Personal Project Report

Simulation of Quantum Wave Packet Evolution Using the Crank–Nicolson Method

---

A Numerical Study of the Time-Dependent Schrödinger Equation

Shashank Verma

Self-Initiated Project
(Undergraduate Student, Department of Physics)
NIT Surat

Date: August 11, 2025

Abstract

This project presents a detailed numerical investigation into the quantum behavior of an electron confined within a one-dimensional infinite potential well of width 10 nanometers. The study employs the Crank–Nicolson method—an implicit finite-difference technique known for its unconditional stability and second-order accuracy in both time and space—to solve the time-dependent Schrödinger equation (TDSE). The electron is initially described by a Gaussian wave packet centered within the well, and its evolution is simulated over 2000 discrete time steps.

The simulation captures the dynamic evolution of the wavefunction, illustrating key quantum mechanical features such as standing wave formation, interference patterns, and the spread of the wave packet due to dispersion. At the 2000th step, the wavefunction exhibits spatially oscillatory behavior characterized by well-defined nodes and antinodes, indicative of constructive and destructive interference among the system's eigenstates. The corresponding probability density shows localization in specific regions of the well, aligning with expected quantum confinement effects.

In addition to final-state analysis, the study includes both 2D and 3D visualizations of the wavefunction at multiple intermediate time steps (500, 1000, 1500, and 2000), offering insight into the time-dependent nature of quantum superposition. These visualizations vividly demonstrate the coherent evolution of the system and enhance understanding of quantum phase, norm conservation, and wave–particle duality.

Overall, this simulation confirms the robustness and reliability of the Crank–Nicolson method for modeling time-dependent quantum systems. The results not only validate theoretical predictions but also serve as a foundation for exploring more complex potentials and multidimensional quantum scenarios using numerical techniques.

Contents

Chapter 1

Introduction

Quantum mechanics has profoundly reshaped our understanding of the physical world at microscopic scales. Central to this framework is the Schrödinger equation, which governs the behavior and evolution of quantum systems. However, obtaining exact analytical solutions—especially for time-dependent scenarios—is often impractical due to the mathematical complexity involved. This has led to the growing importance of numerical techniques that can reliably approximate quantum dynamics under a variety of physical constraints.

The motivation behind this project stems from the desire to explore the time evolution of a quantum system governed by specific initial and boundary conditions. Traditional analytical tools frequently fall short in offering intuitive or complete insight into such dynamical processes, largely due to the abstract and non-classical nature of quantum mechanics. In response, computational methods serve as powerful alternatives, capable of revealing detailed temporal behavior and supporting applications in modern technologies such as quantum computing, secure communication, and tunneling-based devices.

In this work, the objective is to simulate the quantum behavior of an electron initially localized within a one-dimensional infinite potential well of width 10 nanometers. The system is initialized with a Gaussian wave packet at time $t = 0$. Solving the time-dependent Schrödinger equation for this setup poses unique computational challenges, primarily due to the need for both numerical stability and high accuracy over extended time periods. To address this, the Crank–Nicolson method—a well-established finite-difference scheme—is employed. This method offers unconditional stability and second-order accuracy in both space and time domains.

To implement this, the spatial and temporal domains are discretized uniformly, and Taylor series expansions are used to derive a central finite-difference equation. This forms the basis for a tridiagonal matrix equation that relates the wavefunction at the current time step to its future evolution. The simulation tracks the progression of the wavefunction's real and imaginary components, along with its probability density, at each time step. Throughout, the normalization of the wavefunction and conservation of probability are monitored to ensure physical validity and numerical integrity of the simulation.

Chapter 2

Methodology

2.1 Finite Difference Method

Before deriving the Crank-Nicolson method, it's important to understand the finite difference techniques. Consider a heat equation(or diffusion equation),

$$u_t = \mathcal{L}u_{xx} \tag{1}$$

Suppose you restrict the domain of our work to a rectangular configuration [2.1] with x ranging from x_{min} to x_{max} and t ranging from 0 to T. Discretize the the ranges $[0, T]$ and $[x_{min}, x_{max}]$ into N and I equally spaced intervals, indexed at $n = 0, 1, ..., N$ and $i = 0, 1, ..., I$ respectively. Once we have these u values, since u is assumed to be smooth almost everywhere, we can interpolate within this grid to get values for arbitrary x, t. We can approximate the partial derivatives of u at each gridpoint by difference expressions in the yet unknown u_i^n's. Let $u_{i,n}$ denote our numerical approximation at the gridpoint, such that position and time are discretized as $x = x_{min} + ih$ and $t = nk$. Consider the boundary condition that $x_{min} = 0$. We can now apply this set of finite difference equations to the grid, using the grid point $u = u(x_i, t_n) = u_i^n$.

Here, $h = \Delta x$ is the mesh spacing on the x-axis and $k = \Delta t$ is the time step.

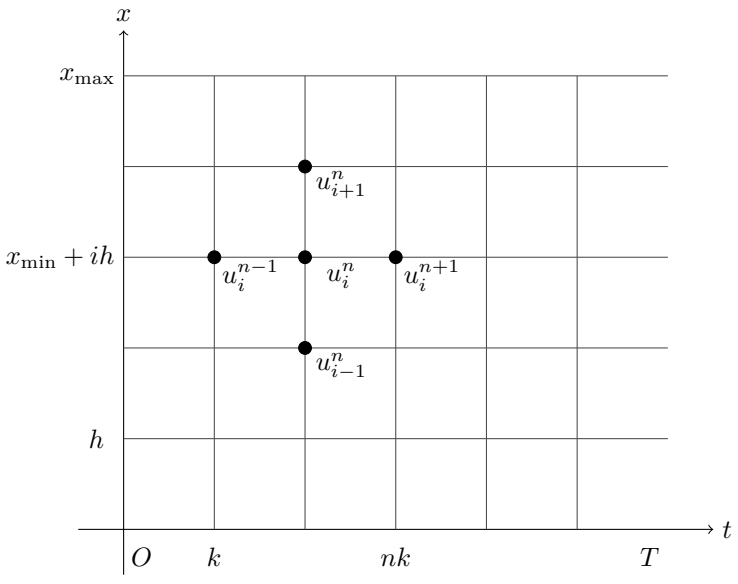


Figure 2.1: PDE solution grid

The methods involved in approximation are either explicit or implicit:

Explicit Methods : Explicit methods calculate the next state of a system based directly on the current state. It's like extrapolating forward in time using known information. These methods may be unstable and require smaller time steps for accuracy.
Implicit Methods : Implicit methods involve solving a system of equations to determine the next state, often involving the future state as well, and often involving both current and future states. These methods are comparatively more stable and can handle larger time steps, especially in static or quasi-static problems. Let's look into two finite difference schemes.

1. Forward Euler Method
2. Backward Euler Method

Forward Euler Method :
Since the heat equation is an evolution that can be solved forward in time, we set up our difference equations in a form where we can march forward in time, determining the values of $u_{i,n+1}$ for all i from the values $u_{i,n}$ at the previous time level, or perhaps using also values at earlier time levels with a multi-step formula.
We can approximate the partial derivatives at an internal grid point by the equations,

$$u = u_i^n, \quad \frac{\partial u}{\partial t} = \frac{u_i^{n+1} - u_i^n}{k} = \mathcal{L}u_i^n \tag{2}$$

Inserting into the PDE, we obtain,

$$\frac{u_i^{n+1} - u_i^n}{k} = \mathcal{L}u_i^n \implies u_i^{n+1} = u_i^n + k \cdot \mathcal{L}u_i^n \tag{3}$$

This method is conditionally stable because it requires a small $\Delta t = k$ for accuracy and stability. Also, it is an explicit method.

Backward Euler Method :
Using the same finite difference method as earlier, evaluate $\mathcal{L}u$ at the next time step.

$$\frac{u_i^{n+1} - u_i^n}{k} = \mathcal{L}u_i^{n+1} \implies (I - k \cdot \mathcal{L})u_i^{n+1} = u_i^n \tag{4}$$

This method is unconditionally stable for linear problems but requires solving a linear system at every time step.

2.2 Crank-Nicolson Method

Crank-Nicolson method is a numerical technique used for solving time-dependent partial differential equations (PDEs) of the form

$$\frac{\partial y(x,t)}{\partial t} = \mathcal{L}y(x,t) \tag{5}$$

where y(x,t) is a function dependent on position and time, variations in position being the function of variations in time.
This method can be applied to the analysis of time-dependent equations, particularly parabolic equations such as the heat equation (or diffusion equation) and the time-dependent Schrödinger equation.

Crank-Nicolson is based on the finite difference method, and it improves both explicit and implicit time-stepping methods. It involves the averaging of the forward (explicit) and backward (implicit) Euler methods. This is an implicit method, meaning it requires solving a system of equations at each step, but it offers better stability (unconditional stability, i.e., for all k/h^2) and accuracy compared to explicit methods.

Intuition: Imagine you are tracking how a function $y(x,t)$ evolves over time - e.g., temperature in a rod or a wavefunction in a potential well. You know how it changes now (at time t^n, and want to predict its value at the next step t^{n+1}. For the physics problems, we can average between the current and the next step.

It's more like instead of using what we know only now or assuming the next, we can average both for better accuracy and stability.

So far, we have been introduced to the concepts of forward and backward difference methods. However, the forward method was not stable for large Δt, while the backward method was stable but had less accuracy. So to avoid these issues, the Crank-Nicolson method averages both(or we can also say that we take the time-derivative at $t + k/2$ rather than around t),

$$\frac{u_i^{n+1} - u_i^n}{k} = \frac{1}{2}(\mathcal{L}u_i^n + \mathcal{L}u_i^{n+1}) \tag{6}$$

This keeps both the accuracy of forward Euler(second order) and the stability of backward Euler(implicit) and avoids large errors even for larger Δt.
For this method we need to discretize $\mathcal{L}$ with cetnral finite differences,

$$u = u_i^n, \quad \frac{\partial u}{\partial t} = \frac{u_i^{n+1} - u_i^n}{k} = \mathcal{L}u_i^n \tag{7}$$

$$\frac{\partial u}{\partial x} = \frac{u_{i+1}^n - u_{i-1}^n}{2h}, \quad \frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2} \tag{8}$$

Let's derive this discretization, First expand $u(x + \Delta x)$ and $u(x - \Delta x)$ using in a Taylor series about the point x:

$$u(x + \Delta x) = u(x) + \frac{\partial u}{\partial x}\Delta x + \frac{1}{2}\frac{\partial^2 u}{\partial x^2}(\Delta x)^2 + \mathcal{O}((\Delta x)^3) \tag{9}$$

$$u(x - \Delta x) = u(x) - \frac{\partial u}{\partial x}\Delta x + \frac{1}{2}\frac{\partial^2 u}{\partial x^2}(\Delta x)^2 + \mathcal{O}((\Delta x)^3) \tag{10}$$

Here, $O((\Delta x)^3)$ denotes the remainder terms in the Taylor series expansion, which become negligible as Δx becomes very small.
Adding (10) from (9) to eliminate the first derivative term, while neglecting the higher-order terms:

$$u(x + \Delta x) + u(x - \Delta x) = 2u(x) + \frac{\partial^2 u}{\partial x^2}(\Delta x)^2 \tag{11}$$

On adding, the first derivative terms cancel out, and the odd-powered remainder terms also cancel out, leaving us with only the even-powered terms, specifically the second derivative term. Rearranging equation (11),

$$u(x + \Delta x) - 2u(x) + u(x - \Delta x) = \frac{\partial^2 u}{\partial x^2}(\Delta x)^2 \tag{12}$$

We get the central difference approximation for the second derivative,

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{j-1}^n}{(\Delta x)^2} \tag{13}$$

So we have approximated the continuous second derivative with discrete points, which can be computed numerically. Now, putting the values of partial derivatives into equation (6)

$$\frac{u_i^{n+1} - u_i^n}{k} = \frac{1}{2}(\frac{\partial^2 u_i^{n+1}}{\partial x^2} + \frac{\partial^2 u_i^n}{\partial x^2}) \tag{14}$$

$$\frac{u_i^{n+1} - u_i^n}{k} = \frac{1}{2}(\frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{h^2} + \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2}) \tag{15}$$

Rewriting this equation,

$$u_i^{n+1} = u_i^n + \frac{k}{2h^2}(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1} + u_{i+1}^n - 2u_i^n + u_{i-1}^n) \tag{16}$$

rearranging the equation,

$$-\gamma u_{i-1}^{n+1} + (1 + 2\gamma)u_i^{n+1} - \gamma u_{i+1}^{n+1} = \gamma u_{i-1}^n + (1 - 2\gamma)u_i^n + \gamma u_{i+1}^n, \tag{17}$$

where $\gamma = k/2h^2$. This is an implicit method and gives a tridiagonal[1] system of equations to solve for all the values of u_i^{n+1} simultaneously.

$$\begin{bmatrix} (1+2\gamma) & -\gamma & & & & & \\ -\gamma & (1+2\gamma) & -\gamma & & & & \\ & -\gamma & (1+2\gamma) & -\gamma & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -\gamma & (1+2\gamma) & -\gamma & \\ & & & & -\gamma & (1+2\gamma) & \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_{m-1}^{n+1} \\ u_m^{n+1} \end{bmatrix}$$

$$= \begin{bmatrix} \gamma(g_0(t_n)) + g_0(t_{n+1}) + (1-2\gamma)u_1^n + \gamma u_2^n \\ \gamma u_1^n + (1-2\gamma)u_2^n + \gamma u_3^n \\ \gamma u_2^n + (1-2\gamma)u_3^n + \gamma u_4^n \\ \vdots \\ \gamma u_{m-2}^n + (1-2\gamma)u_{m-1}^n + \gamma u_m^n \\ \gamma u_{m-1}^n + (1-2\gamma)u_m^n + \gamma(g_1(t_n) + g_1(t_{n+1})) \end{bmatrix} \tag{18}$$

Note how the boundary conditions $u(0,t) = g_0(t)$ and $u(1,t) = g_1(t)$ come into these equations. Since a tridiagonal system of m equations can be solved with $O(m)$ work, this method is essentially as efficient per time step as an explicit method.

[1] A tridiagonal matrix is a square matrix where non-zero elements are limited to the main diagonal, the diagonal above it (superdiagonal), and the diagonal below it (subdiagonal).

Chapter 3

Quantum Wave Packet Simulation using Crank-Nicolson Method

3.1 Crank-Nicolson Discretization for Schrödinger equation

The one-dimensional Schrödinger equation for a particle with potential energy $V(x)$ is given by :

$$i\hbar\frac{\partial\Psi(x,t)}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\Psi(x,t)}{\partial x^2} + V(x)\Psi(x,t) \tag{19}$$

Now, let's discretize this problem using the Crank-Nicolson method: We didcretize the spatial domains into point $x_i = i\Delta x$ and the temporal domain into points $t_n = n\Delta t$ for $n = 0, 1, ..., M$. The wavefunctions at these points are represented as Ψ_i^n.

For the time derivative at the midpoint between t_n and t_{n+1}, we use the following finite difference approximation:

$$\frac{\partial\Psi}{\partial t} \approx \frac{\Psi_i^{n+1} - \Psi_i^n}{\Delta t} \tag{20}$$

For the spatial derivative, we use the central differences at times t_n and t_{n+1}, averaging them to center the spatial derivative in time:

$$\frac{\partial^2\Psi}{\partial x^2} \approx \frac{1}{2}\left(\frac{\Psi_{i+1}^{n+1} - 2\Psi_i^{n+1} + \Psi_{i-1}^{n+1}}{\Delta x^2} + \frac{\Psi_{i+1}^n - 2\Psi_i^n + \Psi_{i-1}^n}{\Delta x^2}\right) \tag{21}$$

We substitute these finite differences into the Schrödinger equation, yielding a discretized form that connects two time levels:

$$i\hbar\frac{\Psi_i^{n+1} - \Psi_i^n}{\Delta t} = -\frac{\hbar^2}{4m}\left(\frac{\Psi_{i+1}^{n+1} - 2\Psi_i^{n+1} + \Psi_{i-1}^{n+1}}{\Delta x^2} + \frac{\Psi_{i+1}^n - 2\Psi_i^n + \Psi_{i-1}^n}{\Delta x^2}\right) + \frac{V_i}{2}(\Psi_i^{n+1} + \Psi_i^n) \tag{22}$$

Rearranging such that the same time levels are on the same side:

$$\begin{aligned}\Psi_i^{n+1} + \frac{i\Delta t}{2\hbar}\left[\frac{\hbar^2}{2m\Delta x^2}\left(-\Psi_{i+1}^{n+1} + 2\Psi_i^{n+1} - \Psi_{i-1}^{n+1}\right) - V_i\Psi_i^{n+1}\right] \\ = \Psi_i^n - \frac{i\Delta t}{2\hbar}\left[\frac{\hbar^2}{2m\Delta x^2}\left(-\Psi_{i+1}^n + 2\Psi_i^n - \Psi_{i-1}^n\right) - V_i\Psi_i^n\right]\end{aligned} \tag{23}$$

We'll define matrix A, such that its elements are the coefficients of the equation, $-\Psi_{i+1} + 2\Psi_i - \Psi_{i-1} = \begin{pmatrix}-1 & 2 & -1\end{pmatrix}\begin{pmatrix}\Psi_{i-1}\\ \Psi_i \\ \Psi_{i+1}\end{pmatrix}$ where $i = 1, 2, 3..., N$, and boundary condition, $\Psi_0 = 0$ and $\Psi_N = 0$. We can

construct the matrix A from above equation, such that $A = \{-1, 2, 1\}$: In this equation, the coefficients for Ψ_{i-1}, Ψ_i and Ψ_{i+1} are $-1, 2$ and -1 respectively. Observing this pattern, we can understand how the matrix A is constructed.

$$A = \begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ 0 & -1 & 2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & -1 \\ 0 & 0 & \dots & -1 & 2 \end{pmatrix} \tag{24}$$

Now, we represent this equation in matrix form. Define matrices T_1 and T_2 based on the spatial discretization of the kinetic and potential energy terms:

$$T_1 = \frac{\hbar^2}{2m\Delta x^2} A = \frac{\hbar^2}{2m\Delta x^2} \begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ 0 & -1 & 2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & -1 \\ 0 & 0 & \dots & -1 & 2 \end{pmatrix} \tag{25}$$

$$T_2 = diag(V_1, V_2, ..., V_N) \tag{25}$$

The Crank-Nicolson method then takes the form:

$$(I + \frac{i\hbar\Delta t}{2\hbar^2}(T_1 + T_2))\Psi^{n+1} = (I - \frac{i\hbar\Delta t}{2\hbar^2}(T_1 + T_2))\Psi^n \tag{26}$$

where I is the identity matrix of the same dimension as A.

3.2 Algorithm for solving the Schrödinger equation

The simulation program is designed to illustrate how the electron wavefunction evolves over time in a one-dimensional potential well. It employs the Crank-Nicolson method, which is a numerical technique known for its stability and accuracy. This method is unconditionally stable and achieves second-order accuracy in both time and space.

3.2.1 Constants and Parameters :

- $L = 1 \times 10^{-8}$ meters (Length of the potential Well)
- $N = 1024$ (Number of spatial grids)
- $x = L/N$ (Spatial grids)
- $\hbar = 1.0545718 \times 10^{-34}$ Joule-seconds (Reduced Planck's constant).
- $m = 10938356 \times 10^{-34}$ kilograms (Mass of an electron)
- $h = 1 \times 10^{-18}$ seconds (Time step for the simulation)
- $k = x[1] - x[0]$ (Spacing between adjacent points in the array)
- $\sigma = 1 \times 10^{-10}$ meters (Width of the Guassian wae packet)
- $x_0 = L/2$ meters (Initial center of the wavefunction)
- $k_0 = 5 \times 10^{10}$ meters^{-1} (Initial wavenumber)
- steps = 2000 (Time step)

3.2.2 Initial Setup :

Our problem involves a particle trapped in a potential well of length L where $V(x) = \infty$ at the boundaries of the finite well, and zero at rest other positions in between.

Wavefunction initialization, Ψ_0 : The wavefunction is initialized using a Gaussian wave packet centered at $x_0 = L/2$, with a width $\sigma = 1 \times 10^{-10}$ meters and a wave number $k_0 = 5 \times 10^{10}$ meters^{-1}. This initial wavefunction is normalized to ensure its integral over the spatial domain equals unity.

$$\Psi(x,0) = Ae^{-\frac{(x-x_0)^2}{2\sigma^2}} e^{ik_0 x} \tag{27}$$

where :

- A is a normalization constant (Here, taken to be unity, $A = 1$),
- x_0 is the initial position of the center of the wave packet,
- σ specifies the width of the wave packet,
- k_0 is the initial wave number, related to the particle's momentum,

3.2.3 Matrix Setup with Crank-Nicolson Method :

The above equation (22) can be expressed in the following form:

$$i\hbar\frac{\Psi_i^{n+1} - \Psi_i^n}{\Delta t} = -\frac{\hbar^2}{4m}\left(\frac{\Psi_{i+1}^{n+1} - 2\Psi_i^{n+1} + \Psi_{i-1}^{n+1}}{\Delta x^2} + \frac{\Psi_{i+1}^{n} - 2\Psi_i^{n} + \Psi_{i-1}^{n}}{\Delta x^2}\right) \tag{28}$$

In this case, we must get the following matrix forms, where the matrices A and B are defined for the Crank-Nicolson method:

$$A = I + \frac{i\hbar\Delta t}{2m\Delta x^2}T \tag{29}$$

$$B = I - \frac{i\hbar\Delta t}{2m\Delta x^2}T \tag{30}$$

where T is the tridiagonal matrix:

$$T = \frac{\hbar^2}{2m\Delta x^2}\begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ 0 & -1 & 2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & -1 \\ 0 & 0 & \dots & -1 & 2 \end{pmatrix} \tag{31}$$

and I is the identity matrix of the same dimension as T.
Boundary conditions:

$$\Psi^n(0) = \Psi^n(L) = 0, \quad \text{for all } n \tag{32}$$

The equation then solves for Ψ^{n+1} from:

$$A\Psi^{n+1} = B\Psi^n \tag{33}$$

The matrix equation represents a linear system that can be efficiently solved using tridiagonal matrix algorithms, given the initial condition Ψ^0 and applying the boundary conditions at each time step. Some variables used for Crank-Nicolson matrix construction:

- a is the dimensionless numerical parameter that links the time step k and spatial h A, calculated as $a = i\frac{\hbar\Delta t}{4m\Delta x^2}$,
- **main_diag**: Coefficient for the Main-diagonal elements, calculated as main_diag $= (1 + 2a)I_N$, I_N is the identity matrix of order N.
- **off_diag**: Coefficient for the Off-diagonal elements, calculated as off_diag $= -aI_{N-1}$, I_{N-1} is the identity matrix of order $N - 1$.

3.2.4 Matrix Construction

Discretization is done via tridiagonal matrices A and B:

- A : Sparse matrix A for coefficients which multiply with Ψ^{n+1} at time level $n+1$.
- B : Sparse tridiagonal matrix B for coefficients which multiply with Ψ^{n} at time level n.

The Matrix construction uses spsolve() for tridiagonal sparse matrices.

3.2.5 Static Visualization

- **2D Visualization:** The real, imaginary, and probability density parts of the wavefunction are plotted at selected time steps such as 500th, 1000th, 1500th, and 2000th, to illustrate the wavefunction's future evolution.
- **3D Visualization:** The 3D plot shows the complex nature of the wavefunction, combining real and imaginary components across the selected time steps.

3.2.6 Animations

a) **2D Animation:** Separate animations for the real part, imaginary part, and probability density of the wavefunction are generated using `FuncAnimation`. Each frame dynamically updates the corresponding wavefunction component to show its evolution over time.

- Each frame shows the wavefunction's respective component along the spatial axis.
- `xlim` and `ylim` parameters are configured to fixed ranges to ensure a consistent visual scale throughout the animation.
- The animation progresses through each timestep, illustrating how the wavefunction evolves from its initial state to the end of the simulation.

b) **3D Animation:** The 3D animations combine the real and imaginary parts of the wavefunction to provide a comprehensive view of its complex nature over time.

- The 3D visualization is established using `Axes3D`, enabling 3D plotting capabilities.
- The real and imaginary parts of the wavefunction are plotted along the spatial axis, providing a dynamic view of its temporal evolution.
- The `animate_3D` function synchronously updates both components in each frame, allowing observation of their coupled behavior over time.

3.2.7 Libraries and Modules

The simulation employs various Python libraries and modules, each fulfilling specific roles in numerical calculations, data visualization, and the creation of dynamic animations. In this section, we will explore the purpose of each module within the simulation process.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from scipy.sparse import diags
from scipy.sparse.linalg import spsolve
from mpl_toolkits.mplot3d import Axes3D

import os
def clear_screen():
    os.system('cls' if os.name == 'nt' else 'clear')
clear_screen()
```

a) **NumPy (Numerical Python):**

NumPy is the fundamental package for scientific computing in Python. This library is utilized for its robust support of numerical arrays and mathematical operations. It enables efficient manipulation of large datasets, such as arrays and matrices, which are essential for carrying out the intricate calculations involved in solving quantum mechanical problems.

It provides:

- Efficient array operations (vectors/matrices) that are faster than Python loops.
- Complex number support, crucial for quantum wavefunctions like $\Psi(x) = e^{-x^2}e^{ikx}$.
- Vectorized math (e.g., `np.exp(), np.abs(...), np.real(...), np.imag(...), np.sum(...), etc.`).

```
import numpy as np
```

Role in Simulation:

- `np.linspace(...)`: creates the spatial grid(x) over the wavefunction is defined.
- `np.exp(...)`: computes the Gaussian envelope and plane wave components of the initial wavefunction.
- `np.abs(psi)**2`: calculates the probability density, $|\Psi(x,t)|^2$, at each time step.
- `np.real(...)` and `np.imag(...)`: extract the real and imaginary parts of the wavefunction.
- `np.full(...)` and np.ones(...): create constant-valued arrays used to build the diagonals of the Hamiltonian matrix.

Quantum mechanics relies on manipulating complex-valued wavefunctions over discretized space and time. NumPy is optimized for this kind of linear algebra and array-based computation, making it both readable and high-performance.

b) **MatPlotLib :**

This library is used to generate static, interactive, and animated visualizations in Python. It enables the plotting of various components of the wavefunction—such as its real part, imaginary part, and probability density—throughout the simulation. With its user-friendly interface for creating figures and graphs, it plays a crucial role in analyzing and visually presenting the simulation outcomes.

i) **matplotlib.pyplot-(2D Plotting and Visualization):** This is a plotting module in Python that mimics MATLAB's syntax. It is widely used for visualizing scientific data, including wavefunctions and quantum probability densities.

```
import matplotlib.pyplot as plt
```

Core Functions Used:

- `plt.subplots(...)`: Creates a figure and one or more subplot axes (used to generate 3 vertically stacked plots for real part, imaginary, and probability density).
- `ax.plot(...)`: Plots line graphs of spatial variables like $\psi(x)$, Real$(\psi(x))$, Imag$(\psi(x))$, and $|\psi|^2$.
- `ax.set_xlabel, ax.set_ylabel,ax.set_title`: Label the axes and plots clearly for better understanding and report-ready figures.
- `ax.legend(...)`: Adds a legend to each subplot identifying the quantity plotted (e.g., Real, Imaginary, or Probability Density).
- `plt.tight_layout()` and `constrained_layout=True`: Prevent overlapping labels by automatically adjusting spacing between subplots.
- `plt.show()`: Renders to plot wavefunction snapshots at the 2000th step (or at multiple steps like 500, 1000, 1500, and 2000).
- Helps to visually analyze time-evolution, spreading of the wavepacket, and behavior of real vs. imaginary components.

Visual inspection of $\Psi(x,t)$ and $|\Psi(x,t)|^2$ is crucial in quantum mechanics. Matplotlib gives publication-quality plots, is highly customizable, and integrates smoothly with NumPy arrays.

ii) **matplotlib.animation-(Dynamic Animation of Time Evolution):** This is a sub-module of Matplotlib that enables real-time animations of evolving data, which is essential for visualizing time-dependent quantum phenomena like wave packet evolution.

`FuncAnimation` from Matplotlib's animation module is crucial for generating animations that showcase the time evolution of the wavefunction. These dynamic visualizations effectively illustrate the quantum behavior of the particle throughout the simulation, enhancing the understanding of its temporal dynamics.

```
import matplotlib.animation as animation
```

Core Functions Used:

- `animation.FuncAnimation(...)`: This is the engine that drives your live simulation. It repeatedly calls a user-defined update function (`animate_2D() or animate_3D()`) to refresh plot data at each time step.

  ```
  ani_2D = animation.FuncAnimation(fig, animate_2D, frames=steps,
      interval=20, blit=True)
  ```

- `fig`: The figure on which the animation is rendered.
- `animate_2D or animate_3D`: Function to update the plot each frame.
- `frames=steps`: Number of time steps (2000) over which the animation runs.
- `interval=20`: milliseconds between frames, controls the animation speed.
- `blit=True`: Optimizes redrawing by only updating the changed parts of the figure.

Role in Simulation:

- Animates the evolution if the wavefunction $\Psi(x,t)$ in real-time.
- Visually demonstrates:
  - Quantum dispersion, spreading of a Gaussian packet.
  - Wave interference (in extended applications).
  - The dynamics of both magnitude and phase of the wavefunction.
- Makes the simulation much more intuitive and engaging.

iii) **mpl_toolkits.mplot3d-(3D Plotting Toolkit for MatPlotLib:** This extension of the Matplotlib library is used to create three-dimensional plots. This module is essential for representing the complex form of the wavefunction in 3D space, allowing for a clearer understanding of its time-dependent behavior and structural evolution.

```
from mpl_toolkits.mplot3d import Axes3D
```

- `mpl_tomplkits.mplot3d` is an add-on toolkit that extends Matplotlib's capabilities to 3D plotting.
- It provides the `Axes3D` class, which lets use plot funusons with three variables, like a complex wavefunction with real and imaginary parts over space.

Even though we don't directly use `Axes3D`, importing it registers the 3D projection system with MatPlotLib, enabling us to write:

```
ax = fig.add_subplot(111, projection='3d')
```

Without `from mpl_toolkits.mplot3d import Axes3D`, this line would raise an error.

Role in Simulation:
In our function `Plot3D()` and `Animate3D()`:

```
fig = plt.figure(figsize=(12,8)
ax = fig.add_subplot(111, projection='3d')
ax.plot(x, np.real(psi), np.imag(psi))
```

This creates a 3D curve plot where, spatial position is represented along the x-axis, the Real part of the wavefunction $\mathcal{R}[\Psi(x,t)]$ along the y-axis, and the Imaginary part $Im[\Psi(x,t)]$ along the z-axis. These curves help visualize the complex nature of Ψ, which is otherwise difficult to interpret in 2D.
The `mpl_toolkits.mplot3d` module was used to extend MatPlotLib's plotting capabilities into three dimensions, enabling clear visualization of the real and imaginary components of the quantum wavefunction across space.

c) **SciPy (Scientific Python):**

SciPy is an open-source Python library built on top of NumPy. This Python library, specifically its `linalg` module, provides the `spsolve` function, which is crucial for efficiently solving the sparse matrix systems of linear equations generated by the Crank-Nicolson method. This capability is vital for the iterative computation of the wavefunction at each time step.

It provides advanced numerical routines for Linear algebra, optimization, integration, sparse matrix handling, and differential equations. It is ideal for scientific research and numerical simulations like solving,g the time-independent Schrödinger equation in our case.

i) **scipy.sparse.diags-(Efficient Construction of Sparse Matrices):**

```
        from scipy.sparse import diags
```

In simulations using finite difference or Crank-Nicolson schemes, we often need to construct large matrices representing differential operators. Most of the matrices are tridiagonal.

Instead of storing all elements (including zeros), texttt scipy.sparse.diags allows efficient creation and storage of these matrices in sparse format, significantly reducing memory usage and errors, and speeding up matrix operations.

```
        A = diags([off_diag, main_diag, off_diag], [-1, 0, 1], format='csr')
B = diags([-off_diag, (1 - 2*a)*np.ones(N), -off_diag], [-1, 0, 1], format='csr')
```

- `off_diag`: the subd-diagonal and super-diagonal elements (constant complex number -a).
- `main_diag`: the central diagonal values (i.e., 1+2a).
- `[-1,0,1]`: specifies the diagonals' positions (sub-diagonal, main, and super-diagonal).
- `format='csr'`: build the matrix in Compressed Space Row format for fast row access (used with `spsolve`).

Role in Simulation:

- Constructs matrices A and B used in the Crank-Nicolson method

$$A\Psi^{n+!} = B\Psi^n$$

- These matrices represent the discretized form of the time-dependent Schrödinger equation, incorporating kinetic energy terms (second derivative).

Sparse matrix saves memory and avoids unnecessary computation, and it is automatically optimized for matrix-vector multiplication and linear solutions.

ii) **scipy.sparse.linalg.spsolve-(Sparse Linear System Solver):** It is used to solve a sparse linear system of equations.

```
        from scipy.sparse.linalg import spsolve
        psi = spsolve(A, B @ psi)
```

This line performs one full time-step by solving the Crank-Nicolson matrix equation, efficiently computing the next state of the wavefunction.

- `B @ psi`: Matrix-vector multiplication gives the RHS vector.

- `spsolve(...)`: Solves for ψ^{n+1} using LU decomposition for sparse matrices.
  It solves the linear system:

  $$A \cdot x = b,$$

  for the unknown vector x, where A is a sparse matrix.
  - It is optimized for solving equations where A is a sparse square matrix (e.g., a tridiagonal matrix in finite difference schemes).
  - In our simulation, this corresponds to:

    $$A \cdot \psi^{n+1} = B \cdot \psi^n,$$

    where, A is a sparse matrix built using `diags(...)` (from Crank-Nicolson scheme), $B \cdot \psi^n$ is the right-hand side vector, and ψ^{n+1} is the updated wavefunction at the next time step.
- `psi`: The wavefunction gets updates at every step, enabling time evolution. This line evolves the wavefunction in time by one step, using the Crank-Nicolson discretized form of the Schrödinger equation.

Summary:

Library / Module	Purpose and Usage
`numpy` (`import numpy as np`)	Used for creating and handling arrays, mathematical operations, and efficient vectorized computations. Handles wavefunction arrays, grid generation (`linspace`), and operations like normalization, complex exponentials, and absolute values.
`matplotlib.pyplot` (`import matplotlib.pyplot as plt`)	Used to generate 2D static visualizations such as plots of the probability density and the real and imaginary parts of the wavefunction at different time steps.
`matplotlib.animation` (`import matplotlib.animation as animation`)	Used to animate the evolution of the wavefunction over time by updating plots at each frame. Enables dynamic visualization of the simulation results.
`scipy.sparse.diags` (`from scipy.sparse import diags`)	Creates tridiagonal sparse matrices (A and B) used in the Crank–Nicolson time evolution method. Saves memory and speeds up computation for large systems.
`scipy.sparse.linalg.s` (`from scipy.sparse.linalg import spsolve`)	Efficiently solves the linear system $A\psi^{n+1} = B\psi^n$ at each time step. Works directly with sparse matrices.
`mpl_toolkits.mplot3d` (`from mpl_toolkits.mplot3d import Axes3D`)	Adds support for 3D plots, allowing the complex wavefunction to be visualized as a 3D curve (real vs. imaginary parts over space).
`os` (`import os`)	Used to clear the console before running the simulation using platform-specific shell commands. (`cls` or `clear`)
`time` (`import time`)	Introduces a short delay before terminating the program for smoother ua ser experience.
`sys` (`import sys`)	Provides access to the Python runtime for clean termination of the program with `sys.exit()`.

3.2.8 Function-wise Explanation

a) **Start():**

- Initializes physical constants, temporal grid, constants ($\hbar, m, L,$, etc.), and initial wavefunction (Gaussian wave packet) and, ,
- Constructs matrices A and B for Crank-Nicolson update using sparse tridiagonal diagonals.

```
# Start function to initiate wavefunction
def Start():
    L = 1e-8  # Length of the box in meters
    N = 1024  # Number of spatial slices

    hbar = 1.0545718e-34  # Reduced Planck's constant, J*s
    m = 9.10938356e-31  # Mass of electron, kg
    h = 1e-18  # Time-step in seconds

    # Initial wave function
    x = np.linspace(0, L, N)    # Spatial grid from 0 to L with N points
    x0 = L / 2  # Initial center of the wave packet
    sigma = 1e-10
    k0 = 5e10
    k = x[1]-x[0] #spacing between adjacent points in the array
    dx = k

    # Time grid
    dt = h
    steps = 2000

    # Initial wave packet
    psi0 = np.exp(-(x - x0)**2 / (2 * sigma**2)) * np.exp(1j * k0 * x)  # Initial
        wavefunction
    psi0 /= np.sqrt(np.sum(np.abs(psi0)**2) * dx)  # Normalize

    # Hamiltonian matrix using Crank-Nicolson Method
    a = 1j * hbar * dt / (4 * m * dx**2)
    main_diag = (1 + 2 * a) * np.ones(N)  # Main-diagonal elements
    off_diag = -a * np.ones(N - 1)        # Off-diagonal elements

    #Sparse matrix A and B
    A = diags([off_diag, main_diag, off_diag], [-1, 0, 1], format='csr') #Builds
        tridiagonal matrix A, which multiplies with the wavefunction at (n+1) time
        level
    B = diags([-off_diag, (1 - 2 * a) * np.ones(N) , -off_diag], [-1, 0, 1], format='
        csr') # Builds matrix B for wavefunction at n time level
    return x, psi0, A, B, steps
```

b) **Plot2D():**

- Evolves the wavefunction for 2000 steps, and
- Plots the probability density, real part, and imaginary part of Ψ at the final step.

```
# Function for 2D plots at 2000th step of the evolution
def Plot2D():
    x, psi0, A, B, steps = Start()
    psi = psi0.copy()
    for step in range(steps):
        psi = spsolve(A, B @ psi)
        psi[0] = psi[-1] = 0

    #Set limits of y-axis of probability density, real part and imaginary part
    fig, ax = plt.subplots(3, 1, figsize=(14, 18), constrained_layout=True)
    line, = ax[0].plot(x, np.abs(psi)**2, label='Probability Density')
    line_real, = ax[1].plot(x, np.real(psi), label='Real Part')
    line_imag, = ax[2].plot(x, np.imag(psi), label='Imaginary Part')
```

```
    for a in ax:
        a.set_xlabel('Position x (m)')
        a.legend()
    ax[0].set_title('Probability Density at 2000th Step')
    ax[1].set_title('Real Part of the Wavefunction at 2000th Step')
    ax[2].set_title('Imaginary Part of the Wavefunction at 2000th Step')
    plt.show()
```

c) **Plot3D():**

- Same as `Plot2D()`, creates a 3D static plot showing the final state of the wavefunction Ψ in the complex plane with real and imaginary components at y and z axes respectively.

```
# Function for 3D plots at 2000th step of the evolution
def Plot3D():
    x, psi0, A, B, steps = Start()
    psi = psi0.copy()
    for step in range(steps):
        psi = spsolve(A, B @ psi)
        psi[0] = psi[-1] = 0
    # 3D plot of the wavefunction at the final step
    fig = plt.figure(figsize=(12, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(x, np.real(psi), np.imag(psi))
    ax.set_xlabel('Position x (m)')
    ax.set_ylabel('Real Part of Psi')
    ax.set_zlabel('Imaginary Part of Psi')
    ax.set_title('3D Visualization of the Wavefunction at Final Step')
    plt.show()
```

d) **Plot2D_1():**

- Stores and plots at 500th, 1000th, 1500th and 2000th steps and,
- Visualize time evolution in discrete stages plotting the,e wavefunction's real and imaginary parts as a function of position.

```
# Function for 2D plots at 500th, 1000th, 1500th and 2000th steps
def Plot2D_1():
    x, psi0, A, B, steps = Start()
    selected_steps = [500, 1000, 1500, 2000]
    evolution = {step: {} for step in selected_steps}
    psi = psi0.copy()
    for step in range(1, steps+1):
        psi = spsolve(A, B @ psi)
        psi[0] = psi[-1] = 0
        # Dictionary named evolution
        if step in selected_steps:
            evolution[step]['prob'] = np.abs(psi)**2
            evolution[step]['real'] = np.real(psi)
            evolution[step]['imag'] = np.imag(psi)
    # Visualization of the real, imaginary parts, and probability density of the
        wavefunction at selected steps
    fig, ax = plt.subplots(3, 1, figsize=(12, 18), constrained_layout=True)
    for step in selected_steps:
        ax[0].plot(x, evolution[step]['prob'], label=f'Probability Density at Time Step
             {step}')
        ax[1].plot(x, evolution[step]['real'], label=f'Real Part at Time Step {step}')
        ax[2].plot(x, evolution[step]['imag'], label=f'Imaginary Part at Time Step {
            step}')

    ax[0].set_title('Probability Density Over Time')
    ax[0].set_xlabel('Position x (m)')
    ax[0].set_ylabel('Probability Density')
    ax[0].legend(loc='upper left', fontsize=8)
```

```
    ax[1].set_title('Real Part of the Wavefunction Over Time')
    ax[1].set_xlabel('Position x (m)')
    ax[1].set_ylabel('Real Part of Psi')
    ax[1].legend(loc='upper left', fontsize=8)

    ax[2].set_title('Imaginary Part of the Wavefunction Over Time')
    ax[2].set_xlabel('Position x (m)')
    ax[2].set_ylabel('Imaginary Part of Psi')
    ax[2].legend(loc='upper left', fontsize=8)
    plt.show()
```

e) **Plot3D_1():**

- Same as `Plot2D_1()` but in 3D plots at those time steps.

```
# Function for 3D plots at 500th, 1000th, 1500th and 2000th steps
def Plot3D_1():
    x, psi0, A, B, steps = Start()
    L = 1e-8
    selected_steps = [500, 1000, 1500, 2000]
    evolution = {step: {} for step in selected_steps}
    psi = psi0.copy()
    for step in range(1, steps+1):
        psi = spsolve(A, B @ psi)
        psi[0] = psi[-1] = 0
        # Dictionary named evolution
        if step in selected_steps:
            evolution[step]['prob'] = np.abs(psi)**2
            evolution[step]['real'] = np.real(psi)
            evolution[step]['imag'] = np.imag(psi)
    # 3D Visualization of the wavefunction at specified time steps
    fig = plt.figure(figsize=(14, 18), constrained_layout = True)
    for i, step in enumerate(selected_steps, 1):    #loops over the list and gives an
        index starting at 1
        ax = fig.add_subplot(2, 2, i, projection='3d')      # creates 3D figure with 2
            x2 grid
        ax.plot3D(x, evolution[step]['real'], evolution[step]['imag'], label=f'Time
            step {step}')
        ax.set_xlabel('Position x (m)')
        ax.set_ylabel('Real Part of Psi')
        ax.set_zlabel('Imaginary Part of Psi')
        ax.set_title(f'3D Visualization of the Wavefunction at Time Step {step}')
        ax.legend()
    plt.show()
```

f) **Animate2D():**

- Live animation showing evolution of $|\Psi|^2$, $\mathrm{Re}(\Psi)$, and $\mathrm{Im}(\Psi)$ over time.
- Uses `FuncAnimation()` from `matplotlib.animation`.

```
# Function for 2D Animation of Time evolution of Probability density and Imaginary and
    Real part of wavefunction
def Animate2D():
    x, psi0, A, B, steps = Start()
    psi = psi0.copy() #copy the initial wavefunction to another mutable and independent
         variable
    Prob0 = np.abs(psi)**2
    Real0 = np.real(psi)
    Imag0 = np.imag(psi)
    Prob_ylim = 1.1*Prob0.max()
    Real_ylim = 1.1*max(abs(Real0.min()), abs(Real0.max()))
    Imag_ylim = 1.1*max(abs(Imag0.min()), abs(Imag0.max()))

    # Time evolution
```

```
    fig, ax = plt.subplots(3,1, figsize=(12, 17), sharex=True, constrained_layout =
        True) #creates a figure and an array of 3 axes objects
    #Probability density
    line, = ax[0].plot(x, np.abs(psi)**2, label = 'Probability density', color='black')
         #Plots initial Probability density and stoes a reference line object
    ax[0].set_ylabel('|  (x, t)|  ') #label for the probability density axis
    ax[0].set_ylim(0,Prob_ylim) #sets the vertical range of the the Probabaility
        density plot
    #Real part
    line_real, = ax[1].plot(x,np.real(psi), label = 'Real part', color='red') #Plots
        initial Real part
    ax[1].set_ylabel('Re(  (x, t))') #label for the real part axis
    ax[1].set_ylim(-Real_ylim,Real_ylim) #sets the vertical range of the the Real part
        plot
    #Imaginary part
    ax[2].set_ylabel('Im(  (x, t))') #label for the imaginary part axis
    line_imag, = ax[2].plot(x,np.imag(psi), label = 'Imaginary part',color='blue') #
        Plots initial Imaginary part
    ax[2].set_ylim(-Imag_ylim, Imag_ylim) #sets the vertical range of the the Imaginary
         part plot
    ax[2].set_xlabel('Position x (m)') #label for the x-axis

    #Legends for all
    ax[0].legend(loc='upper right')
    ax[1].legend(loc='upper right')
    ax[2].legend(loc='upper right')

    fig.suptitle('1D Time Evolution of the wavefunction', fontsize=14) #Title of the
        whole figure
    def animate_2D(frame): #defines the animate function
        nonlocal psi # declares the wavefunction as global variable to modify it inside
             animate()
        psi = spsolve(A, B @ psi)   #compute the RHS of the Crank-Nicolson equation and
             update the wavefunction to the next time step and store it back in
        psi[0] = psi[-1] = 0
        line.set_ydata(np.abs(psi)**2) #updates the values of |  (x, t)|
        line_real.set_ydata(np.real(psi)) #updates the values of Re(  (x, t))
        line_imag.set_ydata(np.imag(psi)) #updates the values of Im(  (x, t))
        return line, line_real,line_imag

    # Call the animator
    ani_2D = animation.FuncAnimation(fig, animate_2D, frames=steps, interval=20, blit=
        True) #starts animation and calls the object animate_1()
    choice = int(input("Save Video? Enter '1' for Yes and '0' for No (show the
        animation) : "))
    if choice == 1:
        clear_screen()
        print("Saving the animation as a video file...")
        ani_2D.save('1D_TDSE_Particle_in_a_box (Animate2D()).mp4', writer='ffmpeg', fps
            =30) #saves the animation as a mp4 file
    else:
        print("Not saving the animation, just showing it.")
        plt.show() # Show the simulation
```

g) **Animate3D():**

- 3D animation of $\mathrm{Re}(\Psi)$ vs $\mathrm{Im}(\Psi)$ vs x over time.
- Simulates dynamic behavior of the quantum wave packet.

```
# Function for 3D Animation of Time evolution of Probability density and Imaginary and
    Real part of wavefunction
def Animate3D():
    x, psi0, A, B, steps = Start()
    L = 1e-8
    psi = psi0.copy() #copy the intial wavefunction to another mutable and independent
        variable
```

```
    Real0 = np.real(psi)
    Imag0 = np.imag(psi)
    Real_lim = 0.4*max(abs(Real0.min()), abs(Real0.max()))
    Imag_lim = 0.4*max(abs(Imag0.min()), abs(Imag0.max()))

    # Time evolution
    fig = plt.figure(figsize=(12,8), constrained_layout=True)
    ax = fig.add_subplot(111, projection='3d') #creates a figure and an array of 3 axes
         objects
    ax.set_xlabel('Position x (m)') #label for the x-axis
    ax.set_xlim(0,L) #sets the vertical range of the the Probabaility density plot
    line, = ax.plot(x, np.real(psi), np.imag(psi), lw=1.5)

    # Real part
    ax.set_ylabel('Re(  (x, t))') #label for the real part axis
    ax.set_ylim(-Real_lim,Real_lim) #sets the vertical range of the the Real part plot

    # Imaginary part
    ax.set_zlabel('Im(  (x, t))') #label for the imaginary part axis
    ax.set_zlim(-Imag_lim, Imag_lim) #sets the vertical range of the the Imaginary part
         plot
    ax.set_title('3D Time Evolution of the Wavefunction') #Title of the figure

    def animate_3D(frame): #defines the animate function
        nonlocal psi # declares the wavefunction as global variable to modify it inside
             animate()
        psi = spsolve(A, B @ psi)   #compute the RHS of the Crank-Nicolson equation and
             update the wavefunction to the next time step and store it back in
        psi[0] = psi[-1] = 0
        line.set_data_3d(x, np.real(psi), np.imag(psi)) #updates the values of Im(  (x,
             t))
        return line

    # Call the animator
    ani_3D = animation.FuncAnimation(fig, animate_3D, frames=steps, interval=20, blit=
        False) #starts animation and calls the object animate()
    choice = int(input("Save Video? Enter '1' for Yes and '0' for No (show the
        animation) : "))
    if choice == 1:
        clear_screen()
        print("Saving the animation as a video file...")
        ani_3D.save('1D_TDSE_Particle_in_a_box (Animate3D()).mp4', writer='ffmpeg', fps
            =30) #saves the animation as a mp4 file
    else:
        print("Not saving the animation, just showing it.")
        plt.show() # Show the simulation
```

To call these functions as per the requirement, there is a user-controlled loop:

```
# Call the function to display the required simulation
while True:
    clear_screen()
    p = int(input("Enter '1' to start and '0' to terminate : "))
    if p==1:
        print("What should be done:\n1. 2D plot at 2000th step\n2. 3D plot at 2000th
            step\n3. 2D plots at 500th, 1000th, 1500th and 2000th steps\n4. 2D plots at
             500th, 1000th, 1500th and 2000th steps\n5. 2D Animation\n6. 3D Animation")
        n = int(input("Enter your choice : "))
        if n==1:
            Plot2D()
        elif n==2:
            Plot3D()
        elif n==3:
            Plot2D_1()
        elif n==4:
            Plot3D_1()
        elif n==5:
            Animate2D()
```

```
        elif n==6:
            Animate3D()
        else:
            print("Wrong Choice")
    elif p==0:
        print("Terminated")
        import time
        time.sleep(2)
        import sys
        sys.exit()
    else:
        print("Wrong Choice")
```

3.3 Simulation Results

This section details the key observations from our quantum wave packet simulation, focusing both on the final moment of evolution and on intermediate stages for a broader temporal understanding.

At the 2000th time step, the real component of the wavefunction—depicted in the left panel of Figure 3.1-displays a standing wave profile, characterized by regularly spaced nodes and antinodes. This wave-like structure illustrates the fundamental quantum concept of interference within a confined region, where amplitude fluctuations arise from superposed wave modes. These oscillations represent spatial variations in the probability amplitude, indicating regions where the presence of the electron is more or less likely.

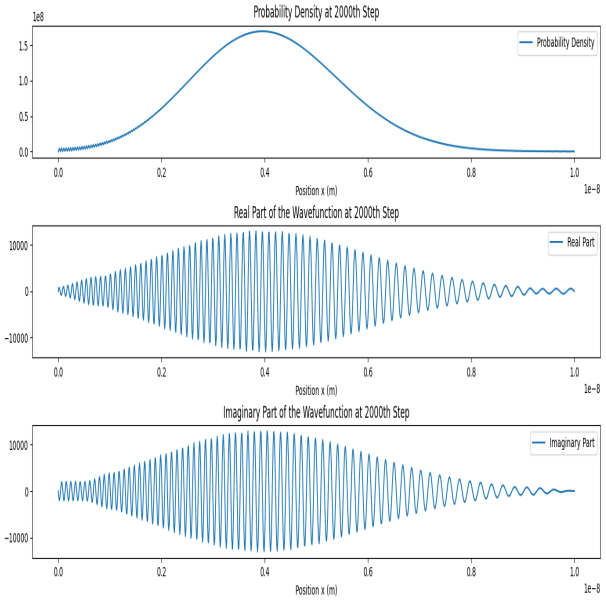
The imaginary component, shown alongside the real part, exhibits similar behavior. This is in line with the fact that quantum wavefunctions are inherently complex-valued, and both real and imaginary parts evolve together under the rules set by the time-dependent Schrödinger equation. Together, they define the complete quantum state, with their interplay governing the system's dynamics over time.

The probability density, calculated as the squared modulus of the wavefunction, offers a direct measure of the likelihood of locating the electron at various positions. In Figure 3.1, this quantity is concentrated near the center of the potential well and drops off as we approach the boundaries. This distribution matches the expected outcome for a particle confined within a 1D potential box, where the wavefunction is constrained to vanish at the edges due to boundary conditions.

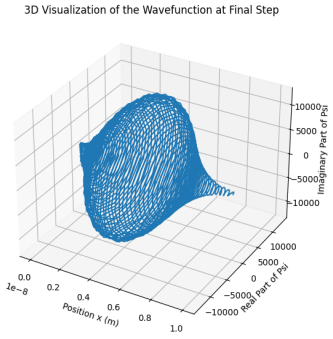
A complementary perspective is provided by the 3D visualization shown in the right panel of Figure 3.1, where the real and imaginary parts are plotted together against position. This representation captures the complex structure of the wavefunction in full, offering an intuitive view of its behavior that extends beyond flat 2D plots. The central peak remains dominant, indicating a high probability region. Notably, the wavefunction appears more delocalized at this stage, having spread across the well, which signifies a transition towards behavior more characteristic of wave propagation than classical localization.

In addition, Figure 3.2 tracks the evolution of the wavefunction across selected time steps: 500, 1000, 1500, and 2000. Early on, particularly at steps 500 and 1000, the wavefunction retains a more compact and organized form. These snapshots suggest the wave packet is initially more localized, responding to the constraints of the potential well. As the simulation advances to steps 1500 and 2000, the structure becomes more intricate and extended, reflecting the combined effects of quantum dispersion and eigenstate superposition.

Throughout this process, the real part of the wavefunction reflects the momentum distribution via its oscillatory patterns, while the imaginary part encodes essential phase information. Both are crucial, as their combination determines observable properties such as probability densities. The increasing complexity observed with time captures the rich, time-dependent dynamics of quantum systems, governed by interference among multiple energy eigenstates and boundary-driven constraints.

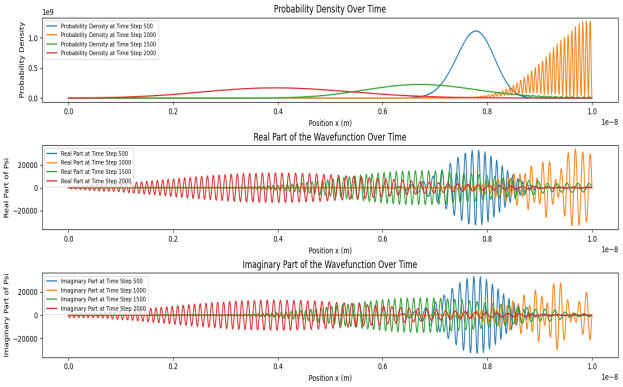


(a) 2D Plot at 2000th step of the evolution

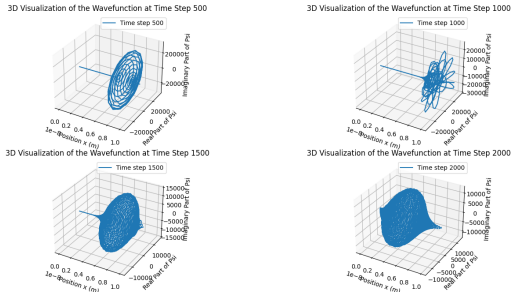


(b) 3D Plot at 2000th step of the evolution

Figure 3.1: Visualization of 2D and 3D wavefunction evolution at 2000th step.
The left panel displays 2D plots of the real part, imaginary part, and probability density of the wavefunction as a function of position at the final time step. In contrast, the right panel illustrates a 3D visualization of the real and imaginary components of the wavefunction at the same final time step.



(a) 2D Plot at selected steps



(b) 3D Plot at selected steps

Figure 3.2: The left panel presents 2D plots of the wavefunction's real part, imaginary part, and probability density as functions of position across several selected time steps. In contrast, the right panel illustrates the 3D evolution of the wavefunction by combining its real and imaginary components over those same time intervals.

The probability density plots offer deep insight into how the electron's spatial distribution within the potential well evolves over time. Initially, the electron is most likely to be found near the center of the well, consistent with the Gaussian profile of the initial wave packet. As time progresses—particularly evident at the final step (2000)—the distribution broadens and distinct peaks begin to emerge. This behavior reflects the formation of standing wave patterns, a hallmark of quantum confinement, where the wavefunction must vanish at the well's boundaries due to imposed boundary conditions.

The 3D time evolution plots across four selected time steps (500th, 1000th, 1500th, and 2000th) provide a compelling visualization of the wavefunction's dynamical behavior. Each snapshot combines the real and imaginary components into a unified three-dimensional representation. These visualizations vividly illustrate the wave-particle duality central to quantum mechanics: the electron is not confined to a fixed location but rather exists as a delocalized wave structure within the well. The progressive changes in the waveform—characterized by shifting peaks and valleys—demonstrate how the wavefunction evolves under the influence of both quantum superposition and the restrictive boundary conditions.

In addition to the static visualizations presented in this study, the time-dependent dynamic evolution of the wavepacket can also be observed, although it is not included here. Such animations can be generated using the functions `Animate2D()` and `Animate3D()`, which allow for a real-time depiction of the wavefunction's evolution throughout the simulation.

Watch the animation video (Click to watch):

- 2D Animation: Animate2D()
- 3D Animation: Animate3D()

Chapter 4

Conclusion

This study presents a comprehensive numerical simulation of the time-dependent evolution of a quantum wave packet confined within a one-dimensional infinite potential well using the **Crank–Nicolson method**. As an implicit finite-difference scheme, the Crank–Nicolson method offers second-order accuracy in both time and space and is unconditionally stable, making it particularly well-suited for long-time quantum simulations. The simulation successfully evolved the wavefunction over 2000 discrete time steps without exhibiting numerical instability or energy drift, effectively preserving the norm of the wavefunction throughout the process.

By discretizing both the spatial and temporal domains of the time-dependent Schrödinger equation (TDSE), we were able to investigate a range of quantum mechanical behaviors through both static and dynamic visualizations. The resulting 2D and 3D plots provided clear physical insights into the following quantum phenomena:

- **Standing Wave Behavior:** The real and imaginary parts of the wavefunction at later stages exhibit oscillatory structures with distinct nodes and antinodes, characteristic of standing waves arising from the system's boundary-imposed eigenstate interference.
- **Wave Packet Spreading:** The initially localized Gaussian wave packet gradually disperses over time due to the uncertainty principle and the contribution of multiple momentum eigenstates.
- **Probability Density Evolution:** The squared modulus of the wavefunction highlights the dynamic redistribution of the particle's positional probability. Initially concentrated at the center of the well, the density spreads outward, reflecting interference and confinement effects.
- **Wave–Particle Duality and Superposition:** The combined real and imaginary components, visualized in both 2D and 3D, showcase the dual wave-particle nature of quantum systems and demonstrate how coherent superposition drives the evolution of quantum states.

While this report emphasizes static visualizations, the simulation also includes dynamic animation functions—`Animate2D()` and `Animate3D()`—that can be employed to visualize the wavefunction's time evolution in real time. These animations offer a vivid and intuitive representation of quantum dynamics, making them especially useful for educational and research purposes. They help elucidate abstract features such as phase shifts, interference effects, and quantum state transitions.

The simulation confirms that the Crank–Nicolson method is not only numerically robust and efficient but also highly adaptable for future extensions, including:

- more complex potential landscapes (e.g., harmonic oscillators, barriers, and wells),
- varying boundary conditions (e.g., periodic or absorbing),
- and the modeling of quantum systems in higher spatial dimensions.

In conclusion, this project demonstrates the power of numerical methods—particularly the Crank–Nicolson scheme—in modeling the time evolution of quantum systems. The successful integration of mathematical modeling, algorithmic stability, and physical interpretation bridges the gap between theoretical quantum mechanics and computational visualization, providing a strong foundation for future simulations and research in computational quantum physics.

References

[1] D. J. Griffiths, *Introduction to Quantum Mechanics*, 2nd ed., Pearson Prentice Hall, 2005.

[2] R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*, SIAM, 2007.

[3] Adib Kabir, *Numerical Simulation of the Time-Dependent Schrödinger Equation Using the Crank–Nicolson Method*, arXiv preprint, 2024.

[4] Simon Fraser University – BUS864 Lecture Notes, *Numerically Solving PDEs*, Instructor: R. Jones.

[5] Full Source Code: *GitHub*