Alona Samoylova **12.12.2024**

Project 4 report: Numerical solutions to the time-dependent Schrödinger Equation

Introduction

This project focuses on solving the one-dimensional, time-dependent Schrödinger equation using numerical methods to model the evolution of a particle's quantum wave function. The wave function provides the probability density of finding a particle in space and time, making it a fundamental tool in quantum mechanics.

The goal is to implement and compare two numerical methods, FTCS and Crank-Nicholson, for solving the equation under different conditions, including user-defined potentials and initial states. The project also includes a visualization function to analyze results and assess numerical stability, providing a practical exploration of quantum dynamics.

1. Theory

The Schrödinger equation is a fundamental equation in quantum mechanics that describes how the quantum state of a physical system evolves over time.
For a particle of mass m in one dimension, the time-dependent Schrödinger equation may be written as:

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

where ψ(x,t) is the wave function and V(x) is the potential.

In operator notation, we may write the Schrödinger equation as:

$$i\hbar\frac{\partial\psi}{\partial t} = \hat{H}\psi$$

where $\hat{H}$ is the Hamiltonian operator.

At the same time, the formal solution of the Schrödinger equation is:

$$\psi(x,t) = e^{-\frac{i}{\hbar}\hat{H}t}\psi(x,0)$$

The **FTCS scheme** discretizes the Schrödinger equation as:

$$i\hbar\frac{\psi_j^{n+1} - \psi_j^n}{\tau} = -\frac{\hbar^2}{2m}\frac{\psi_{j+1}^n + \psi_{j-1}^n - 2\psi_j^n}{h^2} + V_j\psi_j^n$$

where $V_j \equiv V(x_j)$.

Since the Hamiltonian is a linear operator, solving for ψ_j^{n+1} gives us the numerical scheme; in matrix notation, it may be written as:

$$\Psi^{n+1} = \left(I - \frac{i\tau}{\hbar} H\right) \Psi^n \tag{1}$$

where Ψ^n is a column vector and *I* is the identity matrix. Equation (1) is the explicit FTCS scheme for solving the Schrödinger equation. It can be interpreted as using the first term in the Taylor expansion of the formal $\Psi(x, t)$ solution to advance the solution by one time step.

The disadvantage of the FTCS scheme to be numerically unstable if the time step is too large. To check the stability, we can take the maximum largest eigenvalue of the evolution matrix

$$A = I - i\tau\, \hbar^{-1} H \tag{2}$$

and compare it to 1. If the difference is small enough then the solution is expected to be stable.

This stability problem motivates to look for alternative approaches. For example, we can consider the implicit FTCS scheme is that it is ***unconditionally stable***, as may be shown using von Neumann stability analysis. While the fully implicit scheme is very appealing because of its stability, we also want a method to be accurate. Just because the solution doesn't blow up doesn't mean that it is correct. A more accurate scheme is the **Crank-Nicolson method**. Basically, it takes the average between the implicit and explicit FTCS schemes:

$$i\hbar \frac{\psi_j^{n+1} - \psi_j^n}{\tau} = \frac{1}{2} \sum_{k=0}^{N-1} H_{jk} \left(\psi_k^n + \psi_k^{n+1}\right)$$

.

After rewriting it matrix form and isolating the $\Psi^{(n+1)}$ term on the left-hand side we have:

$$\Psi^{(n+1)} = \left(I + \frac{i\tau}{2\hbar}\mathbf{H}\right)^{-1} \left(I - \frac{i\tau}{2\hbar}\mathbf{H}\right) \Psi^{(n)} \tag{3}$$

In the equation above we can let $A = I + (i\tau\,(2\hbar)^{-1})\ \mathbf{H}$ and $B = I - (i\tau\,(2\hbar)^{-1})\ \mathbf{H}$ and

obtain the Crank-Nicholson matrix: $dCN = A^{-1} * B$, useful to defining this method.

Despite its complicated appearance, the Crank-Nicholson scheme (3) is the best of the considered schemes since it is unconditionally stable and centered in both space and time.

Before writing a program to solve the Schrödinger equation, we need to think about what initial conditions. A reasonable initial condition would be the wave packet

for a particle localized about x_0 with a packet width of σ_0 and an average momentum $p_0 = \hbar k_0$ (where k_0 is the average wave number). Therefore, let's consider a Gaussian wave packet; the initial wave function is

$$\psi(x, t = 0) = \frac{1}{\sqrt{\sigma_0 \sqrt{\pi}}} \exp\left[ik_0 x - \frac{(x - x_0)^2}{2\sigma_0^2}\right] \quad (4)$$

It is important to notice that this wave function is normalized so that $\int|\psi|^2 dx = 1$ and the Gaussian wave packet has the special property that the uncertainty product $\Delta x \Delta p$ has its minimum theoretical value of $\hbar/2$. The probability density is given by $\psi\, \psi^*(x)$ or $|\psi(x)|^2$.

As the wave function evolves over time, the Gaussian wave packet initially moves to the right while gradually spreading out. When the packet exits the right edge, it re-enters on the left due to the periodic boundary conditions.

2. Development and results

Firstly, the **get_input** function is defined to streamline the process of gathering user input. It provides a safe and convenient way to accept parameters like spatial grid points, time steps, and initial conditions. The function uses *ast.literal_eval()* to parse the input and ensure that it matches the expected type (e.g., integer or list). If no input is provided, a default value is used. For example, if the user enters a string for initial such as "[10, 0, 0.5]", it is converted into a list [10, 0, 0.5]. This function improves the usability of the code, ensuring that improper inputs do not cause unexpected errors.

Then, user inputs for key simulation parameters are collected. These include the number of spatial grid points (**nspace**), the number of time steps (**ntime**), the time step size (**tau**), and the simulation **method** (FTCS or Crank-Nicholson). Users are also asked to specify the spatial grid length and the form of the potential energy (either as a single constant or a *list* representing the potential at different spatial points). Initial conditions for the wavefunction in the form a list [σ_0, x_0, k_0] (**wparam)** are asked from the user as well. These inputs allow for the simulation of diverse physical scenarios, from free particle motion to particles in potentials like wells or barriers.

Afterward, the **hamiltonian** function is implemented to construct the Hamiltonian matrix for the system. This matrix represents the kinetic and potential energy operators in the Schrödinger equation. In the considered case, $H = -\partial^2/\partial x^2 + V(x)$ if m = 0.5 and h_{bar}=1 by (9.27) from the textbook. Therefore, kinetic energy is approximated using finite differences to calculate the second spatial derivative, while the potential energy, provided by the user, is added as diagonal terms in the matrix. Periodic boundary conditions are applied by connecting the first and last spatial points, ensuring a consistent solution. The Hamiltonian is scaled with normalized constants ($\hbar = 1$ and $m = 1/2$) to simplify calculations.

To ensure numerical stability, the **spectral_radius** function is defined. This function calculates the largest absolute eigenvalue of the matrix. For the explicit FTCS method, the time step size must satisfy a stability condition based on the spectral radius. This step is crucial for avoiding numerical instabilities that could lead to non-physical results, such as a diverging wavefunction.

Then I defined the main function, **sch_eqn**, responsible for solving the time-dependent Schrödinger equation in one dimension. First, it initializes constants and creates spatial (**x_grid**) and temporal (**t_grid**) grids using *np.linspace*. The initial wavefunction is defined as a Gaussian wave packet (by Formula 4) based on the user-provided parameters for position, spread, and momentum. The wavefunction is normalized to ensure that the total probability over the entire spatial domain is equal to 1, which satisfies the fundamental requirement of quantum mechanics.

Once the wavefunction is initialized, the Hamiltonian matrix is constructed using the **hamiltonian** function. For the FTCS method, the program checks stability by computing the spectral radius. Time evolution of the wavefunction is then carried out using the specified numerical method. In the ***FTCS method*** (*def. by Formula 1),* the wavefunction is updated explicitly at each time step, while in the ***Crank-Nicholson method*** *(Formula 3),* an implicit scheme is used, requiring the solution of a matrix equation at each step. The stability of FTCS method is checked using **spectral_radius** function (evolution matrix is def. by Formula 2). The Crank-Nicholson method is unconditionally stable, making it suitable for larger time steps, whereas FTCS is simpler but requires careful attention to stability conditions.

During time evolution, the program calculates the total probability at each time step and stores it in a **prob_array**. This array ensures that the total probability remains conserved over time, which validates the numerical accuracy of the solution. However, the **prob_array** only represents the integrated probability over the entire spatial domain and not the local probability density. So, for plotting purposes, I had to compute the squared modulus of the wavefunction ($|\psi(x,t)|^2$) directly.

Once complete, the main function returns key results for further analysis and visualization. These include the wavefunction $\psi(x,t)$ at all spatial and temporal points (**psi_grid**), the spatial grid (**x_grid**), the time grid (**t_grid**), and the **prob_array** giving the total probability computed for each timestep. The modular design of the program allows users to study various quantum phenomena, such as wave packet spreading, reflection, tunneling, or probability conservation under different potentials and numerical methods.

Next, the script includes a block to handle visualization and analysis of the results. The plotting section begins with defining **sch_plot** – a function to plot $\psi(x,t)$ or the probability density at a specific time. It takes inputs such as **x_grid**, **t_grid**, **psi_grid**, and any additional parameters like the title or time intervals for plotting. Unfortunately, I could not directly use **prob_array** for plotting, as it represents the total probability across all spatial grid points at each time step, rather than the *local* probability density. So to create meaningful spatial visualizations, the code computes $|\psi(x,t)|^2$ for all grid points at selected time steps. The function loops over selected time steps, displaying the wavefunction's evolution. The results can be plotted in two ways: the real part of $\psi(x, t)$, which shows the oscillatory nature of the wavefunction, and the probability density, providing a physical interpretation of where the particle is most likely to be found at any given time. The plotting is done using matplotlib.pyplot and the obtained graphs are clearly labeled with titles, axis labels, and legends to ensure they are easy to interpret. For example, the x-axis represents the spatial grid, while the y-axis represents either the real part of the wavefunction or the probability

density. This dual visualization helps to connect the mathematical formalism of quantum mechanics to its physical interpretation.

In this final section of the script, the main simulation workflow is executed. First, the user-provided parameters are passed to the **sch_eqn** function to compute the wavefunction evolution, spatial and temporal grids, and the probability array. Once the computation is complete, the results are visualized using the **sch_plot** function. The user can specify the time steps or spatial regions of interest to customize the plots. This modular approach allows for flexible exploration of different quantum mechanical scenarios.

Moreover, the code has been designed to be easily extendable. For instance, new numerical methods (e.g., implicit schemes or higher-order Runge-Kutta methods) can be implemented by modifying the time evolution loop in the **sch_eqn** function. Similarly, the potential energy term can be adapted to simulate more complex systems, such as harmonic oscillators, double wells, or periodic potentials. This extensibility makes the script a powerful tool for studying a wide range of quantum systems.

Finally, the code concludes with optional user interaction. For example, the user may wish to modify parameters, such as the time index, and the filename of the plot. Error messages are provided in cases of invalid inputs or numerical instabilities, guiding the user toward appropriate corrections. By combining reliable numerical methods with clear visualizations and user-friendly input handling, the script ensures that any user can explore quantum dynamics effectively.

The examples of the plots constructed are given below.

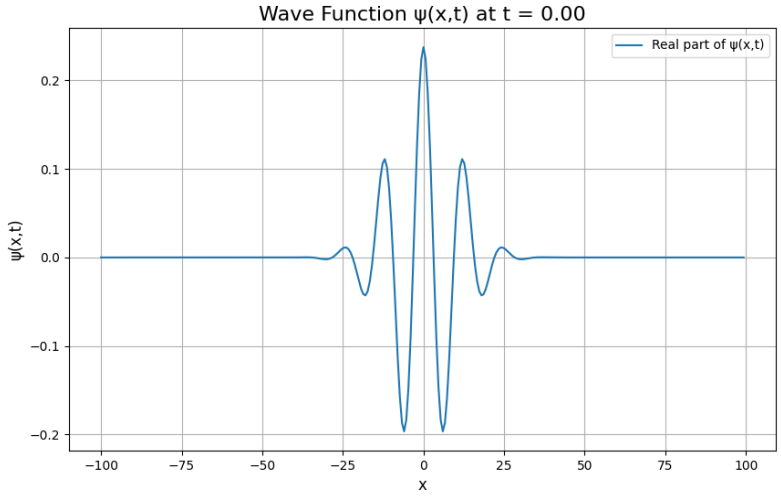

Figure 1: Wavefunction ψ(x,t) plotted at t =0 using FTCS method
for the following parameters: nspace = 300, ntime = 100, tau = 0.001, length (length of spatial grid) = 200, potential = [] (None), wparam = [10, 0, 0.5]

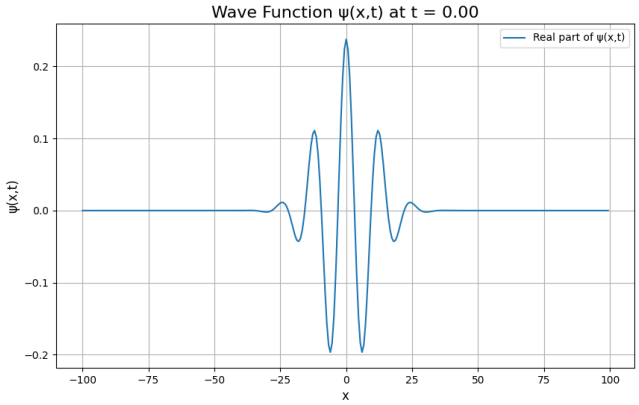

Figure 2: Wavefunction ψ(x,t) plotted at t =0 using Crank-Nicholson method for the following parameters: nspace = 300, ntime = 100, tau = 0.001, length = 200, potential = [], wparam = [10, 0, 0.5]

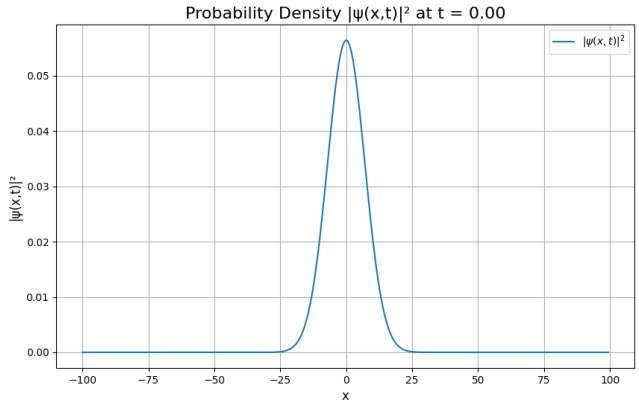

Figure 3: The probability density $|\psi(x,t)|^2$ plotted at t =0 using FTCS method for the following parameters: nspace = 300, ntime = 100, tau = 0.001, length = 200, potential = [], wparam = [10, 0, 0.5]

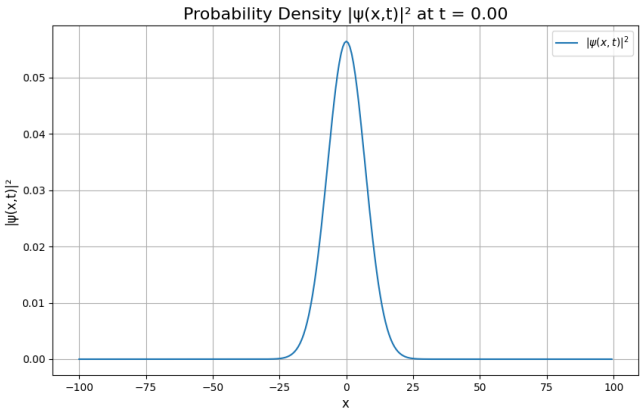

Figure 4: The probability density $|\psi(x,t)|^2$ plotted at t =0 using Crank-Nicholson method for the following parameters: nspace = 300, ntime = 100, tau = 0.001, length = 200, potential = [], wparam = [10, 0, 0.5]

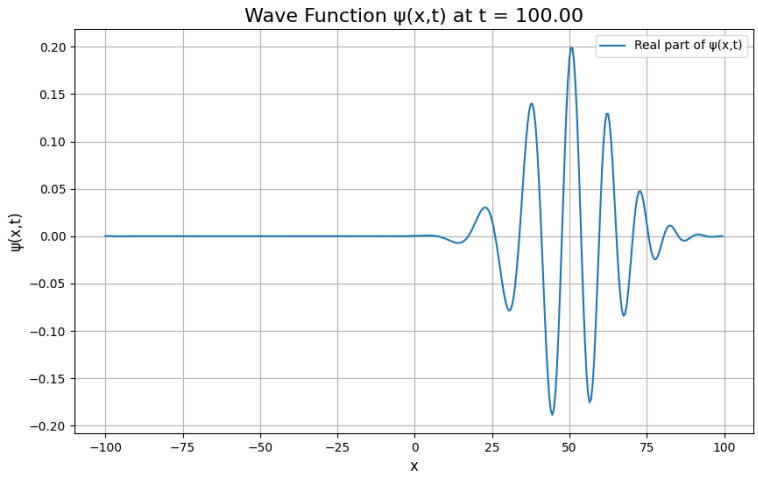

Figure 5: Wavefunction ψ(x,t) plotted at the last time step, t =100, using Crank-Nicholson method for the following parameters: nspace = 400, ntime = 1000, tau = 0.1, length = 200, potential = [], wparam = [10, 0, 0.5]

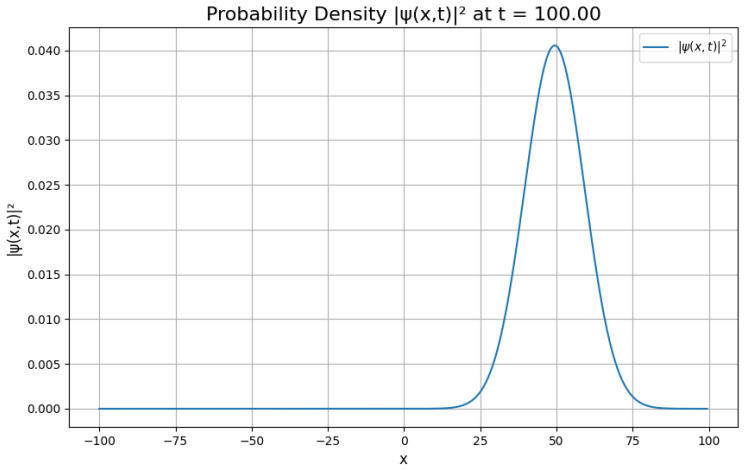

Figure 6: The probability density $|\psi(x,t)|^2$ plotted at at the last time step, t =100, using Crank-Nicholson method for the following parameters: nspace = 300, ntime = 100, tau = 0.001, length = 200, potential = [], wparam = [10, 0, 0.5]

The Figures confirm the expected behavior of the wavefunction described in the **Theory** subsection. For example, as shown in Figures 2 and 5, the wave packet initially propagates to the right while gradually spreading out.

3. User Documentation for Project 4

1. Purpose of the Code

This program solves the 1D time-dependent Schrödinger equation using two numerical methods:

- **FTCS (Forward-Time Central-Space)** scheme.
- **Crank-Nicholson** scheme.

It computes the wavefunction $\psi(x,t)$ and the probability density $|\psi(x,t)|^2$ over a spatial grid for given potential and initial conditions. The program can also visualize the results via plots.

2. Structure of the Code

The script uses the following libraries:

- `numpy` is used for array operations, numerical calculations, and matrix manipulation.
- `matplotlib.pyplot` is used to plot the results.
- `ast` helps interpret user inputs (e.g., lists entered as strings).

The code contains two separate blocks for user prompt hanging and the following functions:

- `get_input(prompt, default=None):`

Prompts the user for input and returns the entered value. If no input is given, it returns the default value instead. Supports single values (int/float), lists, and strings.

Parameters:

`prompt`**:** str
Text message displayed to the user.

`default`: int, float, str, list
Default value if the user does not input anything.

Behavior:

• If the user enters a value, it is returned.
• If the user presses Enter without input, the default value is returned.
• Handles multiple data types:
Integers, floats, or strings.
Lists (if provided as a valid list-like string)

Output: int, float, str, list
The user input as the appropriate type.

Example Usage:

```
nspace = get_input("Enter number of spatial grid points. Default is 300:", 300)
# Input: 200
# Output: 200
```

- `hamiltonian(nspace, potential=None, dx=1):`

Constructs the Hamiltonian matrix H for a free particle with the given potential.

Parameters:

`nspace` **:** int
Number of spatial grid points.

`potential`: 1D array or None
Potential values at each grid point. If None, a zero potential is assumed.

Behavior:

- Constructs Hamiltonian matrix using textbook formula (9.27) .
- The kinetic energy term is approximated with a second-order finite difference scheme:
  - Center term: − 2
  - Right neighbor: +1
  - Left neighbor: +1
- Adds potential energy terms V(x) at specified indices (to the diagonal terms).
- Scales the entire matrix by -ħ² / 2m

Output: ndarray
Real-valued Hamiltonian matrix H of size nspace × nspace.

Example Usage:

```
   H = hamiltonian(nspace, potential, dx)
Output:

[[ 4.        -2.         0.        ...  0.         0.        -2.       ]
 [-2.         3.        -2.        ...  0.         0.         0.       ]
 [ 0.        -2.         4.        ...  0.         0.         0.       ]
 ...
 [ 0.         0.         0.        ...  4.        -2.         0.       ]
 [ 0.         0.         0.        ... -2.         4.        -2.       ]
 [-2.         0.         0.        ...  0.        -2.         4.       ]]
```

- `spectral_radius(A)`:

Calculates the maximum absolute eigenvalue of a 2-D array A

Parameters:

`A`: ndarray (2D)
Input 2D array representing a square matrix.

Behavior:

- Calculates all eigenvalues of the matrix A using NumPy's linalg.eig function.
- Computes the absolute values of the eigenvalues.
- Returns the maximum absolute eigenvalue, which is the spectral radius of A.

Output: float
Maximum absolute value of the eigenvalues of A.

Example Usage: stability check

```
    eigenval = spectral_radius(A) #A is the evolution matrix
           #looking for radius
           rho = eigenval -1 #I don't have to take abs of eigenval. again as it was taken in
           if rho> 1e-6:
               #value error to terminate integration if the soln. is unstable
               raise ValueError(f"Unstable: Time step τ={tau:.4e} exceeds stability limit. Reduce τ.")
```

- `sch_eqn(nspace, ntime, tau, method='ftcs', length=200, potential=[], wparam=[10, 0, 0.5])`:

Solves the 1D time-dependent Schrödinger equation using FTCS or Crank-Nicholson scheme.

Parameters:

`nspace` : int
Number of spatial grid points.

`ntime`: int
Number of time steps for the simulation.

`tau`: float
Time step size.

`method`: str
Numerical method: 'ftcs' (default) or 'crank'.

`length` : float
Length of spatial grid: [-L/2, L/2]. Default to 200.

`potential`: list
Spatial index values at which the potential V(x) = 1. Default to empty.

`wparam`: list
Parameters for initial wave packet [sigma0, x0, k0]. Default [10, 0, 0.5].

Behavior:

1. **Initial Setup:**
   - Constructs the spatial grid and time grid.
   - Initializes the wavefunction $\psi(x,0)$ as a Gaussian wave packet (Formula 4)
   - Defines the Hamiltonian H using the function called `hamiltonian`.

2. **Numerical Method:**

   - **FTCS Method:**
     - Updates the wavefunction using the explicit scheme (Formula 1)
     - Checks numerical stability using the spectral radius of the evolution matrix.

   - **Crank-Nicholson Method:**
     - Solves the implicit Equation 3

3. **Probability normalization:**
   - Monitors the total probability across the spatial domain at each time step to ensure conservation.

Output:

Returns the wavefunction grid `psi_grid` (2-D array), spatial and temporal grids: `x_grid` (1-D array) and `t_grid` (1-D array), and the probability array `prob_array` (2-D array).

Example Usage:

```
# Parameters
nspace = 200
ntime = 500
tau = 0.01
length = 200
potential = [50, 100, 150]  # Example potential indices, random
wparam = [10, 0, 0.5]

# Solves Schrödinger equation using FTCS method
psi_grid, x_grid, t_grid, prob_array = sch_eqn(nspace, ntime, tau, method='ftcs', length=length,
potential=potential, wparam=wparam)
```

- `sch_plot(plot_type='psi', t_index=None, save_to_file=False, filename="sch_plot.png"):`

Plots the wave function $\psi(x,t)$ or the probability density $|\psi(x,t)|^2$ at a specific time.

Parameters:

`plot_type` : str
Specifies what to plot: 'psi' for the wave function or 'prob' for the probability density.

`t_index`: int or None
Time index for the plot. Defaults to the last time step if None.

`save_to_file`: bool
Whether to save the plot to a file. Default: False.

`filename`: str
The filename for saving the plot (effective only if save_to_file=True). Default: "sch_plot.png".

Behavior:

1. **Simulation Results:**
   - Calls the `sch_eqn` function to solve the Schrödinger equation and retrieve the wavefunction grid, spatial grid points, time steps, and total probability array.

2. **Time Index Validation:**
   - If t_index is None, the function defaults to the last time step.
   - Raises a ValueError if t_index is outside the valid range [0, number of time steps−1].
   -
3. **Plot Types:**
   - **Wave Function (plot_type='psi'):**
     - Plots the real part of $\psi(x,t)$ at the specified time step.
     -
   - **Probability Density (plot_type='prob'):**
     - Plots $|\psi(x,t)|^2$ (local probability density) at the specified time step.
4. **Customization:**
   - Adds a grid, legend, and appropriate labels to the plot.
   - Optionally, it saves the plot to a file.

5. **Display:**
   - Displays the plot using matplotlib.

Outputs:
None (Displays the plot)

Example Usage:

```
# plotting parameters
plot_type = 'prob'  # to lot the probability density
t_index = 100       # time step index for the plot
save_to_file = True # saves the plot as an image
filename = "prob_density_t100.png"

# generates the plot
sch_plot(plot_type=plot_type, t_index=t_index, save_to_file=save_to_file, filename=filename)
```

3. Workflow

a. Input Parameters
The program begins by prompting the user to input simulation parameters. Suggested values are displayed for reference. Users can accept the defaults by pressing Enter.

b. Running the Simulation
The Schrödinger equation is solved using the specified parameters, and the total probability is computed.

c. Visualization
After the simulation:
1. Users are asked if they want to generate a plot.
2. If yes, they specify:
   - Plot type (psi or prob).
   - Time index (specific step or last step by default).
   - Whether to save the plot and if so, the filename.

4. Example execution

Example Input Sequence:

```
Please enter number of spatial grid points. The default value is: 300. : 200
Please enter the number of time steps. The default value is: 100. :
Please enter time step. The default value is: 0.001. : 0.002
Please enter method. Type ftcs or crank. The default value is: ftcs. : crank
Please enter the length of spatial grid. The default value is: 200. :
Please enter spatial index values at which the potential V(x) = 1. The default value is: []. : [50, 100]
Please enter parameters for initial wave packet [sigma0, x0, k0]. The default value is: [10, 0, 0.5]. :
Do you want to plot the results? Please enter 'yes' to plot: yes
Please enter plot type ('psi' for wavefunction, 'prob' for probability density): prob
Please enter time index (leave blank for the last time step, enter 0 for t=0):
Do you want to save the plot to a file? (yes or no): no
```

Example Output

- **Simulation runs for 200 spatial points over 100 time steps.**
- **Probability density $|\psi(x,t)|^2$ is plotted for the last time step.**
- **No file is saved.**

5. Error Handling

- **Invalid Inputs: Prompts the user for valid input. Multiple ValueErrors are foreseen.**
- **Unstable Simulation: Stops execution if the FTCS method becomes unstable (detected via spectral radius check).**
- **Invalid Plot Parameters: Defaults to safe values (e.g., last time step, plot type = 'psi').**

6. File Output

If the user opts to save a plot, the file will be saved with the provided filename (e.g., `sch_plot.png`).

**The following User Documentation was structured in a similar manner to the scipy.integrate.solve_ivp page*

Conclusion

In this project, I successfully implemented FTCS and Crank-Nicholson methods to solve the one-dimensional, time-dependent Schrödinger equation. The FTCS method appears computationally simple, but it requires small time steps to maintain stability. In contrast, the Crank-Nicholson method is unconditionally stable and more accurate, making it the preferred approach for most simulations.

At the same time, added figures provide clear insight into the evolution of the wave function and confirm theoretical predictions, while the successful verification of probability conservation demonstrates the program's accuracy and reliability. The scripts's modular design, gives user the opportunity to define custom initial conditions allowing futher exploration of phenomena like wave packet spreading. Although the code is fully functional, it was designed to be flexible, with the potential for future extensions such as the addition of higher-order integration methods or more complex potentials, thereby expanding its capacity as a powerful tool for visualizing and studying quantum dynamics.

In conclusion, this report describes the effective solution of the Schrödinger equation using reliable numerical methods. The code successfully models the time evolution of wavefunctions under various potentials. Moreover, the results I've achieved have deepened my understanding of quantum behavior and laid the foundation for exploring more advanced topics, paving the way for further studies in the field of quantum mechanics.

References

1) Alejandro L. Garcia. Numerical Methods for Physics (Python), 2017

Version Conrol

The code is published in the following repository:
[URL]: https://github.com/AlonaSamoylova/Project4

The screenshot of commit history is provided below.