Computational Physics - Exercise 8: Time-Dependent Schrödinger Equation

Torben Steegmann Chih-Chun Kuo

Introduction

Whenever a particle encounters a barrier in classical mechanics, it has no chance of being found on the other side if its kinetic energy is less than the potential energy of the barrier. However, in quantum theory, there is a nonzero probability for the particle to tunnel through the barrier, an effect known as quantum tunneling. This phenomenon can be described and calculated using the Time-Dependent Schrödinger Equation (TDSE).

In this exercise we are going to solve the TDSE numerically with the product formular approach.

Simulation model and method

Background

Quantum mechanics is a framework for understanding the microscopic world. Unlike classical physics, which deals with macroscopic phenomena in a deterministic manner, quantum mechanics reveals that particles exhibit both wave-like and particle-like properties. Central to this theory is the concept that particles exist in a superposition of states, described by a wave function, until an observation or measurement forces them into a specific state.

In classical physics, energy and momentum are time-dependent real numbers. In the Schrödinger picture of quantum mechanics, all information about a particle is contained in its wave function. Dynamical variables like energy and momentum are derived from this wave function using time-independent operators. Thus, classical time-dependent variables are replaced by operators acting on the time-dependent wave function in quantum mechanics.

In classical dynamics, the total energy of a non-relativistic particle can be expressed as the sum of its kinetic and potential energy,

$$E = H = T + V = \frac{p^2}{2m} + V,$$

where H is the Hamiltonian. The quantum mechanical counterpart can be obtained by replacing the momentum and the energy with their corresponding operators, in natural unit ($\hbar = 1$)

$$\hat{\boldsymbol{p}} = -i\nabla, \hat{E} = i\frac{\partial}{\partial t}.$$

This gives rise to the time-dependent Schrödinger equation,

$$i\frac{\partial \Phi(\boldsymbol{x}, t)}{\partial t} = \widehat{H}\Phi(\boldsymbol{x}, t) = \left(\frac{-1}{2m}\nabla^2 + \hat{V}\right)\Phi(\boldsymbol{x}, t).$$

For a one-dimensional system this reduces to

$$i\frac{\partial \Phi(x, t)}{\partial t} = \frac{-1}{2m}\frac{\partial^2 \Phi(\boldsymbol{x}, t)}{\partial x^2} + \hat{V}\Phi(\boldsymbol{x}, t).$$

The physical interpretation of the wave function $\Phi(x, t)$ is that $\Phi^*\Phi dx$ represents the probability of finding the particle in the infinitesimal interval dx. This leads to the probability density being defined as

$$P(x, t) = \Phi^*(x, t)\Phi(x, t) = |\Phi(x, t)|^2.$$

When we place a potential barrier that is higher than the kinetic energy of a particle, classical physics predicts that the particle cannot overcome the barrier. However, in quantum mechanics, the particle has a finite probability of "tunneling" through this barrier due to the wave-like nature of its existence. This tunneling effect has practical applications in various fields. For instance, scanning tunneling microscopy (STM) relies on quantum tunneling to image surfaces at the atomic level, providing information about the arrangement and behavior of atoms. In quantum chromodynamics (QCD), instantons are non-per-

turbative solutions to the QCD equations that represent tunneling events, contributing to phenomena like the spontaneous breaking of chiral symmetry and therefore helping to explain the confinement mechanism between quarks and hadrons.

The formal solution can be written as

$$\Phi(x,t) = e^{-itH}\Phi(x,t=0).$$

In order to calculate the solution numerically, we express the spatial derivative with the finite difference approximation

$$\frac{\partial^2\Phi(x,t)}{\partial x^2} \approx \frac{\Phi(x+\Delta,t) - 2\Phi(x,t) + \Phi(x-\Delta,t)}{\Delta^2},$$

where Δ is the spatial resolution. Then we discretize the wave function spatially with our grid size L

$$\Phi(t) = \begin{pmatrix} \Phi_1(t) \\ \Phi_2(t) \\ \vdots \\ \Phi_{L(t)} \end{pmatrix},$$

where $\Phi_i(t)$ is the wave function at position i.

Let m=1, the Schrödinger equation can then be expressed in matrix form,

$$i\frac{\partial}{\partial t}\begin{pmatrix} \Phi_1(t) \\ \Phi_2(t) \\ \Phi_3(t) \\ \vdots \\ \vdots \\ \Phi_L(t) \end{pmatrix} = \underbrace{\Delta^{-2}\begin{pmatrix} 1+\Delta^2V_1 & -1/2 & 0 & & & 0 \\ -1/2 & 1+\Delta^2V_2 & -1/2 & & & \\ 0 & -1/2 & 1+\Delta^2V_3 & & & \\ & & & \ddots & & 0 \\ & & & & 1+\Delta^2V_{L-1} & -1/2 \\ 0 & & & 0 & -1/2 & 1+\Delta^2V_L \end{pmatrix}}_{H}\begin{pmatrix} \Phi_1(t) \\ \Phi_2(t) \\ \Phi_3(t) \\ \vdots \\ \vdots \\ \Phi_L(t) \end{pmatrix}$$

. Then we decompose H into three block-diagonal matrices for computational calculation, which are

$$H = V + K_1 + K_2 = \Delta^{-2}\begin{pmatrix} 1+\Delta^2 V_1 & 0 & 0 & & & 0 \\ 0 & 1+\Delta^2 V_2 & 0 & & & \\ 0 & 0 & 1+\Delta^2 V_3 & & & \\ & & & \ddots & & 0 \\ & & & & 1+\Delta^2 V_{L-1} & 0 \\ 0 & & & 0 & 0 & 1+\Delta^2 V_L \end{pmatrix}$$

$$+\Delta^{-2}\begin{pmatrix} 0 & -1/2 & 0 & & & 0 \\ -1/2 & 0 & 0 & & & \\ 0 & 0 & 0 & & & \\ & & & \ddots & -1/2 & 0 \\ & & & -1/2 & 0 & 0 \\ 0 & & & 0 & 0 & 0 \end{pmatrix} + \Delta^{-2}\begin{pmatrix} 0 & 0 & 0 & & & 0 \\ 0 & 0 & -1/2 & & & \\ 0 & -1/2 & 0 & & & \\ & & & \ddots & & 0 \\ & & & & 0 & -1/2 \\ 0 & & & 0 & -1/2 & 0 \end{pmatrix}$$

. Therefore,

$$e^{-i\tau H} \approx e^{-i\tau\frac{K_1}{2}} e^{-i\tau\frac{K_2}{2}} e^{-i\tau V} e^{-i\tau\frac{K_2}{2}} e^{-i\tau\frac{K_1}{2}}$$

, where τ is the temporal resolution. We can then obtain the result by the formal solution

$$\Phi(t+\tau) = e^{-i\tau H}\Phi(t).$$

Calculating the exponentials of each matrix gives

$$e^{-i\tau K_1/2} = \begin{pmatrix} c & is & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ is & c & 0 & 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & c & is & 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & is & c & 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 & c & is & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & is & c & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 & \ddots & is & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & is & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix} \quad e^{-i\tau K_2/2} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & c & is & 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & is & c & 0 & 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 0 & c & is & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & is & c & 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 & c & is & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & is & \ddots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 & \ddots & is \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & is & c \end{pmatrix}$$

, where $c = \cos\left(\frac{\tau}{4\Delta^2}\right)$ and $s = \sin\left(\frac{\tau}{4\Delta^2}\right)$. Since the V matrix is diagonal, its exponential is obtained by taking the exponential of each diagonal element and placing these values on the diagonal of a new matrix.

The time evolution of wave function is computed at each discrete time step by multiplying $\Phi(t)$ with these sub-matrices, as done in exercise 7. Briefly, we take the pair of Φ_i and Φ_{i+1} and multiply them by 2x2 repetitive matrices appearing in K_1 and K_2.

Implementation

The implementation closely follows the described model. We begin by defining the constants given for the simulation. The values of the parameters are as follows: $m = 1, \hbar = 1, \Delta = 0.1, L = 1001, \tau = 0.001$. We also define a macro to easily control the existence of the barrier V(x)=2 at $50 \leq x \leq 50.5$. Then we initialize the wave function with $\Phi(x, t = 0) = \left(2\pi\sigma^2\right)^{-\frac{1}{4}} e^{iq(x-x_0)} e^{-\frac{(x-x_0)^2}{4\sigma^2}}$, which is a Gaussian wave packet centered around $x_0 = 20$ with a width $\sigma = 3$ and wave vector q=1. The algorithm is implemented as the function `TDSE()`. Since these values do not change, we can precompute the values for 'c' and 'is' as described in the product formula. After this we call 'Phi_0' to calculate the initial value of Φ as described in the exercise. Now follows the main loop of the program where we iterate over 50000 time-steps. It is important to note that we set $m = 50001$ as we have to consider the initial time-step for $t = 0$. For each element of the grid we then update Φ according to the product formula. As a last step for each time-step we save the results in a matrix. This is only done for easy plotting and should not be done in larger simulations as the resulting matrix can become too big.

Before plotting we need to calculate the probabilities as $|\Phi(x, t)|^2$. Here we also need to account for the chosen spacial resolution Δ, thus the probability distribution is calculated as $|\Phi(x, t)|^2 * \Delta$. Because the probability for a part of the wave to tunnel is low, we also sum up the total probabilities after the barrier. We use this to normalize the probabilities after the barrier as 'probabilities /= post_max', where 'post_max' is the maximum total sum of probabilities among the measured times 't' after the barrier. Lastly, we plot the results.

Simulation results

The resulting graphs of the described program can be found in Figure 4 and Figure 5.

In Figure 4 we see that: first, the wave propagates to the right since q>0; secondly, distribution gets wider and the peak value decreases over time due to the growing uncertainty of the particle. While the maximum probability starts at $P \approx 0.0135$ at $t = 0$ it already decreases to $P \approx 0.0125$ at $t = 5$. Because there is no potential barrier yet, over time it further decreases to $P \approx 0.0045$ at $t = 50$. We also see that the peak moves at constant speed.

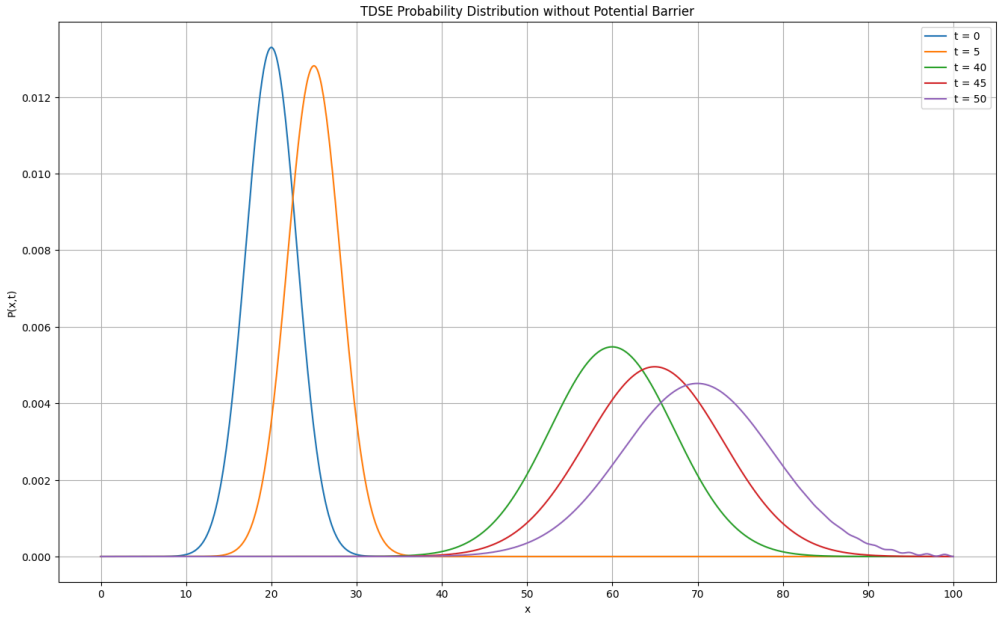

Figure 4: This graph illustrates the probability distribution according to the numerical solution of the TDSE without a potential barrier. The x-axis is the position of the wave. The y-axis is the probability of the wave to be at a position x at time t. Different snapshots were taken at different times t, ranging from t=0 (blue) to t=50 (purple).

In Figure 5 we see identical results as in Figure 4 for $t = 0$ and $t = 5$ because the wave did not yet reach the potential barrier. For $t = 40$, the wave packet did reach the barrier. We can see that the probability

of the wave packet to be found on the left side of the barrier is fluctuating with a high frequency due to the interference of the incident wave and reflected wave. This continues for times $t = 45$ and $t = 50$ however, at a reduced frequency. We can also see that the probability of the wave packet to be found on the right side of the barrier is non-zero. It is important to mention, again, that the probability is normalized here, as it would otherwise be hard to analyze the behavior as the real probabilities are much lower. We can see that after the barrier, the wave that tunnel through keeps propagating to the right.

In comparison to Figure 4, the peak probabilities of the parts of the wave packets in Figure 4 that reach past the barrier are found further along the x-axis. As an example, the maximum probability past the barrier in Figure 5 at $t = 40$ can be found at $x \approx 63$. This is in contrast to the maximum probability in Figure 4, where it can be found $x \approx 60$. This increase in speed also holds for further time-steps. This is because we initialize the wave function with Gaussian distribution, which can be seen as a superposition of waves of different momenta. Since the probability of tunneling through a barrier increases with higher energies and momenta, the barrier acts as a high-momentum pass filter for the wave packet. The partial waves that successfully tunnel through have, on average, higher momentum, resulting in a higher speed of the transmitted wave packet.

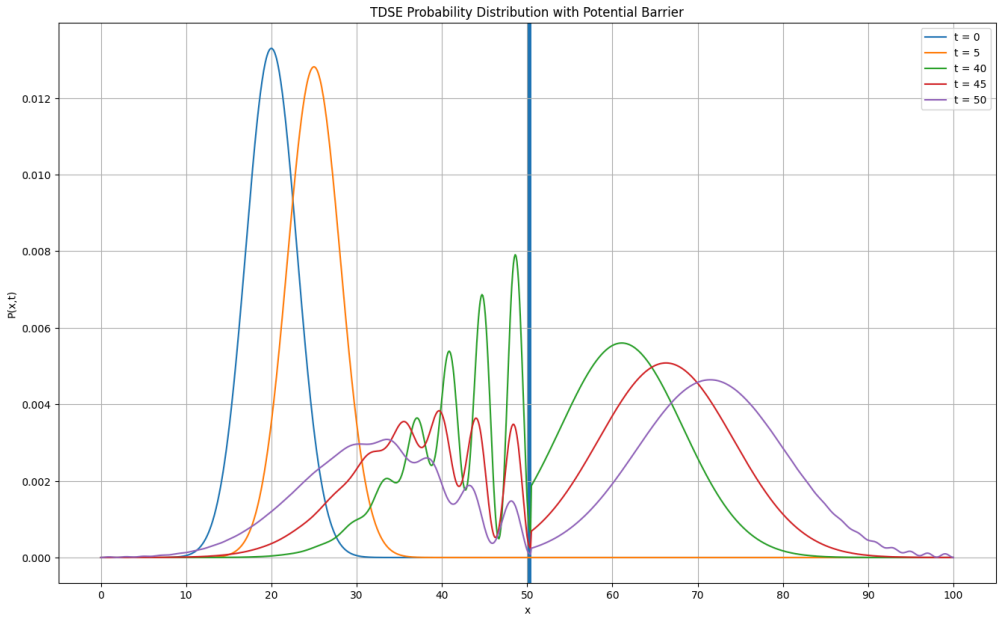

Figure 5: This graph illustrates the probability distribution according to the numerical solution of the TDSE without a potential barrier. The x-axis is the position of the wave. The y-axis is the probability of the wave to be at a position x at time t. Different snapshots were taken at different times t, ranging from t=0 (blue) to t=50 (purple).

Discussion

In the experimental results, we observed two main phenomena: the high fluctuation of probability on the left side of the barrier, and the increase in speed on the right side of the barrier.

The high fluctuation observed on the left side of the barrier is caused by the interference of the reflected wave with itself. This interference results from the wave packet encountering the potential barrier, leading to a superposition of the incoming and reflected waves, which causes the observed oscillations in the probability density.

The increase in speed on the right side of the barrier is that, the part of the wave that does tunnel through the potential barrier has more kinetic energy than the part that does not, thus it also has higher speed.

In conclusion, the product formula approach for numerically solving the TDSE proves to be a powerful method for simulating quantum dynamics. The phenomena observed in our simulations, including the interference-induced fluctuations and the increased speed of the transmitted wave, highlight the complex nature of quantum systems and the effectiveness of numerical methods in studying them.

Appendix

```cpp
#define _USE_MATH_DEFINES
#include <cmath>
#include <complex>
#include <iostream>
#include <vector>

#define WITHOUT_NUMPY
#include "matplotlibcpp.h"

using namespace std::complex_literals;
typedef std::complex<double> complex;

#define BARRIER 1 // Controll whether or not a Barrier is used

namespace e1
{
// Configuration
double const pi = M_PI;
double const sigma = 3.;
double const sigma_squared = 9.;
double const x_0 = 20.;   // starting position
double const q = 1;       // direction
double const Delta = 0.1; // spacial resolution
double const tau = 0.001; // temporal resolution
double const M = 1.;
double const h = 1.;
double const barrier_start = 50.;
double const barrier_end = 50.5;
int const timesteps = 50001;
int const L = 1001;

// V(x) barrier function calculation
double V(float x)
{
#if !BARRIER
    return 0;
#endif
    if (x >= barrier_start && x <= barrier_end)
    {
        return 2.;
    }
    return 0.;
}

// Calculate Starting Value
std::vector<complex> Phi_0()
{
```

```
    std::vector<complex> positions(L);
    double x = 0.;
    for (int i = 0; i < L; ++i)
    {
        double x_Delta = x - x_0;
         double const prefix = std::pow(2 * pi * sigma_squared, -0.25) * std::exp(-(x_Delta *
x_Delta) / (4 * sigma_squared));

        double a = prefix * std::cos(q * x_Delta);
        double b = prefix * std::sin(q * x_Delta);

        positions[i] = a + 1i * b;

        x += Delta;
    }

    return positions;
}

// Main TDSE function
void TDSE()
{
    complex const c = std::cos(tau / (4 * Delta * Delta));
    complex const is = 1i * std::sin(tau / (4 * Delta * Delta));
    std::vector<complex> positions(L);
    std::vector<std::vector<complex>> time(timesteps, positions);

    time[0] = Phi_0();                  // Set Phi(x, t=0)
    for (int i = 1; i < timesteps; ++i) // Loop over all timesteps
    {
         for (int j = 0; j < L; j += 2) // Loop over all grid points for 1. part of product
formula
        {
            positions[j] = time[i - 1][j] * c + is * time[i - 1][j + 1];
            positions[j + 1] = time[i - 1][j] * is + c * time[i - 1][j + 1];
        }
        positions[L - 1] = time[i - 1][0]; // K1

        positions[0] *= 1;
         for (int j = 1; j < L; j += 2) // Loop over all grid points for 2. part of product
formula
        {
            complex pj = positions[j] * c + is * positions[j + 1];
            complex pj1 = positions[j] * is + c * positions[j + 1];
            positions[j] = pj;
            positions[j + 1] = pj1;
        } // K2

        double x = 0.;
        for (int j = 0; j < L; ++j) // Loop over all grid points for 3. part of product formula
        {
            complex v_pot = -(1i * tau * (1. + (Delta * Delta) * V(x)) / (Delta * Delta));
            complex e_v = std::exp(v_pot);

            positions[j] *= e_v;
            x += Delta;
        } // V

        positions[0] *= 1;
         for (int j = 1; j < L; j += 2) // Loop over all grid points for 4. part of product
formula
        {
```

```
            complex pj = positions[j] * c + is * positions[j + 1];
            complex pj1 = positions[j] * is + c * positions[j + 1];
            positions[j] = pj;
            positions[j + 1] = pj1;
        } // K2

         for (int j = 0; j < L; j += 2) // Loop over all grid points for 5. part of product
formula
        {
            complex pj = positions[j] * c + is * positions[j + 1];
            complex pj1 = positions[j] * is + c * positions[j + 1];
            positions[j] = pj;
            positions[j + 1] = pj1;
        }
        positions[L - 1] *= 1; // K1

         time[i] = positions; // We only store this for simpler plotting, we could and should
not store the entire matrix for real simulations
    }

    // plotting starts here
    namespace plt = matplotlibcpp;

    std::vector<double> probabilities_0(L);
    std::vector<double> probabilities_1(L);
    std::vector<double> probabilities_2(L);
    std::vector<double> probabilities_3(L);
    std::vector<double> probabilities_4(L);
    double post_barrier_0 = 0.;
    double post_barrier_1 = 0.;
    double post_barrier_2 = 0.;
    double post_barrier_3 = 0.;
    double post_barrier_4 = 0.;

    for (int i = 0; i < L; ++i)
    {
            probabilities_0[i] = std::sqrt(std::pow(time[0][i].real(), 2) + std::pow(time[0]
[i].imag(), 2));
           probabilities_0[i] *= std::sqrt(std::pow(time[0][i].real(), 2) + std::pow(time[0]
[i].imag(), 2));
        probabilities_0[i] *= Delta;

       probabilities_1[i] = std::sqrt(std::pow(time[(5 / tau)][i].real(), 2) + std::pow(time[(5 /
tau)][i].imag(), 2));
      probabilities_1[i] *= std::sqrt(std::pow(time[(5 / tau)][i].real(), 2) + std::pow(time[(5 /
tau)][i].imag(), 2));
        probabilities_1[i] *= Delta;

                probabilities_2[i] = std::sqrt(std::pow(time[(40 / tau)][i].real(), 2) +
std::pow(time[(40 / tau)][i].imag(), 2));
               probabilities_2[i] *= std::sqrt(std::pow(time[(40 / tau)][i].real(), 2) +
std::pow(time[(40 / tau)][i].imag(), 2));
        probabilities_2[i] *= Delta;

                probabilities_3[i] = std::sqrt(std::pow(time[(45 / tau)][i].real(), 2) +
std::pow(time[(45 / tau)][i].imag(), 2));
               probabilities_3[i] *= std::sqrt(std::pow(time[(45 / tau)][i].real(), 2) +
std::pow(time[(45 / tau)][i].imag(), 2));
        probabilities_3[i] *= Delta;

                probabilities_4[i] = std::sqrt(std::pow(time[(50 / tau)][i].real(), 2) +
std::pow(time[(50 / tau)][i].imag(), 2));
```

```
                probabilities_4[i]  *=  std::sqrt(std::pow(time[(50  /  tau)][i].real(),  2)  +
std::pow(time[(50 / tau)][i].imag(), 2));
        probabilities_4[i] *= Delta;

        if (i * Delta < barrier_end)
        {
            continue;
        }
        post_barrier_0 += probabilities_0[i];
        post_barrier_1 += probabilities_1[i];
        post_barrier_2 += probabilities_2[i];
        post_barrier_3 += probabilities_3[i];
        post_barrier_4 += probabilities_4[i];
    }

#if BARRIER
    // normalize probabilities by maximum of all probabilities post barrier
   double post_max = std::max(post_barrier_0, std::max(post_barrier_1, std::max(post_barrier_2,
std::max(post_barrier_3, post_barrier_4))));
    for (int i = 0; i < L; ++i)
    {
        if (i * Delta < barrier_end)
            continue;

        probabilities_0[i] /= post_max;
        probabilities_1[i] /= post_max;
        probabilities_2[i] /= post_max;
        probabilities_3[i] /= post_max;
        probabilities_4[i] /= post_max;
    }
#endif

    plt::plot(probabilities_0, {{"label", "t = 0"}});
    plt::plot(probabilities_1, {{"label", "t = 5"}});
    plt::plot(probabilities_2, {{"label", "t = 40"}});
    plt::plot(probabilities_3, {{"label", "t = 45"}});
    plt::plot(probabilities_4, {{"label", "t = 50"}});

    plt::title("TDSE Probability Distribution without Potential Barrier");
#if BARRIER

    plt::title("TDSE Probability Distribution with Potential Barrier");
    plt::axvspan(barrier_start / Delta, barrier_end / Delta);
#endif
    plt::xlabel("x");
    plt::ylabel("P(x,t)");
    std::vector<double> ticks = {0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
     std::vector<std::string> labels = {"0", "10", "20", "30", "40", "50", "60", "70", "80",
"90", "100"};
    plt::xticks(ticks, labels); // Set custom ticks and labels
    plt::grid(true);
    plt::legend();
    plt::show();
}

} // namespace e1

int main()
{
    e1::TDSE();
```

```
    return 0;
}
```