$

$$\begin{aligned}\left(\frac{\partial \Psi}{\partial t}\right)_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} &= \frac{1}{2}\left(\left(\frac{\partial \Psi}{\partial t}\right)_{\mathrm{i}}^{\mathrm{n+1}} + \left(\frac{\partial \Psi}{\partial t}\right)_{\mathrm{i}}^{\mathrm{n}}\right)\\ &= \frac{1}{2}\left(\frac{\Psi_{\mathrm{i}}^{\mathrm{n+1}} - \Psi_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}}}{\frac{\delta t}{2}} + \frac{\Psi_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} - \Psi_{\mathrm{i}}^{\mathrm{n}}}{\frac{\delta t}{2}}\right) + \mathcal{O}(\delta t^2)\end{aligned}$$

$$\implies \left(\frac{\partial \Psi}{\partial t}\right)_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} = \frac{\Psi_{\mathrm{i}}^{\mathrm{n+1}} - \Psi_{\mathrm{i}}^{\mathrm{n}}}{\delta t} + \mathcal{O}(\delta t^2) \tag{2.4}$$

In position, the have second order differentiation. This can be calculated easily by using central difference formula,

$$\left(\frac{\partial^2 \Psi}{\partial x^2}\right)_{\mathrm{i}}^{\mathrm{n}} = \frac{\Psi_{\mathrm{i+1}}^{\mathrm{n}} - 2\Psi_{\mathrm{i}}^{\mathrm{n}} + \Psi_{\mathrm{i-1}}^{\mathrm{n}}}{\delta x^2} + \mathcal{O}(\delta x^2)$$

Now,

$$\left(\frac{\partial^2 \Psi}{\partial x^2}\right)_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} = \frac{1}{2}\left(\left(\frac{\partial^2 \Psi}{\partial x^2}\right)_{\mathrm{i}}^{\mathrm{n+1}} + \left(\frac{\partial^2 \Psi}{\partial x^2}\right)_{\mathrm{i}}^{\mathrm{n}}\right)$$

$$\implies \left(\frac{\partial^2 \Psi}{\partial x^2}\right)_{\mathrm{i}}^{\mathrm{n+\frac{1}{2}}} = \frac{\Psi_{\mathrm{i+1}}^{\mathrm{n+1}} - 2\Psi_{\mathrm{i}}^{\mathrm{n+1}} + \Psi_{\mathrm{i-1}}^{\mathrm{n+1}} + \Psi_{\mathrm{i+1}}^{\mathrm{n}} - 2\Psi_{\mathrm{i}}^{\mathrm{n}} + \Psi_{\mathrm{i-1}}^{\mathrm{n}}}{2\delta x^2} + \mathcal{O}(\delta x^2) \tag{2.5}$$

Also,

$$\Psi_{\mathrm{i}}^{\mathrm{n}+\frac{1}{2}} = \frac{1}{2}(\Psi_{\mathrm{i}}^{\mathrm{n}+1} + \Psi_{\mathrm{i}}^{\mathrm{n}}) \tag{2.6}$$

We can write Eq. (2.1) at the point $(\mathrm{i}, \mathrm{n}+\frac{1}{2})$,

$$-\frac{\hbar^2}{2m}\left(\frac{\partial^2\Psi}{\partial x^2}\right)_{\mathrm{i}}^{\mathrm{n}+\frac{1}{2}} + V(x_{\mathrm{i}})\Psi_{\mathrm{i}}^{\mathrm{n}+\frac{1}{2}} = i\hbar\left(\frac{\partial\Psi}{\partial t}\right)_{\mathrm{i}}^{\mathrm{n}+\frac{1}{2}}$$

$$\implies -\frac{\hbar^2}{2m}\left(\frac{\Psi_{\mathrm{i}+1}^{\mathrm{n}+1} - 2\Psi_{\mathrm{i}}^{\mathrm{n}+1} + \Psi_{\mathrm{i}-1}^{\mathrm{n}+1} + \Psi_{\mathrm{i}+1}^{\mathrm{n}} - 2\Psi_{\mathrm{i}}^{\mathrm{n}} + \Psi_{\mathrm{i}-1}^{\mathrm{n}}}{2\delta x^2} + \mathcal{O}(\delta x^2)\right)$$

$$+\frac{V(x_{\mathrm{i}})}{2}(\Psi_{\mathrm{i}}^{\mathrm{n}+1} + \Psi_{\mathrm{i}}^{\mathrm{n}}) = i\hbar\left(\frac{\Psi_{\mathrm{i}}^{\mathrm{n}+1} - \Psi_{\mathrm{i}}^{\mathrm{n}}}{\delta t} + \mathcal{O}(\delta t^2)\right)$$

This equation has an error of $\mathcal{O}(\delta x^2 + \delta t^2)$, which is very less. Now, take

$$\alpha = \frac{\hbar\delta t}{4m\delta x^2} \quad \text{and} \quad \beta = \frac{\delta t}{2\hbar} \tag{2.7}$$

So, the above equation will be,

$$-\alpha(\Psi_{\mathrm{i}+1}^{\mathrm{n}+1} - 2\Psi_{\mathrm{i}}^{\mathrm{n}+1} + \Psi_{\mathrm{i}-1}^{\mathrm{n}+1} + \Psi_{\mathrm{i}+1}^{\mathrm{n}} - 2\Psi_{\mathrm{i}}^{\mathrm{n}} + \Psi_{\mathrm{i}-1}^{\mathrm{n}}) + \beta V(x_{\mathrm{i}})(\Psi_{\mathrm{i}}^{\mathrm{n}+1} + \Psi_{\mathrm{i}}^{\mathrm{n}}) = i(\Psi_{\mathrm{i}}^{\mathrm{n}+1} - \Psi_{\mathrm{i}}^{\mathrm{n}})$$

$$\implies (\beta V(x_{\mathrm{i}}) + 2\alpha - i)\Psi_{\mathrm{i}}^{\mathrm{n}+1} + (\beta V(x_{\mathrm{i}}) + 2\alpha + i)\Psi_{\mathrm{i}}^{\mathrm{n}} = \alpha(\Psi_{\mathrm{i}+1}^{\mathrm{n}+1} + \Psi_{\mathrm{i}-1}^{\mathrm{n}+1} + \Psi_{\mathrm{i}+1}^{\mathrm{n}} + \Psi_{\mathrm{i}-1}^{\mathrm{n}})$$

Take

$$\gamma_{\mathrm{i}} = \beta V(x_{\mathrm{i}}) + 2\alpha \tag{2.8}$$

Thus,

$$-\alpha\Psi_{\mathrm{i}-1}^{\mathrm{n}+1} + (\gamma_{\mathrm{i}} - i)\Psi_{\mathrm{i}}^{\mathrm{n}+1} - \alpha\Psi_{\mathrm{i}+1}^{\mathrm{n}+1} = \alpha\Psi_{\mathrm{i}-1}^{\mathrm{n}} - (\gamma_{\mathrm{i}} + i)\Psi_{\mathrm{i}}^{\mathrm{n}} + \alpha\Psi_{\mathrm{i}+1}^{\mathrm{n}} \tag{2.9}$$

where, $\mathrm{i} = 1, 2, 3, \ldots, N_x - 2$.

At the boundary the condition is,

$$\Psi_0^{\mathrm{n}+1} = \Psi_0^{\mathrm{n}} \quad \text{and} \quad \Psi_{N_x-1}^{\mathrm{n}+1} = \Psi_{N_x-1}^{\mathrm{n}} \tag{2.10}$$

for all values of n. The reason for choosing this boundary condition and effect of this, is discussed in Section 3.2.

All these equations can be written together in a matrix equation,

$$A\Psi^{\mathrm{n}+1} = B\Psi^{\mathrm{n}} \tag{2.11}$$

where, $A = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 & 0 \\ -\alpha & \gamma_1 - i & -\alpha & \ldots & 0 & 0 \\ 0 & -\alpha & \gamma_2 - i & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & \gamma_{N_x-2} - i & -\alpha \\ 0 & 0 & 0 & \ldots & 0 & 1 \end{bmatrix}$,

$B = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 & 0 \\ \alpha & -\gamma_1 - i & \alpha & \ldots & 0 & 0 \\ 0 & \alpha & -\gamma_2 - i & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & -\gamma_{N_x-2} - i & \alpha \\ 0 & 0 & 0 & \ldots & 0 & 1 \end{bmatrix}$, $\Psi^{\mathrm{n}+1} = \begin{bmatrix} \Psi_0^{\mathrm{n}+1} \\ \Psi_1^{\mathrm{n}+1} \\ \Psi_2^{\mathrm{n}+1} \\ \vdots \\ \Psi_{N_x-2}^{\mathrm{n}+1} \\ \Psi_{N_x-1}^{\mathrm{n}+1} \end{bmatrix}$ and $\Psi^{\mathrm{n}} = \begin{bmatrix} \Psi_0^{\mathrm{n}} \\ \Psi_1^{\mathrm{n}} \\ \Psi_2^{\mathrm{n}} \\ \vdots \\ \Psi_{N_x-2}^{n} \\ \Psi_{N_x-1}^{\mathrm{n}} \end{bmatrix}$

Our main problem is now reduced to calculation of $\Psi^{\mathrm{n}+1}$ when Ψ^{n}, A and B are known. This can be done by many methods. Several methods are discussed in Section 2.3.

2.3 Methods for Solving Matrix Equations

We have different ways to solve a matrix equation in Python. Selection of a fast method is important because for visualisation of time evolution, the most important task is to solve Eq. (2.11) for a large value of N_x and many values of n. We would write a matrix (similar to the matrix in the mentioned equation) and solve the system of linear equations.

Here I am going to define a function that solves the system of linear equations for tridiagonal matrices only by LU decomposition [5] [6].

Let's define lower and upper triangular matrices L and U such that,

$$A = LU \tag{2.12}$$

where, $L = \begin{bmatrix} l_{00} & 0 & 0 & \dots & 0 \\ l_{10} & l_{11} & 0 & \dots & 0 \\ 0 & l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & l_{N-1,N-1} \end{bmatrix}$ and $U = \begin{bmatrix} 1 & u_{01} & 0 & \dots & 0 \\ 0 & 1 & u_{12} & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$. So,

$$\begin{bmatrix} a_{00} & a_{01} & 0 & \dots & 0 \\ a_{10} & a_{11} & a_{12} & \dots & 0 \\ 0 & a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{N-1,N-1} \end{bmatrix} = \begin{bmatrix} l_{00} & l_{00}u_{01} & 0 & \dots & 0 \\ l_{10} & l_{10}u_{01} + l_{11} & l_{11}u_{12} & \dots & 0 \\ 0 & l_{21} & l_{21}u_{12} + l_{22} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & l_{N-1,N-2}u_{N-2,N-1} + l_{N-1,N-1} \end{bmatrix} \tag{2.13}$$

Thus, we can generalize the equations

$$a_{00} = l_{00} \tag{2.14}$$
$$a_{i,i-1} = l_{i,i-1} \tag{2.15}$$
$$a_{i,i} = l_{i,i-1}u_{i-1,i} + l_{i,i} \tag{2.16}$$
$$a_{i-1,i} = l_{i-1,i-1}u_{i-1,i} \tag{2.17}$$

for $i \in [1, N-1]$.

Once we decompose the tridiagonal matrix A, we can then solve it by methods given in [5] (page 159) and [6] (page 321). The code is given in Section 2.3.4.

2.3.1 Solution of a Matrix Equation in Python

In python, a matrix equation can be solved using different functions. Here I will consider a matrix equation,

$$AX = B$$

As throughout the project, I am dealing with tridiagonal matrices, here I am testing speed for tridiagonal matrix only.

Let's write the matrices A and B as a *list*.

[1]:
```
al, bt = -2, 5 -1j
N = 12
A = [[0 for j in range(N)] for i in range(N)]
B = [2,8,3,4,3,8,3,0,6,2,3,5]
```

```
A[0][0], A[N-1][N-1] = 1, 1
for j in range(1, N-1):
    A[j][j-1], A[j][j], A[j][j+1] = al, bt, al
display(A, B)
```

```
[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [-2, (5-1j), -2, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, -2, (5-1j), -2, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, -2, (5-1j), -2, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, -2, (5-1j), -2, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, -2, (5-1j), -2, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, -2, (5-1j), -2, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, -2, (5-1j), -2, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, -2, (5-1j), -2, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, -2, (5-1j), -2, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, -2, (5-1j), -2],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]
```

```
[2, 8, 3, 4, 3, 8, 3, 0, 6, 2, 3, 5]
```

In most of the cases, the process of finding X is by the equation,

$$X = A^{-1}B$$

Now, in python we have different ways available for finding inverse of a matrix.

Here the matrix equation is solved using different methods and the speed of computation is also checked.

2.3.2 Using numpy.linalg

```
[2]: import numpy as np
     import numpy.linalg as npLA

     Aarr, Barr = np.array(A), np.array(B)
     npLA.inv(A) @ B
```

```
[2]: array([2.        +4.30252080e-16j, 3.07246374+1.33729704e+00j,
            2.34980786+1.80701073e+00j, 2.20556128+2.00532585e+00j,
            2.16675826+2.10352326e+00j, 2.76309599+2.17010316e+00j,
            1.82603331+1.94018666e+00j, 1.2720806 +1.76734683e+00j,
            2.23784162+1.84214011e+00j, 2.2435935 +1.71908264e+00j,
            3.23068345+1.33376975e+00j, 5.        +0.00000000e+00j])
```

```
[3]: npLA.solve(Aarr, Barr)
```

```
[3]: array([2.        +4.44089210e-16j, 3.07246374+1.33729704e+00j,
            2.34980786+1.80701073e+00j, 2.20556128+2.00532585e+00j,
            2.16675826+2.10352326e+00j, 2.76309599+2.17010316e+00j,
            1.82603331+1.94018666e+00j, 1.2720806 +1.76734683e+00j,
            2.23784162+1.84214011e+00j, 2.2435935 +1.71908264e+00j,
            3.23068345+1.33376975e+00j, 5.        +0.00000000e+00j])
```

2.3.3 Using scipy.linalg

```
[4]: import scipy.linalg as splA

     splA.inv(A) @ B
```

```
[4]: array([2.        +3.11352320e-17j, 3.07246374+1.33729704e+00j,
            2.34980786+1.80701073e+00j, 2.20556128+2.00532585e+00j,
            2.16675826+2.10352326e+00j, 2.76309599+2.17010316e+00j,
            1.82603331+1.94018666e+00j, 1.2720806 +1.76734683e+00j,
            2.23784162+1.84214011e+00j, 2.2435935 +1.71908264e+00j,
            3.23068345+1.33376975e+00j, 5.        +0.00000000e+00j])
```

2.3.4 By LU factorization

```
[5]: lu1, piv1 = splA.lu_factor(A)
     splA.lu_solve((lu1, piv1), B)
```

```
[5]: array([2.        +4.44089210e-16j, 3.07246374+1.33729704e+00j,
            2.34980786+1.80701073e+00j, 2.20556128+2.00532585e+00j,
            2.16675826+2.10352326e+00j, 2.76309599+2.17010316e+00j,
            1.82603331+1.94018666e+00j, 1.2720806 +1.76734683e+00j,
            2.23784162+1.84214011e+00j, 2.2435935 +1.71908264e+00j,
            3.23068345+1.33376975e+00j, 5.        +0.00000000e+00j])
```

```
[6]: lu2, piv2 = splA.lu_factor(A)
     splA.lu_solve((lu2, piv2), B)
```

```
[6]: array([2.        +4.44089210e-16j, 3.07246374+1.33729704e+00j,
            2.34980786+1.80701073e+00j, 2.20556128+2.00532585e+00j,
            2.16675826+2.10352326e+00j, 2.76309599+2.17010316e+00j,
            1.82603331+1.94018666e+00j, 1.2720806 +1.76734683e+00j,
            2.23784162+1.84214011e+00j, 2.2435935 +1.71908264e+00j,
            3.23068345+1.33376975e+00j, 5.        +0.00000000e+00j])
```

Now, I am going to define a function for LU factorization. This function is helpful for solution corresponding to tridiagonal matrices.

```
[7]: def LU_solve_tridiag(A, b):
         '''
         Function to solve matrix equation by LU factorization (efficient for␣
      ↪tridiagonal matrices)
         Function: CFtriag(A, b)
             ARGUMENTS
         A, b: matrices for for equation, AX = b
             RETURNS
         X: solution of the matrix equation
         '''
         A_arr, b_arr = np.array(A), np.array([b])
         ab = np.concatenate((A_arr, b_arr.T), axis=1)
         n = len(ab)
         l = [[0 for j in range(n)] for i in range(n)]
         u = [[0 for j in range(n)] for i in range(n)]
         z = [0 for i in range(n)]
```

```
    for i in range(n-1):
        l[i][i-1] = ab[i][i-1]
        l[i][i] = ab[i][i] -l[i][i-1]*u[i-1][i]
        u[i][i+1] = ab[i][i+1]/l[i][i]
        z[i] = (ab[i][n]-l[i][i-1]*z[i-1])/l[i][i]
    l[n-1][n-2] = ab[n-1][n-2]
    l[n-1][n-1] = ab[n-1][n-1] -l[n-1][n-2]*u[n-2][n-1]
    z[n-1] = (ab[n-1][n]-l[n-1][n-2]*z[n-2])/l[n-1][n-1]
    X = [0 for i in range(n)]
    X[n-1] = z[n-1]
    for i in range(n-2, -1, -1):
        X[i] = z[i] - u[i][i+1]*X[i+1]
    return X
```

[8]:
```
LU_solve_tridiag(A, B)
```

[8]:
```
[(2+0j),
 (3.0724637363898277+1.3372970379232836j),
 (2.349807859936212+1.8070107266132946j),
 (2.205561276757351+2.005325848641847j),
 (2.166758256278089+2.1035232566126476j),
 (2.7630959922441964+2.170103164750727j),
 (1.8260333067077656+1.9401866591420722j),
 (1.2720806040962533+1.7673468297505694j),
 (2.2378416184081527+1.8421401131862254j),
 (2.2435934985172405+1.7190826440109177j),
 (3.2306834498904067+1.3337697475824486j),
 (5+0j)]
```

[ ]:
```

```

2.4 Function for getting Wavefunctions

The following code was initially taken from [6]. But a lot of modifications are been done on the codes by me.

fn_TDSE_solve.py

```
import numpy as np
import numpy.linalg as npLA
import scipy.linalg as spLA
import matplotlib.pyplot as plt

def LU_solve_tridiag(A, b):
    '''
    Function to solve a matrix equation by LU factorization (efficient for␣
 ↪tridiagonal matrices)
        ARGUMENTS
    A, b: matrices for for equation, AX = b
        RETURNS
    X: solution of the matrix equation
    '''
    A_arr, b_arr = np.array(A), np.array([b])
    ab = np.concatenate((A_arr, b_arr.T), axis=1) # augmented matrix
    n = len(ab)
    l = [[0 for j in range(n)] for i in range(n)]
    u = [[0 for j in range(n)] for i in range(n)]
    z = [0 for i in range(n)]
    for i in range(n-1):
        l[i][i-1] = ab[i][i-1]
        l[i][i] = ab[i][i] -l[i][i-1]*u[i-1][i]
        u[i][i+1] = ab[i][i+1]/l[i][i]
        z[i] = (ab[i][n]-l[i][i-1]*z[i-1])/l[i][i]
    l[n-1][n-2] = ab[n-1][n-2]
    l[n-1][n-1] = ab[n-1][n-1] -l[n-1][n-2]*u[n-2][n-1]
    z[n-1] = (ab[n-1][n]-l[n-1][n-2]*z[n-2])/l[n-1][n-1]
    X = [0 for i in range(n)]
    X[n-1] = z[n-1]
    for i in range(n-2, -1, -1):
        X[i] = z[i] - u[i][i+1]*X[i+1]
    return X

# SOLVING THE TDSE MATRIX EQUATION
def TDSE_mat_solve(hbar_m, psiR, psiI, V, dx, dt, method_):
    '''
        ARGUMENTS
    hbar_m: [hbar, m]
    psiR: (array) real part of wavefunction
    psiI: (array) imaginary part of wavefunction
    V: Potential
    dx, dt: steps in x, steps in t
    method_: numpy.linalg.inv, scipy.linalg.lu_solve
        RETURNS
    psiR: (array) real part of wavefunction
    psiI: (array) imaginary part of wavefunction
    '''
    hbar, m = hbar_m
    Nx = len(psiR)
    psicomp0 = [0 for i in range(Nx)]
    for i in range(Nx):
        psicomp0[i] = complex(psiR[i], psiI[i])
    al, bt = hbar*dt/(4*m*dx**2), dt/(2*hbar)
    gam = [2*al + bt*V[i] for i in range(Nx)]
    A = [[0 for j in range(Nx)] for i in range(Nx)]
    A[0][0], A[Nx-1][Nx-1] = 1, 1
    B = [[0 for j in range(Nx)] for i in range(Nx)]
    B[0][0], B[Nx-1][Nx-1] = 1, 1
    for j in range(1, Nx-1):
        A[j][j-1], A[j][j], A[j][j+1] = -al, gam[j]-1j, -al
        B[j][j-1], B[j][j], B[j][j+1] = al, -gam[j]-1j, al
    A, B, psicomp0 = np.array(A), np.array(B), np.array(psicomp0)
    b = B.dot(psicomp0)
    if method_ == 'numpy.linalg.inv':
        psicomp = (npLA.inv(A)).dot(b)
    elif method_ == 'numpy.linalg.solve':
        psicomp = npLA.solve(A, b)
    elif method_ == 'scipy.linalg.lu_solve':
        lu1, piv1 = spLA.lu_factor(A)
        psicomp = spLA.lu_solve((lu1, piv1), b)
    elif method_ == 'LU_solve_def':
        psicomp = LU_solve_tridiag(A, b)
    psiR = np.array([psicomp[i].real for i in range(Nx)])
    psiI = np.array([psicomp[i].imag for i in range(Nx)])
    return psiR, psiI

# NORMALIZATION
def psiNormRI(psiR, psiI, dx):
    '''
        ARGUMENTS
    psiR: list or array of real psi values
    psiI: list or array of imaginary psi values
    dx: step in x
        RETURNS
    psiR: normalized list
    psiI: normalized list
    '''
    Nx = len(psiR)
    psiR, psiI = np.array(psiR), np.array(psiI)
    psimod2 = psiR**2 + psiI**2
    psiNorm1 = (np.sum(psimod2)*dx)**0.5
    psiR = [psiR[i]/psiNorm1 for i in range(Nx)]
    psiI = [psiI[i]/psiNorm1 for i in range(Nx)]
    return psiR, psiI

# ULTIMATE FUNCTION
def TDSE_time_evolution(hbar_m, wvfn, prwv, pot, prpt, X, iters_solve, T_max,␣
 ↪pause_time, method_, return_=False, plot_=False):
    '''
        ARGUMENTS
    hbar_m: parameters [hbar, m]
    wvfn: initial wavefunction wvfn(prwv, X)
    prwv: parameters for wvfn
    pot: function for potential pot_fn(prpt, x)
    prpt: parameters for pot_fn
    X: the x array
    iters_solve: No. of iterations interval for real time plot
    T_max: max (final) time
    pause_time: Pause time plt.pause(tps)
    method_: numpy.linalg.inv, np.linalg.solve, scipy.linalg.lu_solve, LU_solve_
 ↪def
    return_: True if the output is required
    plot_: True if plots are required
        RETURNS
    t_arr, E, psiR_arr, psiI_arr, psimd2_arr if return_=True
    None if return_ is not given
    (Wave function plot) if plot_=True
    '''
    hbar, m = hbar_m
    Nx = len(X)
    x0, xN = X[0], X[Nx-1]
    dx = (xN-x0)/(Nx-1)
    E, wvfnx = wvfn(prwv, X)
    V = pot(prpt, X)
    Vmx = max(max(V), abs(min(V)))
    dt = hbar/(hbar**2/(m*dx**2) + Vmx/2)    # not necessary
    psi = wvfnx      # initial
    psiR = [psi[i].real for i in range(Nx)]   # initial real part
    psiI = [psi[i].imag for i in range(Nx)]   # initial imaginary part
    psiR, psiI = psiNormRI(psiR, psiI, dx)
    t_arr, psiR_arr, psiI_arr, psimd2_arr = [], [], [], []
    Xmn, Xmx, Ymx = min(X), max(X), 1.5*max(np.abs(np.array(psiR)))

    # Rescaling
    if plot_ == True:
        if Vmx != 0:
            Efac = Ymx/(2*Vmx)
            Vplot = [V[i]*Efac for i in range(Nx)]
        print(f'Energy of the particle = {E}, Scaled Energy = {E*Efac}')
    t, itr, plti = 0, 1, 1
    while t <= T_max:
        if itr % iters_solve == 0:  # plotting at ts no. of iterations
            psimd2 = [psiR[i]**2+psiI[i]**2 for i in range(Nx)]
            if return_ == True:
                t_arr.append(t)
                psiR_arr.append(psiR), psiI_arr.append(psiI)
                psimd2_arr.append(psimd2)
            if plot_ == True:
                plt.axis([Xmn, Xmx, -Ymx, Ymx])
                if Efac != 0:
                    plt.plot(X, Vplot, ':k')
                    plt.axhline(E*Efac, label=f'E={E:.3}, E_plot={(E*Efac):.3}')
                    plt.fill_between(X, Vplot, facecolor='grey')
                plt.plot(X, psiR, label=r'$\psi_{real}(x,t)$')
                # print(f'Normalization check: {np.sum(psimd2)*dx}')
                plt.plot(X, psimd2, label=r'$|\psi(x,t)|^2$')
                plt.text(0.1*Xmx, -0.9*Ymx, f't = {t}')
                plt.legend(loc='upper right')
                plt.xlabel('$x$')
                # plt.savefig(f'plots/plot_{plti}')
                plt.pause(pause_time)
                plt.clf()
            plti += 1
        psiR, psiI = TDSE_mat_solve(hbar_m, psiR, psiI, V, dx, dt, method_)
        itr += 1
        t += dt
    if return_ == True:
        return np.array(t_arr), E, np.array(psiR_arr), np.array(psiI_arr), np.
↪array(psimd2_arr)
    if plot_ == True:
        plt.show()
```

Chapter 3

Analysis of 1D Bound State Problems

3.1 Introduction

For analysis of a particle in a bound state, the time independent Schrödinger equation is to be solved.

The time independent Schrödinger equation in 1D is,

$$-\frac{\hbar^2}{2m}\frac{\partial \psi}{\partial x}\psi(x) + V(x)\psi(x) = E\psi(x) \tag{3.1}$$

This equation can be solved in detail by *Numerov method*. But an easier method is used for bound state problems. For bound states the energies are discrete and in 1D for each energy state we get a unique wavefunction (i.e. no degeneracy). Now, by knowing the potential, we get the Hamiltonian ($\mathcal{H}$). For n^{th} state (the corresponding energy and wavefunction are E_n and $\phi_n(x)$ respectively), we can write the time independent Schrödinger equation,

$$\mathcal{H}\phi_n(x) = E_n\phi_n(x) \tag{3.2}$$

So it's an eigenvalue equation. Thus we can get all eigenvalues (E_n) and corresponding eigenvectors ($\phi_n(x)$) by knowing the Hamiltonian ($\mathcal{H}$) (as a matrix) only (this concept is applied in Section 3.2. But this approach can only be used for bound state problems, otherwise for scattering problems this method calculates eigenvalues and eigenvectors of the Hamiltonian and gives us some wrong results. For the scattering problems we can use the time dependent Schrödinger equation (discussed in detail in Chapter 5).

Here I am going to solve the bound state problem for **linear harmonic oscillator**. For large energy values, we can get a bound state inside linear harmonic potential. Thus it's safe to choose this potential for a method that is applicable to bound state problems. The numerical solution is shown in Section 3.2.

I am going to show the correctness of this method (finding eigenvalues and eigenvectors) by verifying the energy values (in Chapter 4) and uncertainties (in Section 4.5). Once, we get the eigenstates, we can express any function as a linear superposition of of those eigenstates. Superposing various eigenstates of harmonic oscillator is done in Section 4.6. Once eigenstates are obtained, I have taken a superposed state and shown the time evolution for those states. From these analyses, I have explained the concepts of *stationary states* in Section 4.7.

3.2 Numerical Soplution for Bound State Problems

To have numerical solutions for Eq. (3.1), we need the finite difference methods.

According to finite difference formulae [3] a second order derivative can be written as,

$$\left(\frac{\partial^2\psi}{\partial x^2}\right)_{\mathrm{i}} = \frac{\psi_{\mathrm{i+1}} - 2\psi_{\mathrm{i}} + \psi_{\mathrm{i-1}}}{\delta x^2} + \mathcal{O}(\delta x^2)$$

Inserting this formula into Eq. (3.1) (for ith index of position) we would get,

$$-\frac{\hbar^2}{2m}\frac{\psi_{\mathrm{i+1}} - 2\psi_{\mathrm{i}} + \psi_{\mathrm{i-1}}}{\delta x^2} + V(x_{\mathrm{i}})\psi_{\mathrm{i}} = E\psi_{\mathrm{i}} \tag{3.3}$$

for $\mathrm{i} = 1, 2, \ldots, N_x - 2$. The values of ψ_0 and ψ_{N_x-1} are the limiting values and beyond this, we have no values for ψ. By applying Eq. (3.3) for 0^{th} and $N_x - 1^{th}$ indices,

$$-\frac{\hbar^2}{2m}\frac{\psi_1 - 2\psi_0 + \psi_{-1}}{\delta x^2} + V(x_0)\psi_0 = E\psi_0$$
$$-\frac{\hbar^2}{2m}\frac{\psi_{N_x} - 2\psi_{N_x-1} + \psi_{N_x-2}}{\delta x^2} + V(x_{N_x-1})\psi_{N_x-1} = E\psi_{N_x-1}$$

The values of ψ_{-1} and ψ_{N_x} are obtained actually from boundary conditions at x_0 and x_{N_x-1}. But here, these are values which are out of index (as $\mathrm{i} \in [0, N_x - 1]$). So here I have taken, $\psi_{-1} = 0$ and $\psi_{N_x} = 0$. So,

$$-\frac{\hbar^2}{2m}\frac{\psi_1 - 2\psi_0}{\delta x^2} + V(x_0)\psi_0 = E\psi_0 \tag{3.4}$$
$$-\frac{\hbar^2}{2m}\frac{-2\psi_{N_x-1} + \psi_{N_x-2}}{\delta x^2} + V(x_{N_x-1})\psi_{N_x-1} = E\psi_{N_x-1} \tag{3.5}$$

This is a further insertion on the problem. This makes the consideration of an infinite jump in potential at at x_0 and x_{N_x-1}. The result of this effect can be observed at the output plots of Section 5.2 codes. There we can observe a total reflection of wavepackets at the boundaries which doesn't exists in physical systems.

We can summarize Eq. (3.3) (for $\mathrm{i} = 1, 2, \ldots, N_x - 2$) and Eq. (3.4) into a matrix equation,

$$\mathcal{H}\psi = E\psi$$

Here $\mathcal{H}$ is a $(N_x \times N_x)$ square matrix and ψ is a $(N_x \times 1)$ column matrix. Until now, this relation can be applied for any ψ. Here for bound state problems the above equation is actually an eigenvalue equation. And this can be written as,

$$\mathcal{H}\phi_n = E_n\phi_n \tag{3.6}$$

Here E_n and ϕ_n are eigenvalues and eigenvectors of matrix $\mathcal{H}$. This matrix can be simplified as,

$$\mathcal{H} = T + V \tag{3.7}$$

where,

$$T = -\frac{\hbar^2}{2m}D$$

where, we define D as,

$$D = \frac{1}{\delta x^2}\begin{bmatrix} -2 & 1 & 0 & \ldots & 0 \\ 1 & -2 & 1 & \ldots & 0 \\ 0 & 1 & -2 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & -2 \end{bmatrix}$$

and

$$V = \begin{bmatrix} V(x_0) & 0 & 0 & \dots & 0 \\ 0 & V(x_1) & 0 & \dots & 0 \\ 0 & 0 & V(x_2) & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & V(x_{N_x-1}) \end{bmatrix}$$

This method is applied in [7]

In Python, we have several functions (`scipy.linalg.eigh`, `scipy.sparse.linalg.eigsh`, etc.) for calculating eigenvalues and eigenvectors of a matrix. These functions give a set of eigenvalues and eigenvectors as output. Thus, all we need to do is the formation of $\mathcal{H} = T + V$ matrix. Now, I am going to discuss the 1D quantum linear harmonic oscillator problem in Chapter 4.

Chapter 4

Linear Harmonic Oscillator

4.1 Introduction

The potential for 1D linear harmonic oscillator is $V(x) = \frac{1}{2}m\omega^2x^2$. So, from Eq. (3.1)

$$-\frac{\hbar^2}{2m}\frac{\partial\psi}{\partial x}\psi(x) + \frac{1}{2}m\omega^2x^2\psi(x) = E\psi(x) \tag{4.1}$$

The detailed analytical solution for 1D Quantum Linear Harmonic Oscillator is given in [1] (page 239).

To verify the obtained energy and uncertainty values with the exact values, here I have imported the physical constants from `scipy.constants` [14].

4.2 Computations Using Exact Numerical Values

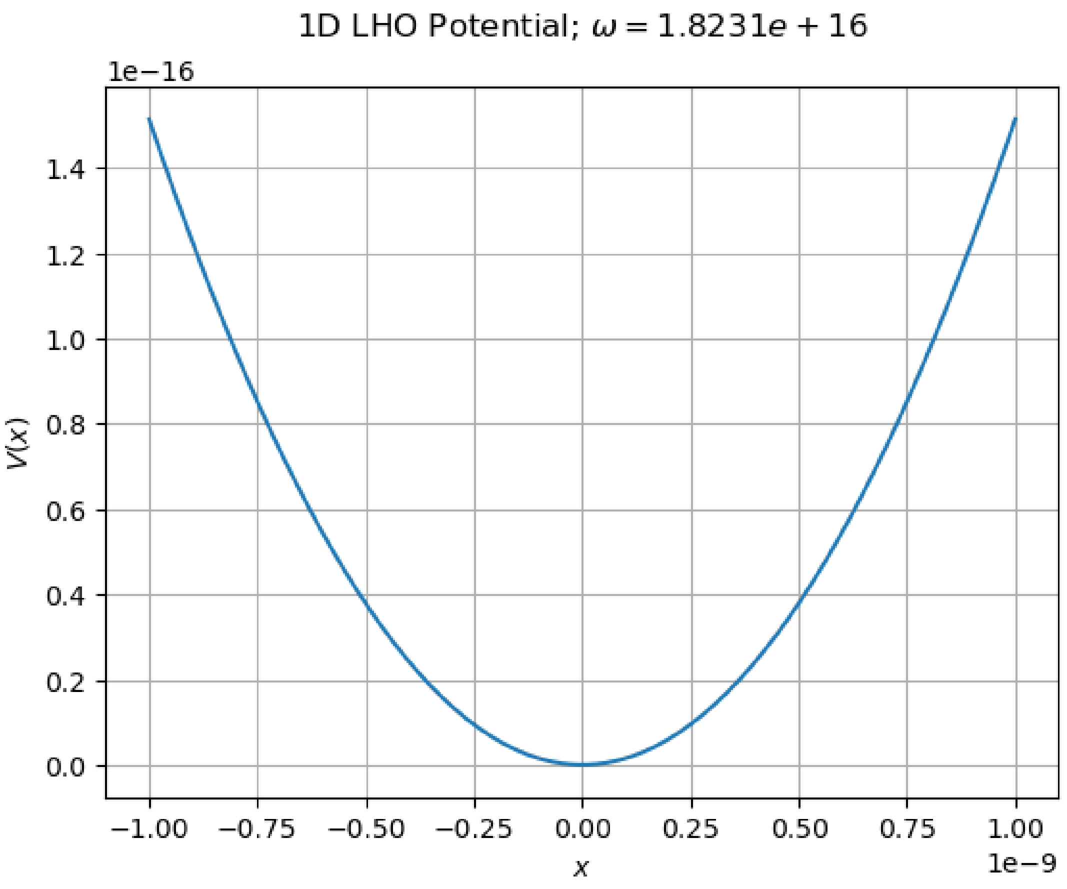
Till now, I have taken all physical constants in atomic units. Now, I am going to consider the exact values to verify the obtained results with the original values.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eigh
import scipy.constants as const
```

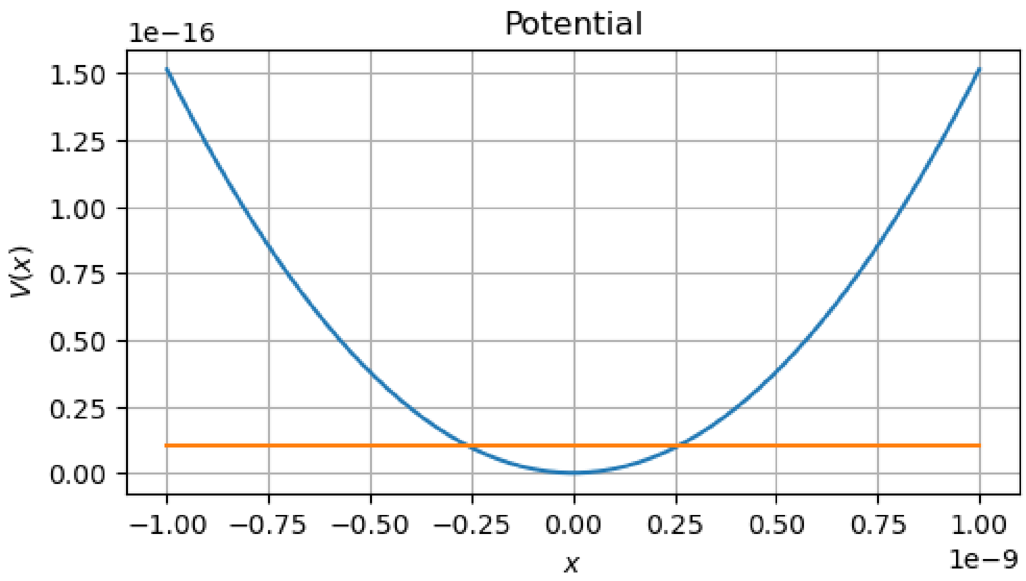
[2]:
```
hcut = const.hbar # INPUT
mass = const.electron_mass   # INPUT
```

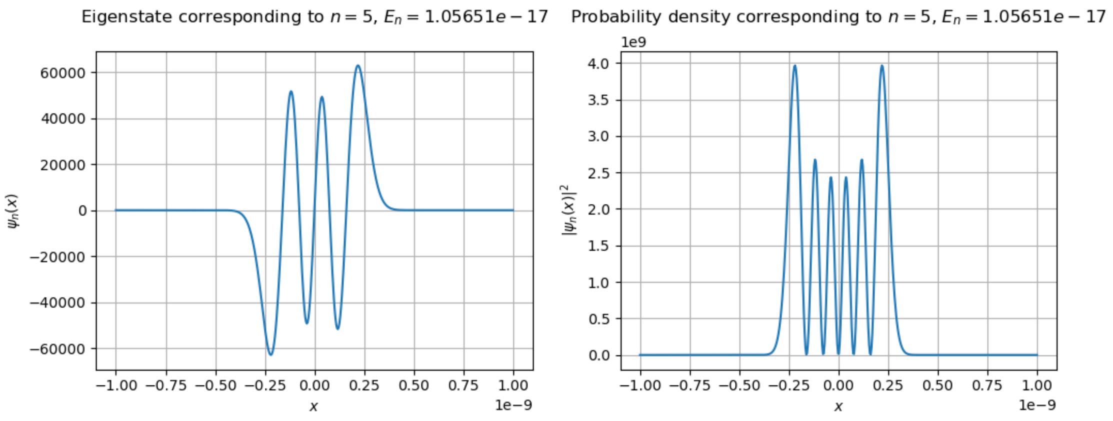
[3]:
```
L = 10     # INPUT
L = 10e-10   # INPUT
x_min, x_max = -L, L
Nx = 500     # no. of grid points
x = np.linspace(x_min, x_max, Nx)    # x axis

# potential
def potential1d(x, omega):
    return 0.5*mass*(omega**2)* x**2

omega1 = 1.0 # INPUT
```

```
omega1 = 12*const.electron_volt/const.hbar   # INPUT

Vx = potential1d(x=x, omega=omega1)

plt.plot(x, Vx)
plt.title(f'1D LHO Potential; $\omega={omega1:.5}$\n')
plt.xlabel('$x$')
plt.ylabel('$V(x)$')
plt.grid()
plt.show()
```



[4]:
```
dx = x[1] - x[0]
D_mat = (np.diag(-2*np.ones(Nx)) + np.diag(np.ones(Nx-1), 1)
                        + np.diag(np.ones(Nx-1), -1))/dx**2
T_mat = (-hcut**2/(2*mass)) * D_mat
V_mat = np.diag(Vx*np.ones(Nx))
H_mat = T_mat + V_mat
eigenvals1, eigenvecsT1 = eigh(H_mat)

def eig_val(n):
    return eigenvals1[n]
def eig_vec(n):
    eigf = eigenvecsT1[:, n]
    eigf = eigf/np.sum(np.abs(eigf)**2*dx)**0.5
    return eigf
def prob_den(n):
    eigf = eigenvecsT1[:, n]
    eigfm2 = eigf.conjugate()*eigf
    eigf = eigf/np.sum(np.abs(eigf)**2*dx)**0.5
    eigfm2 = eigf.conjugate()*eigf
    return eigfm2

n = 5   # INPUT

plt.figure(figsize=(6,3))
plt.plot(x, Vx)
plt.plot(x, eig_val(n)*np.ones(Nx))
plt.title('Potential')
plt.xlabel('$x$')
plt.ylabel('$V(x)$')
plt.grid()
plt.show()

plt.figure(figsize=(12,4))
plt.subplot(121)
plt.plot(x, eig_vec(n))
plt.title(f'Eigenstate corresponding to $n={n}$, $E_n={eig_val(n):.6}$\n')
plt.xlabel('$x$')
plt.ylabel(f'$\psi_n(x)$')
plt.grid()
plt.subplot(122)
plt.plot(x, prob_den(n))
plt.title(f'Probability density corresponding to $n={n}$, $E_n={eig_val(n):.
 ↪6}$\n')
plt.xlabel('$x$')
plt.ylabel(f'$|\psi_n(x)|^2$')
plt.grid()
plt.show()
```





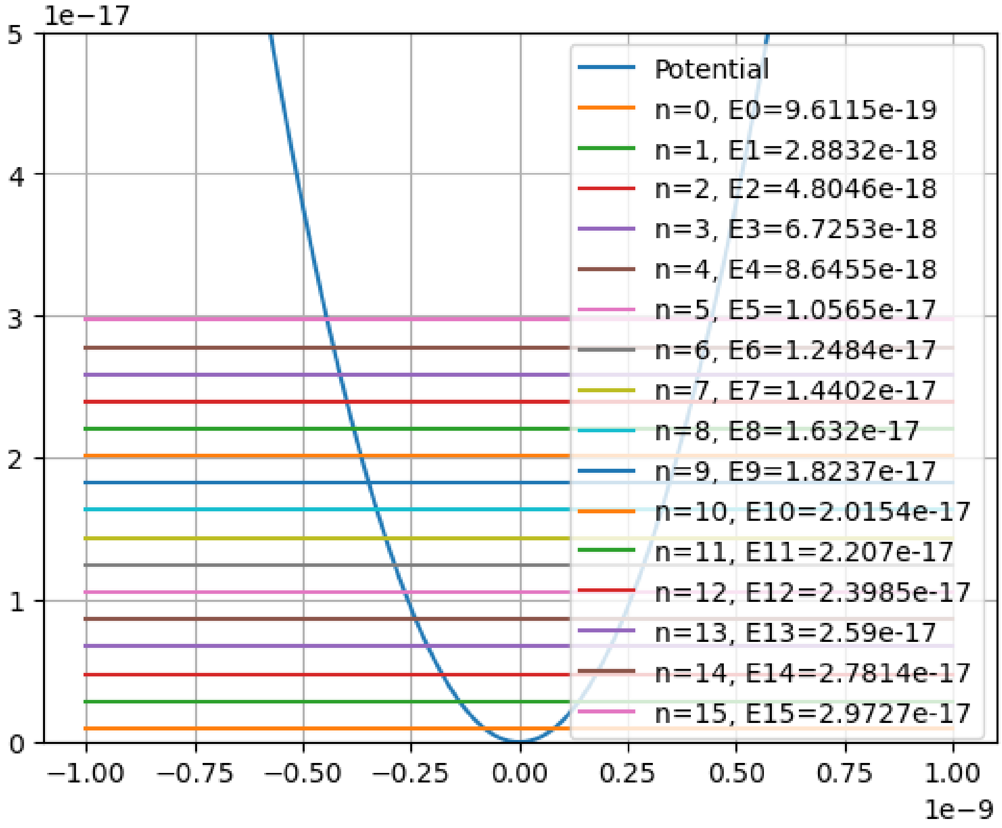
Verification with exact value:

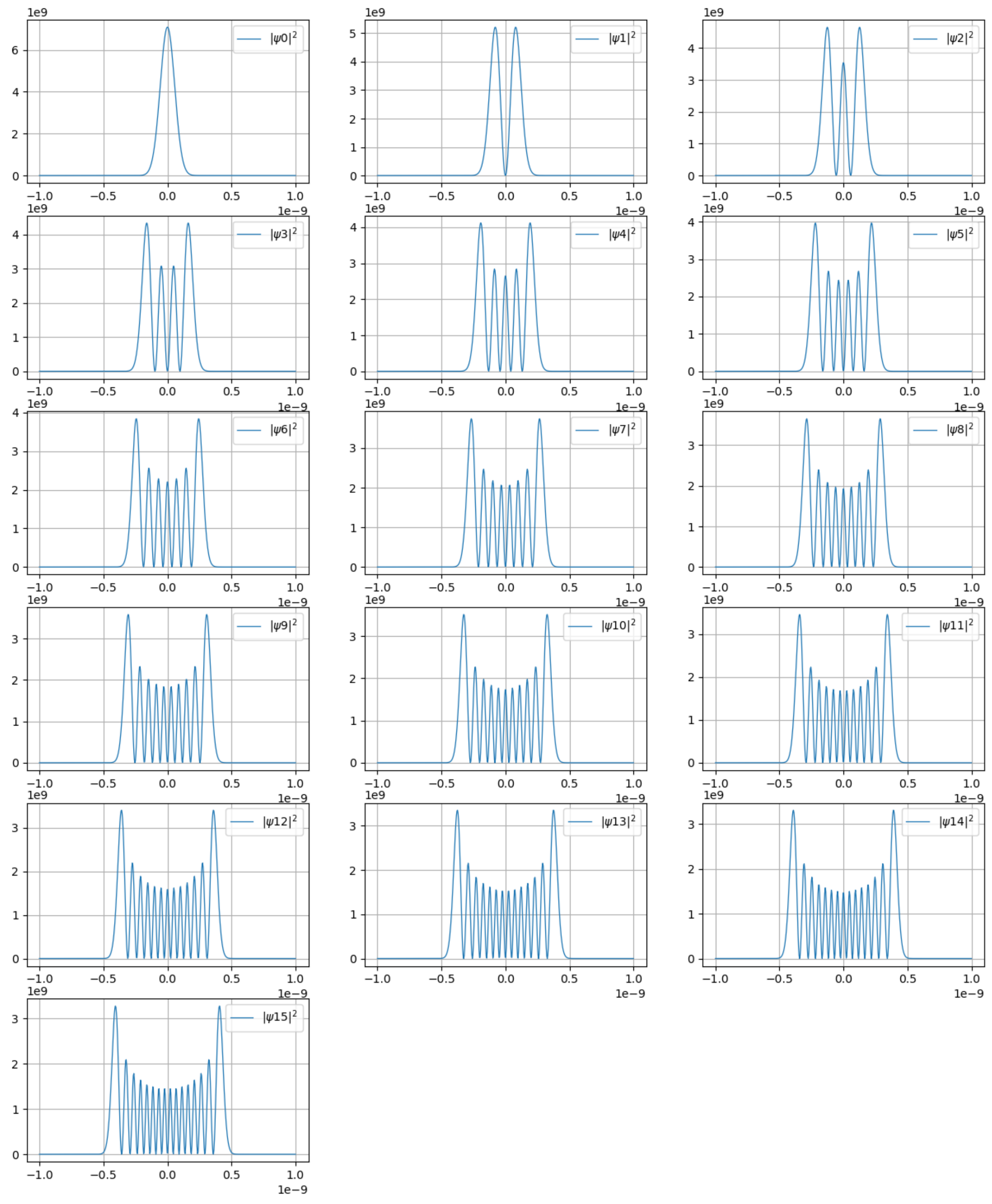
```
[5]: (5 + 1/2)*hcut*omega1    # Exact Value of energy
```

```
[5]: 1.05743657844e-17
```

4.3 Different Eigenstates in a Single Plot

```
[6]: nmin, nmax = 0, 15   # INPUT

plt.plot(x, Vx, label='Potential')
for i in range(nmin, nmax+1):
    plt.plot(x, eig_val(i)*np.ones(Nx), label=f'n={i}, E{i}={eig_val(i):.5}')
plt.legend(loc='best')
plt.ylim(0, 0.5e-16)
plt.grid()
plt.show()

plt.figure(figsize=(15, 18)) # INPUT
for i in range(nmin, nmax+1):
    plt.subplot((nmax+1-nmin)//3 +1, 3, i-nmin+1)
    # plt.plot(x, eig_vec(i), label=f'$\psi {i}$', lw=0.7) # for wavefunction
    plt.plot(x, prob_den(i), label=f'$|\psi {i}|^2$', lw=0.9) # probability␣
 ↪density
    plt.legend(loc='best')
    plt.grid()
plt.show()
```


1e−17
5
4
3
2
1
0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1e−9
Potential
n=0, E0=9.6115e-19
n=1, E1=2.8832e-18
n=2, E2=4.8046e-18
n=3, E3=6.7253e-18
n=4, E4=8.6455e-18
n=5, E5=1.0565e-17
n=6, E6=1.2484e-17
n=7, E7=1.4402e-17
n=8, E8=1.632e-17
n=9, E9=1.8237e-17
n=10, E10=2.0154e-17
n=11, E11=2.207e-17
n=12, E12=2.3985e-17
n=13, E13=2.59e-17
n=14, E14=2.7814e-17
n=15, E15=2.9727e-17



4.4 Momentum space

If we have a known wavefunction in position space, we get that in momentum space by using *Fourier Transformations.*

In position space, the wavefunction can be written as,

$$\psi(x) = \frac{1}{\sqrt{2\pi\hbar}} \int a(p) \exp(\frac{ipx}{\hbar})\, dp$$

So, by Fourier transform,

$$a(p) = \frac{1}{\sqrt{2\pi\hbar}} \int \psi(x) \exp(-\frac{ipx}{\hbar})\, dx$$

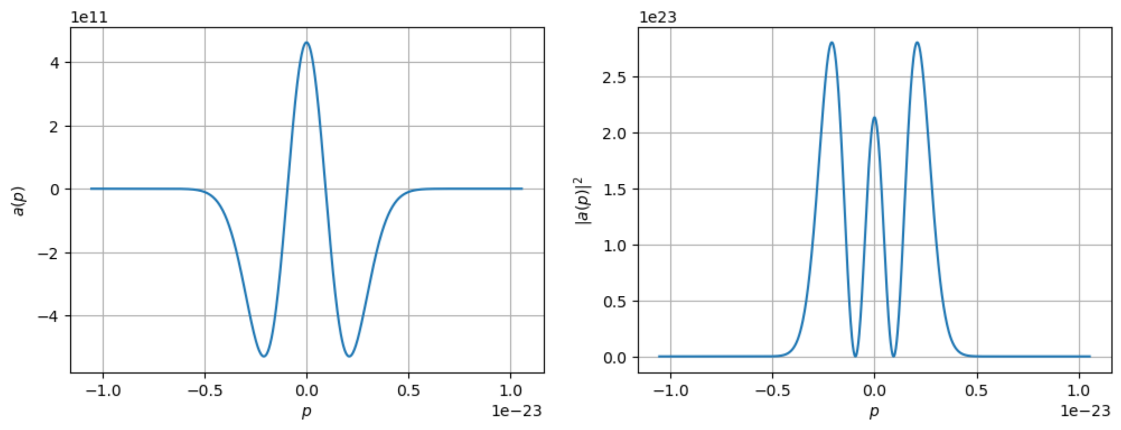
$a(p) is the wave function in Momentum space.$

[7]:
```
k = np.linspace(100/x_min, 100/x_max, Nx*2) # INPUT - same kind of limit that of␣
↪1/x
p = hcut*k
dp = p[1]-p[0]

def eig_vec_pspace(p, psi):
    ap = []
    for pi in p:
        api = (1/np.sqrt(2*np.pi*hcut))*np.sum(psi*np.exp(-1j*pi*x/hcut)*dx)
        ap.append(api)
    return np.array(ap)

def prob_den_pspace(p, psi):
    return np.abs(eig_vec_pspace(p, psi))**2

def eig_vec_return(p, psi):
    psix = []
    for xi in x:
        psixi = (1/np.sqrt(2*np.pi*hcut))*np.sum(eig_vec_pspace(p, psi)*np.
↪exp(1j*p*xi/hcut)*dp)
        psix.append(psixi)
    return np.array(psix)

def prob_den_return(p, psi):
    return np.abs(eig_vec_return(p, psi))**2
```

Momentum distribution for ϕ_2 state:

[8]:
```
psi = eig_vec(2) # INPUT

plt.figure(figsize=(12, 4))
plt.subplot(121)
plt.plot(p, eig_vec_pspace(p, psi))
plt.xlabel('$p$')
plt.ylabel('$a(p)$')
plt.grid()
plt.subplot(122)
plt.plot(p, prob_den_pspace(p, psi))
plt.xlabel('$p$')
plt.ylabel('$|a(p)|^2$')
plt.grid()
plt.show()
```

```
c:\ProgramData\Anaconda3\lib\site-packages\matplotlib\cbook\__init__.py:1298:
ComplexWarning: Casting complex values to real discards the imaginary part
  return np.asarray(x, float)
```



4.5 Expectation values and Uncertainty

[9]:
```
psi = eig_vec(2) # INPUT

def expectation_val_x(psi):
    return np.sum(x*np.abs(psi)**2*dx)
def expectation_val_x2(psi):
    return np.sum(x**2*np.abs(psi)**2*dx)
def uncertainty_x(psi):
    return np.sqrt(expectation_val_x2(psi)-(expectation_val_x(psi)**2))
def expectation_val_p(psi):
    return np.sum(p*prob_den_pspace(p, psi)*dp)
def expectation_val_p2(psi):
    return np.sum(p**2*prob_den_pspace(p, psi)*dp)
def uncertainty_p(psi):
    return np.sqrt(expectation_val_p2(psi)-(expectation_val_p(psi)**2))
def uncertainty_xp(psi):
    return uncertainty_x(psi)*uncertainty_p(psi)

print(uncertainty_xp(psi))
```

```
2.6364295538588518e-34
```

Verification with exact value:

[10]:
```
(2 + 1/2)*hcut   # exact value of uncertainty (for n=2 state of LHO)
```

[10]:
```
2.636429544115391e-34
```

[ ]:
```

```

The formula for uncertainty of an eigenstate of linear harmonic oscillator is taken from [1] (page 249).

4.6 Superposition of the Eigenstates

During the analysis, I am going to use values of $\hbar$ and m in Hartree atomic units [13] to make the numerical calculations more simplified.

lho_fns.py

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

def phi_n(pr, X):
    '''
        ARGUMENTS
    pr: [hbar, m]
    X: X array
        RETURNS
    En: Energy of eigenstate
    eigf: eigenstate as an array
    '''
    hbar_m, omega, n = pr
    hbar, m = hbar_m
    Vx = 0.5*m*(omega**2)* X**2
    Nx = len(X)
    dx = X[1] - X[0]
    D_mat = (np.diag(-2*np.ones(Nx)) + np.diag(np.ones(Nx-1), 1)
                         + np.diag(np.ones(Nx-1), -1))/dx**2
    T_mat = (-hbar**2/(2*m)) * D_mat
    V_mat = np.diag(Vx*np.ones(Nx))
    H_mat = T_mat + V_mat
    eigenvals1, eigenvecsT1 = eigh(H_mat)
    En = eigenvals1[n]
    eigf = eigenvecsT1[:, n]
    eigf = eigf/np.sum(np.abs(eigf)**2*dx)**0.5
    return En, eigf

def pot_lho(pr, X):
    m, omega = pr
    return (1/2)*m*omega*np.array(X)**2

def psi_superposed(pr, X):
    '''
        ARGUMENTS
    pr: [[hbar, m], omega, n_vals, c_vals] for psi = sum(c*phi_n)
    X: X array
        RETURNS
    Es: Energy of superposed state
    psis: Superposed state as an array
    '''
    hbar_m, omega, n_vals, c_vals = pr
    ns, cs = n_vals, np.array(c_vals)
    cs = cs/np.sum(cs**2)**0.5
    Es = 0
    psis = np.array([0 for i in range(len(X))], dtype=float)
    for i in range(len(ns)):
        prphi = hbar_m, omega, ns[i]
        Ei, psii = phi_n(prphi, X)
        Es += (cs[i])**2 *Ei
        psii = cs[i]*psii
        psis += psii
    return Es, psis
```

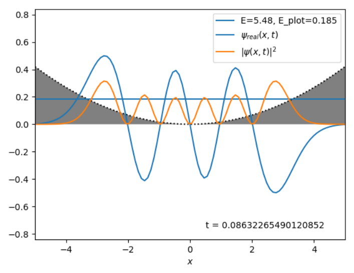
presentation_bndst.py

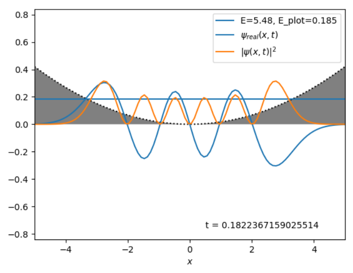
```
import numpy as np
from gaussn_prop import TDSE_time_evolution
from lho_fns import phi_n, pot_lho, psi_superposed

hbar, m = 1, 1
hbar_m = [hbar, m]

X = np.linspace(-5, 5, 100)
omega = 1
hbar_m = [hbar, m]
prpt = m, omega

def bndst_evolution(plot_):
    if plot_ == 'eigenstate':
        n = 5        # INPUT
        prphi = [hbar_m, omega, n]
        iters_solve, T_max, pause_time = 10, 6, 0.01
        TDSE_time_evolution(hbar_m, phi_n, prphi, pot_lho, prpt, X,
                iters_solve, T_max, pause_time, method_='LU_solve_def', plot_
 ↪=True)
    elif plot_ == 'superposed state':
        ns = [2,3,4,5,6,7]   # INPUT
        cs = [10, 6, 4, 5, 4, 7]    # INPUT
        prpsi = [hbar_m, omega, ns, cs]
        iters_solve, T_max, pause_time = 10, 20, 0.01
        TDSE_time_evolution(hbar_m, psi_superposed, prpsi, pot_lho, prpt, X,
                iters_solve, T_max, pause_time, method_='LU_solve_def', plot_
 ↪=True)

bndst_evolution(plot_='eigenstate')
# plot_: eigenstate, superposed state
```

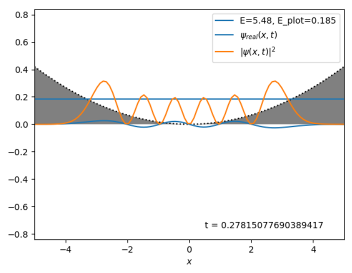
For this code, we would have the following outputs,

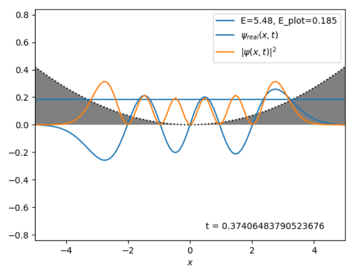
- If we choose `plot_='eigenstate'`: We need to give input for `n` (here `n = 5` is given and ϕ_5

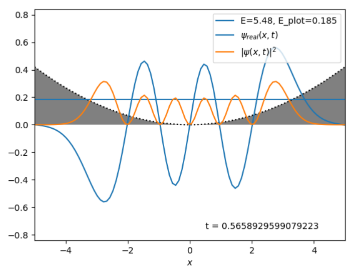
is plotted with its time evolution),

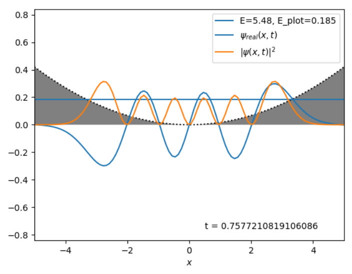


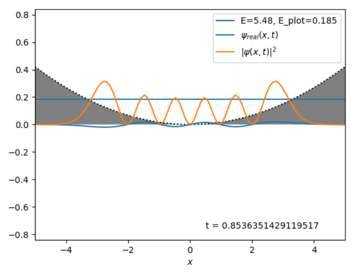


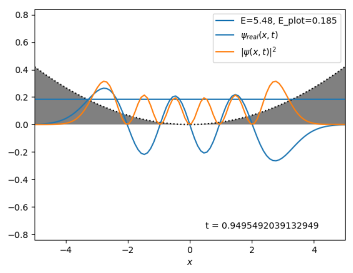


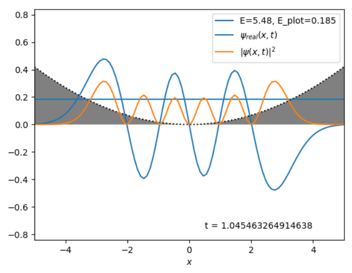






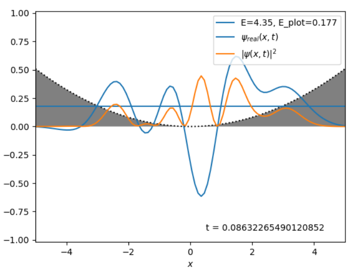


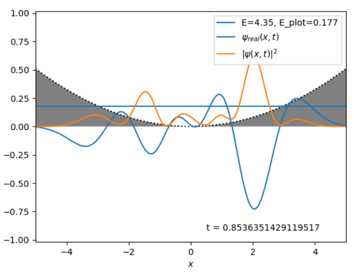


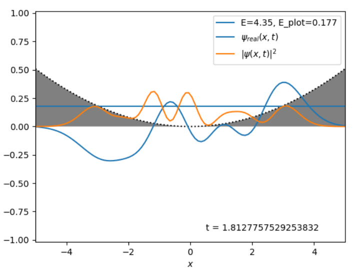


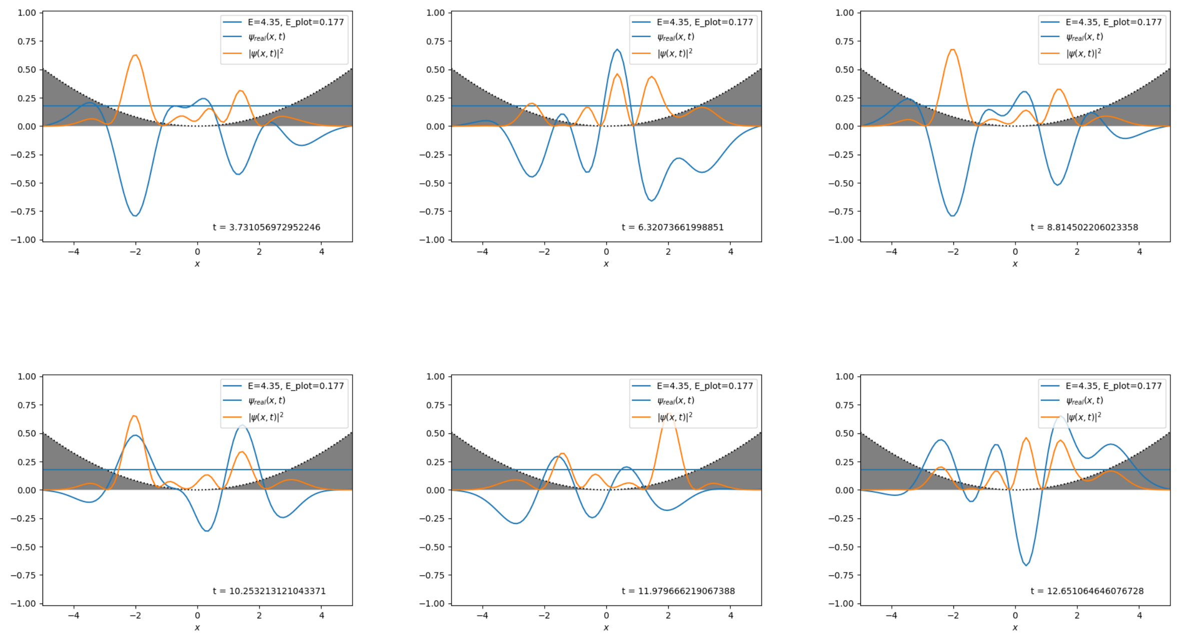
Click to view the plots.

- If we choose `plot_='eigenstate'`: We need to give input for `ns` and `cs` as a list (here `ns = [2,3,4,5,6,7]` and `cs = [10, 6, 4, 5, 4, 7]` are given and for these inputs the superposed state $\psi = 10\phi_2 + 6\phi_3 + 4\phi_4 + 5\phi_5 + 4\phi_6 + 7\phi_7$ is plotted with its time evolution),









Click to view the plots.

The animations can be found at `https://github.com/suman122003/BSc_Physics_Codes/blob/main/DSE_4_Computational_QM_project`.

4.7 Stationary States

For a time dependent energy eigenstate $\phi_n(x,t)$, we can write the time dependent Schrödinger equation,

$$\begin{aligned}\mathcal{H}\phi_n(x,t) &= i\hbar\frac{\partial\phi_n(x,t)}{\partial t}\\ \implies E_n\phi_n(x,t) &= i\hbar\frac{\partial\phi_n(x,t)}{\partial t}\\ \implies \int_{\phi_n(x,t=0)}^{\phi_n(x,t=t)}\frac{\delta\phi_n}{\phi_n} &= -\frac{iE_n}{\hbar}\int_{t=0}^{t=t}\delta t\end{aligned}$$

$$\implies \phi_n(x,t=t) = \phi_n(x,t=0)\exp\left(-\frac{iE_nt}{\hbar}\right) \tag{4.2}$$

So, the real part of the wavefunction has an oscillatory change. But, the probabilities density $|\phi_n(x,t)|^2 = \phi_n^*(x,t)\,\phi_n(x,t)$ remains constant with the time [1] (page 179). For its time independency, the energy eigenstates are called **stationary states**. The stationary state is shown in Section 4.6.

But a superposed state is not a stationary state, as we see the plot for the superposed state in Section 4.6. Actually, the time evolution of a superposed state is also oscillatory in nature. I have discussed it in the next section.

4.8 Oscillation of Superposed State

For a superposed state,

$$\psi = \sum_i c_i\phi_i \tag{4.3}$$

we can calculate expectation values of quantum number and energy by,

$$\langle n \rangle = \sum_i |c_i|^2 n_i \tag{4.4}$$

$$\langle E \rangle = \sum_i |c_i|^2 E_i \tag{4.5}$$

Now, comparing with the formula for energy eigenvalue $E_n = \left(n + \frac{1}{2}\right) \hbar\omega$, I have introduced a similar kind of expression for superposed energy (where the frequency is not ω,

$$\langle E \rangle = \left(\langle n \rangle + \frac{1}{2} \right) \hbar \omega' \tag{4.6}$$

$$\implies \omega' = \frac{\langle E \rangle}{\left(\langle n \rangle + \frac{1}{2} \right) \hbar} \tag{4.7}$$

Once we calculate ω', we can determine time period for oscillation of the probability density by $T' = \frac{2\pi}{\omega'}$.

All these things are calculated in the following Python codes:

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
from lho_fns import phi_n, psi_superposed
```

[3]:
```
ns = [2,3,4,5,6,7]   # INPUT
cs = [10, 6, 4, 5, 4, 7]    # INPUT
ns, cs = np.array(ns), np.array(cs)
cs = cs/np.sum(cs**2)**0.5

hbar, m = 1, 1
hbar_m = [hbar, m]
omega0 = 1
X = np.linspace(-5, 5, 100)
Es, psis = psi_superposed([hbar_m, omega0, ns, cs], X)

n_exp = np.sum(ns*cs**2)

print(f'<n>={n_exp} and <E>={Es}.')
```

```
<n>=3.8677685950413228 and <E>=4.353169610489678.
```

[4]:
```
omega1 = Es/(n_exp + 1/2)
T1 = 2*np.pi/omega1
print(T1)   # exact value (analytical)
```

```
6.304256878802618
```

[ ]:
```

```

From the observed plots and animations in Section 4.6, I have observed, the time period for oscillation of the probability density is nearly **6.32**.

Thus the time period is nearly the same and all the methods (time evolution code, superposition code etc.) implemented for obtaining this stay justified. Actually, the time period would be exactly same by these calculations. The reason we are not getting it is the skipping of multiple iterations for the speed of our calculations. If we skip lesser iterations, we would get exact same time period for the oscillations of superposed state.

Some more possible proceedings related to this topic is discussed in Section 6.3.

Chapter 5

Motion of a Gaussian Wavepacket Under Different Potentials

5.1 Introduction

The interaction of a particle under some potential can be understood by solving time dependent Schrödinger equation for that particle. Now, in quantum mechanics, to represent a particle we always need some wavefunction. We can consider a Gaussian wavepacket representing the particle,

$$\Psi(x,t=0) = \frac{1}{(2\pi\sigma^2)^{1/4}} \exp\left(-\frac{(x-x')^2}{2\sigma^2}\right) \exp\left(-ik_0 x\right) \tag{5.1}$$

$$= \frac{1}{(2\pi\sigma^2)^{1/4}} \exp\left(-\frac{(x-x')^2}{2\sigma^2}\right) (\cos k_0 x + i \sin k_0 x) \tag{5.2}$$

Now, if we use this $\Psi(x,t=0)$, we can obtain $\Psi(x,t)$ by using Eq. (2.1).

Here we can use the defined function at Section 2.4 code. Importing that function, we can get solutions at later times for the initial wavefunction $\Psi(x,t=0)$.

In the next section, I have obtained the time evolution of Gaussian wavepacket (Eq. (5.1)) for the following cases:

- Free Particle.
- Step Potential.
- Barrier Potential.

Some analysis of the outputs are discussed in Chapter 6.

5.2 Solutions for Different Potentials

To solve the time time dependent Schrödinger equation, codes from Section 2.4 are imported here.

gaussn_prop.py

```
import numpy as np
import matplotlib.pyplot as plt
import numpy.linalg as npLA
import scipy.linalg as spLA

from fn_TDSE_solve import TDSE_time_evolution

hbar, m = 1, 1

def gausswv(pr, X):
    x1, sig, k0 = pr
    E = (hbar**2/(2*m))*(k0**2 + 1/(2*sig**2))
    a = 1/((2*np.pi)**0.5 *sig)**0.5
    b = -1/(4*sig**2)
    Nx = len(X)
    psigs = [0 for xi in range(Nx)]
    for xi in range(Nx):
        gswv = a*np.exp(b*(X[xi]-x1)**2)
        if gswv < max(abs(X))*1e-10:
            gswv = 0
        gswv *= complex(np.cos(k0*X[xi]), np.sin(k0*X[xi]))
        psigs[xi] = gswv
    return E, psigs

def pot_free(V0, X):
    Nx = len(X)
    V = [V0 for i in range(Nx)]
    return V
def pot_step(pr, X):
    V0 = pr
    Nx = len(X)
    V = [0 for i in range(Nx)]
    V[int(Nx/2):] = [V0 for i in range(int(Nx/2))]
    return V
def pot_barrier(pr, X):
    V0, thk = pr
    Nx = len(X)
    V = [0 for i in range(Nx)]
    V[int(Nx/2):int(Nx/2)+thk] = [V0 for i in range(thk)]
    return V
def pot_periodic(pr, X):
    V0, b = pr
    Nx = len(X)
    a, n1 = 4*b, int(Nx/5)
    V = [0 for i in range(Nx)]
    while True:
        V[n1:n1+int(b)] = [V0 for i in range(int(b))]
        n1 += a+b
        if n1 + int(b) > Nx:
            break
    return V
def pot_nuclear(pr, X):
    Nx = len(X)
    Vd, Vc = pr
    V = [0 for i in range(Nx)]
    V[:int(Nx/3)] = [Vd for i in range(int(Nx/3))]
    V[int(Nx/3):] = [Vc*X[int(Nx/3)]/X[i] for i in range(int(Nx/3), Nx)]
    return V

def Gaussian_wavepacket_evolution(pot):
    if pot == 'pot_free':
        potfn = pot_free
        x0, xN, Nx = 0, 30, 100
        X = np.linspace(0, 30, 100)
        Nx = len(X)
        x0, xN = X[0], X[Nx-1]
        sig = 0.5
        x1, k0 = 5, np.pi
        V0 = 1
        prpt = V0
        pr = [hbar, m]
        prwv = [x1, sig, k0]
        iters_solve, T_max, pause_time = 2, 10, 0.01
        print('Free Particle plot')
    elif pot == 'pot_step':
        potfn = pot_step
        X = np.linspace(0, 50, 100)
        sig = 1.5
        x1, k0 = 15, 1.5*np.pi
        V0 = 5
        pr = [hbar, m]
        prwv = [x1, sig, k0]
        prpt = V0
        iters_solve, T_max, pause_time = 3, 30, 0.01
        print('Potential Step plot')
    elif pot == 'pot_barrier':
        potfn = pot_barrier
        X = np.linspace(0, 40, 100)
        sig = 1
        x1, k0 = 12, np.pi
        V0, thk = 5, 10
        pr = [hbar, m]
        prwv = [x1, sig, k0]
        prpt = [V0, thk]
        iters_solve, T_max, pause_time = 3, 21, 0.01
        print('Potential Barrier plot')
    TDSE_time_evolution(pr, gausswv, prwv, potfn, prpt, X,
                        iters_solve, T_max, pause_time, method_='LU_solve_def', plot_
 ↪=True)
```

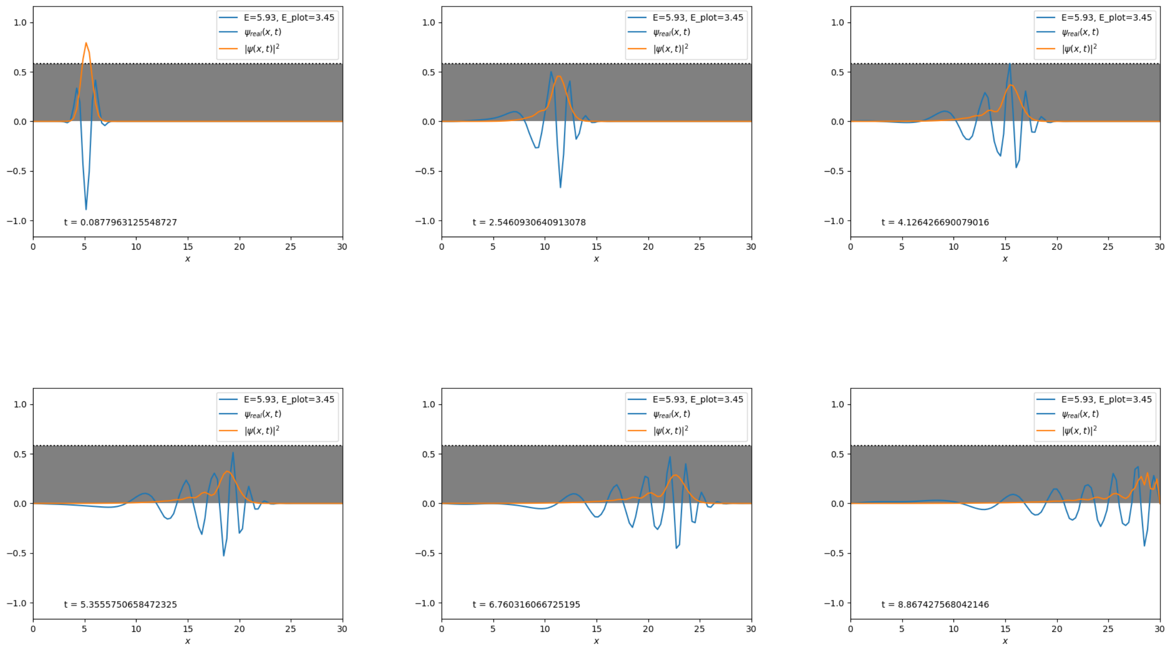
```
    return None
```

presentation_gaussn.py

```
from gaussn_prop import Gaussian_wavepacket_evolution

Gaussian_wavepacket_evolution(pot = 'pot_free')
# pot options: pot_free, pot_step, pot_barrier
```

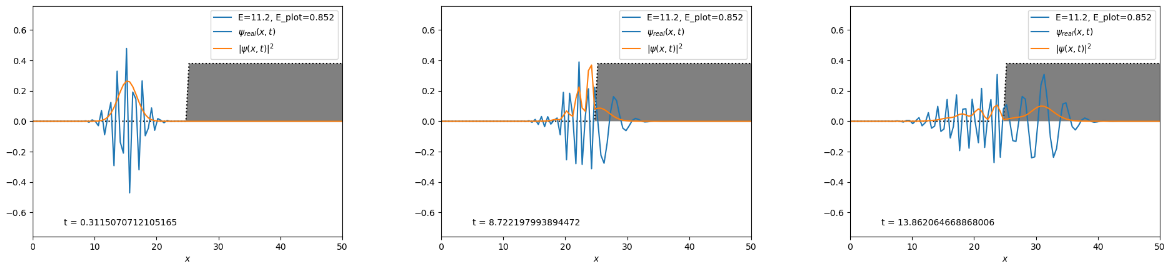
For this code, we would have the following outputs,

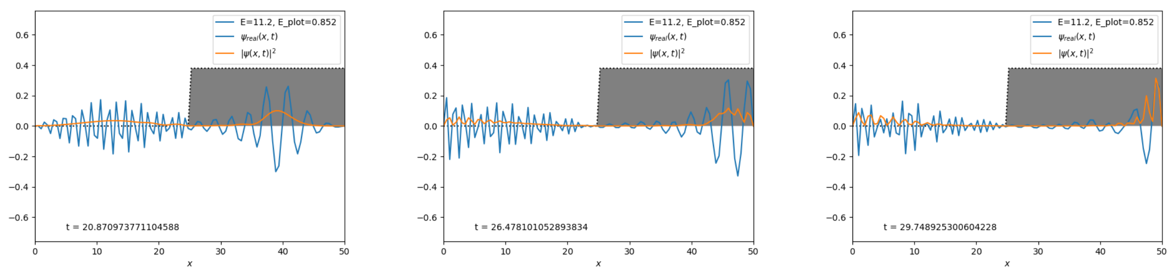
- If we choose `pot = 'pot_free'`,



Click to view the plots.

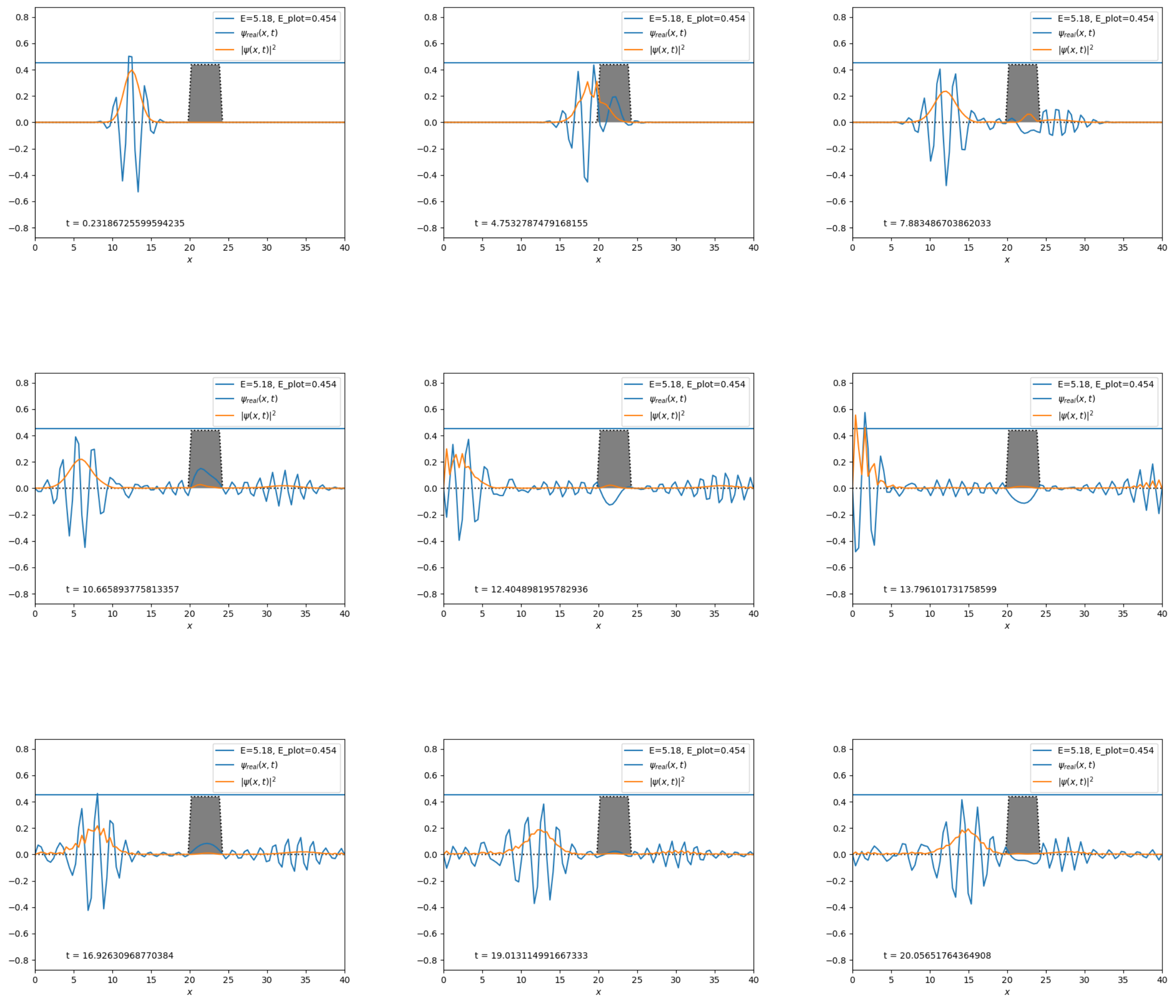
- If we choose `pot = 'pot_step'`,





Click to view the plots.

- If we choose `pot = 'pot_barrier'`,



Click to view the plots.

The animations can be found at `https://github.com/suman122003/BSc_Physics_Codes/blob/main/DSE_4_Computational_QM_project`.

An analytical solution for the motion of a free particle (with Gaussian wavepacket) is given in [1] (page 48).

Chapter 6

Conclusion and Future Works

I have tried to get the values of reflection and transmission coefficients from the obtained wavefunctions after time evolution and match those values with the exact values of reflection and transmission coefficients for the step potential problem, [1] (page 222).

Thus the values are not matched here. This issue is not solved yet and some future works on it may be done later.

6.1 Potential Step Problem

[1]:
```
import numpy as np
import numpy.linalg as npLA
import scipy.linalg as spLA
import matplotlib.pyplot as plt
```

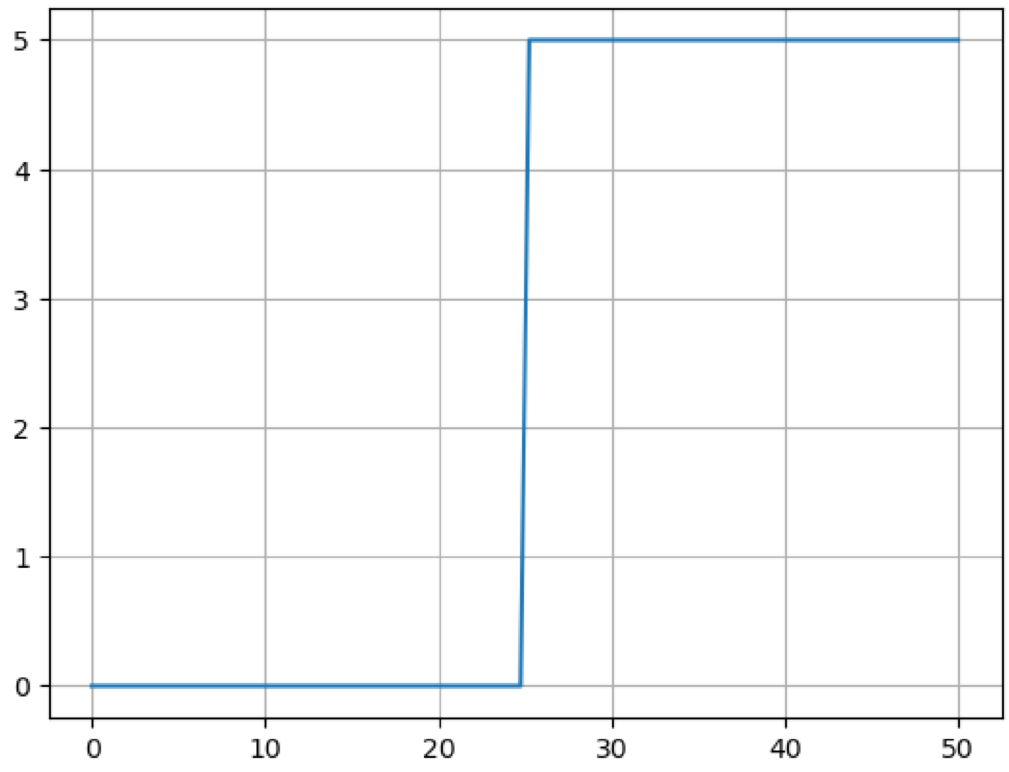
[2]:
```
from fn_TDSE_solve import TDSE_time_evolution
```

[3]:
```
hbar, m = 1, 1

def gausswv(pr, X):
    x1, sig, k0 = pr
    E = (hbar**2/(2*m))*(k0**2 + 1/(2*sig**2))
    a = 1/((2*np.pi)**0.5 *sig)**0.5
    b = -1/(4*sig**2)
    Nx = len(X)
    psigs = [0 for xi in range(Nx)]
    for xi in range(Nx):
        gswv = a*np.exp(b*(X[xi]-x1)**2)
        if gswv < max(abs(X))*1e-10:
            gswv = 0
        gswv *= complex(np.cos(k0*X[xi]), np.sin(k0*X[xi]))
        psigs[xi] = gswv
    return E, psigs
```

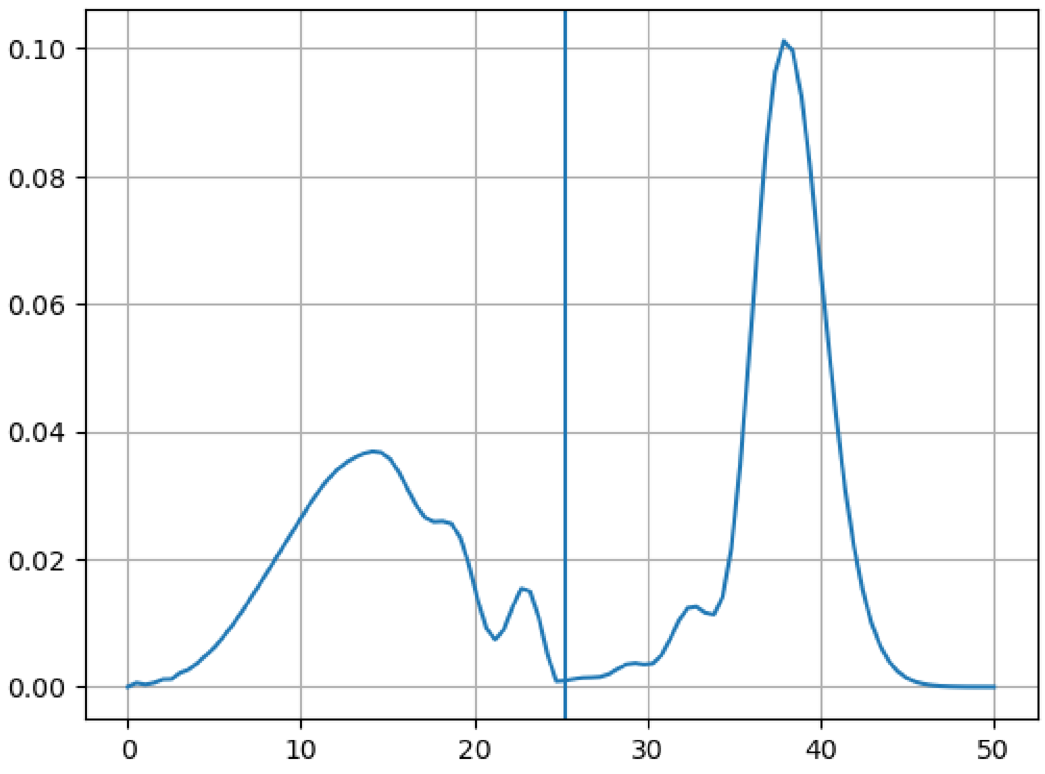
[4]:
```
def pot_step(pr, X):
    V0 = pr
    Nx = len(X)
    V = [0 for i in range(Nx)]
    V[int(Nx/2):] = [V0 for i in range(int(Nx/2))]
    return V

X = np.linspace(0, 50, 100)
sig = 1.5
x1, k0 = 15, 1.5*np.pi
V0 = 5
pr = [hbar, m]
prwv = [x1, sig, k0]
prpt = V0
iters_solve, T_max, pause_time = 3, 30, 0.01

Ts, E, PSIR, PSII, PSIMD2 = TDSE_time_evolution(pr, gausswv, prwv, pot_step,␣
 ↪prpt, X,
                iters_solve, T_max, pause_time, method_='LU_solve_def',␣
 ↪return_=True)
```

```
[5]: # index: near 40
ti = 42
Nx = len(X)
x0, xN = X[0], X[Nx-1]
bnd1 = int(Nx/2)
t, psi2 = Ts[ti], PSIMD2[ti]
E1 = E - hbar**2/(2*m*sig**2)
print(f'Energy={E}, reduced energy={E1}, time={t}')
plt.plot(X, pot_step(prpt, X))
plt.grid()
plt.show()
plt.plot(X, psi2)
plt.axvline(X[bnd1])
plt.grid()
plt.show()
```

```
Energy=11.214416062336639, reduced energy=10.992193840114417,
time=19.936452557473046
```



5
4
3
2
1
0
0
10
20
30
40
50



0.10
0.08
0.06
0.04
0.02
0.00
0
10
20
30
40
50

```
[6]: psi21, psi22 = psi2[:bnd1], psi2[bnd1:]
     dx = (xN-x0)/Nx
     prob_total = np.sum(psi2*dx)
     prob1, prob2 = np.sum(psi21*dx), np.sum(psi22*dx)
     print(prob_total)
     Rplot, Tplot = prob1/prob_total, prob2
     print(Rplot+Tplot)
     print(f'Calculated values: R={Rplot} and T={Tplot}')
     Kp = np.sqrt(1-V0/E1)
     Rexact = (1-Kp)**2/(1+Kp)**2
     print(f'Analytical values: R={Rexact}, T={1-Rexact}')
```

```
0.9899999999999798
0.9944292186112168
Calculated values: R=0.44292186112281884 and T=0.551507357488398
Analytical values: R=0.02265906435242474, T=0.9773409356475753
```

```
[ ]:
```

Thus, the attempt goes unsuccessful here.

6.2 Alpha Decay Problem

Once we sort out the issue for the Section 6.1, then we can have some surety about the correctness of the method we applied. For a method to be correct in simple problems (line potential step problem), we can try that method for solving the α-decay problem. Like Section 6.1, in the case of α-decay problem we need to calculate the transmission coefficient. Once, we get the transmission coefficient, then we can proceed to calculate the lifetime (half-life or mean-life) of α decay for a certain energy of α particle. We can compare these calculated results with the experimental values.

6.3 On Frequency of Oscillations for a Superposed State

This idea is introduced by me in Section 4.8. A further analysis can be made on this by obtaining the value of energy and frequency of oscillations for a particular superposed state under different potentials. This can be done by fitting the superposed state by using eigenstates for that particular potential. Some future works are supposed to be done on this.

Bibliography

[1] Zettili, N. (2009) Quantum Mechanics: Concepts and Applications. Wiley.

[2] Crank J, Nicolson P. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. Mathematical Proceedings of the Cambridge Philosophical Society. 1947;43(1):50-67. doi:10.1017/S0305004100023197.

[3] Wikipedia - Finite difference method.

[4] Newman, M. E. J. (2013). Computational physics. Revised and expanded 2013 [Washington (D.C.): CreateSpace].

[5] Gezerlis, A. (2020) Numerical Methods in Physics with Python. Cambridge: Cambridge University Press.

[6] Dr. Pradipta Kumar Mandal. Physics in Laboratory - Python Programming - B.Sc. Semester IV & V. Santra Publication.

[7] Solving 2D Time Independent Schrodinger Equation Using Numerical Method - Wong Wai Kui, Chris - December 8, 2021.
DOI:10.13140/RG.2.2.12835.99360.

[8] Function: `numpy.linalg.inv`.
Link: `https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html`

[9] Function: `numpy.linalg.solve`.
Link: `https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html`

[10] Function: `scipy.linalg.inv`.
Link: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.inv.html`

[11] Function: `scipy.linalg.lu_factor`.
Link: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_factor.html`

[12] Function: `scipy.linalg.lu_solve`.
Link: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_solve.html`

[13] Wikipedia - Atomic units.

[14] Function: `scipy.constants`.
Link: https://docs.scipy.org/doc/scipy/reference/constants.html