Computational Physics Exercise 8: Time-Dependent Schrödinger Equation

Fynn Janssen 411556[1] and Tamilarasan Ketheeswaran 411069[2]

April 2023

Contents

[1]fynn.janssen@rwth-aachen.de

[2]tamilarasan.ketheswaran@rwth-aachen.de

1 Introduction

In this exercise, we solve the time-dependent Schrödinger equation using the second-order product formula algorithm. At first, we consider a system containing only a free particle without any potential. After that, we introduce a square potential with a higher potential barrier than the system's kinetic energy. In this case, we investigate quantum tunneling and compare it to the first system.

We use the programming language `Python` and the packages `numpy`[1] and to plot our results `matplotlib`. Moreover, from `numba` we import `njit` for a faster simulation.

2 Simulation Model and Method

In this task, we have a free particle propagating along the x-axis with wavenumber q. Initially, the particle is described the Gaussian wave packet

$$\Phi(x,t=0) = \frac{1}{(2\pi\sigma^2)^{1/4}} e^{iq(x-x_0)} e^{-\frac{(x-x_0)^2}{4\sigma^2}}, \tag{1}$$

centered around x_0, with width σ. The units have been chosen, such that the mass of the particle equals $M = \hbar = 1$. For such a system the time-dependent Schrödinger equation can be written as

$$i\frac{\partial}{\partial t}\Phi(x,t) = \underbrace{\left(-\frac{1}{2}\frac{\partial^2}{\partial x^2} + V(x)\right)}_{=H}\Phi(x,t), \tag{2}$$

with $V(x)$ the potential in the system. Solving this system efficiently, we restrict ourselves to calculate the wave function only for $0 \leq x \leq 100$ and discretize the position according to

$$x = (0, \Delta, 2\Delta, ..., (L-2)\Delta, (L-1)\Delta), \quad \text{with} \quad L = 1001. \tag{3}$$

Hence, we have L discrete grid points and the spatial resolution Δ. This can be implemented into python using

```
x = np.linspace (0,(L-1)*Delta,L).
```

At any time the total wave function is given by

$$\Phi(t) = \begin{pmatrix} \Phi(0,t) \\ \Phi(1,t) \\ \vdots \\ \Phi(L-1,t) \end{pmatrix} = \begin{pmatrix} \Phi_0(t) \\ \Phi_1(t) \\ \vdots \\ \Phi_{L-1}(t) \end{pmatrix}, \tag{4}$$

where $\phi(j,t)$ is the value of the wave function at position $x = j\Delta$ and time t. For the initial state at $t = 0$, we use eq.(1) and calculate all the values for the positions defined in `x`. In addition to the position, the time is also discretized as

$$t = n \cdot \tau \tag{5}$$

with n ranging from 0 to $m = 50000$. Using the discretization discussed, in eq.(2), the terms on the right hand side can be discretized as

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

and

$$V(x)\Phi(x,t) \longrightarrow V(x=j\Delta)\Phi(x,t). \tag{6}$$

[1]abbreviated as `np`

With this discretization of the Schrödinger equation and using eq.(4), eq.(2) can be written in matrix notation as with $\Phi_j(t) = \Phi(j\Delta, t)$ and $V_j = V(j\Delta)$.

$$i\frac{\partial}{\partial t}\begin{pmatrix}\Phi_0(t)\\ \Phi_1(t)\\ \Phi_2(t)\\ \vdots\\ \vdots\\ \Phi_{L-1}(t)\end{pmatrix} = \Delta^{-2}\underbrace{\begin{pmatrix}1+\Delta^2V_0 & -1/2 & 0 & & & 0\\ -1/2 & 1+\Delta^2V_1 & -1/2 & & & \\ 0 & -1/2 & 1+\Delta^2V_2 & & & \\ & & & \ddots & & 0\\ & & & & 1+\Delta^2V_{L-2} & -1/2\\ 0 & & & 0 & -1/2 & 1+\Delta^2V_{L-1}\end{pmatrix}\begin{pmatrix}\Phi_0(t)\\ \Phi_1(t)\\ \Phi_2(t)\\ \vdots\\ \vdots\\ \Phi_{L-1}(t)\end{pmatrix}}_{H}$$

source: Computational Physics – Lecture 17: Time-dependent Schrödinger equation I (page 39)(edited)

Instead of discretizing the time derivative in the equation above, we make use of the property that the general solution to eq.(2) is given by

$$\Phi(x,t) = e^{-itH}\Phi(x, t=0). \tag{7}$$

Hence, our key objective is to find a simple possibility to calculate e^{-itH}. Our approach is to decompose H into a sum of different matrices and then calculate the exponential using the product formula approach. For that we decompose the H into three different matrices V, K_1, K_2, with

$$H = V + K_1 + K_2 = \Delta^{-2}\begin{pmatrix}1+\Delta^2V_0 & 0 & 0 & & & 0\\ 0 & 1+\Delta^2V_1 & 0 & & & \\ 0 & 0 & 1+\Delta^2V_2 & & & \\ & & & \ddots & & 0\\ & & & & 1+\Delta^2V_{L-2} & 0\\ 0 & & & 0 & 0 & 1+\Delta^2V_{L-1}\end{pmatrix}$$

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

source: Computational Physics – Lecture 17: Time-dependent Schrödinger equation I (page 49)(edited)

Now, to calculate the e^{-itH}, we make use of the second-order product formula approximation telling us

$$e^{-itH} \approx \left(e^{-i\tau\frac{K_1}{2}}e^{-i\tau\frac{K_2}{2}}e^{-i\tau V}e^{-i\tau\frac{K_2}{2}}e^{-i\tau\frac{K_1}{2}}\right)^n. \tag{8}$$

Here, we can clearly see that $e^{-i\tau V}$ is the simplest of all, being a diagonal matrix with elements that are the exponentials of the diagonal elements. Hence calculating[2]

$$e^{-i\tau V}\Psi(x,t) = \mathrm{diag}(e^{-i\tau V_0}, e^{-i\tau V_1}, e^{-i\tau V_2}, ..., e^{-i\tau V_{L-1}})\Psi(x,t)$$

[2] Ψ is an arbitrary array with the same dimensions as Φ

is done by

$$\exp\left(-i\tau\underbrace{\frac{1}{\Delta^2}(1+\Delta^2V(x=j\Delta))}_{V_j}\right)\Psi(j\Delta,t), \tag{9}$$

with j ranging from 0 to $L-1$. This is a simple elementwise multiplication of two arrays. We can implement this into python by creating an array consisting $\exp\left(-i\frac{\tau}{\Delta^2}(1+\Delta^2V)\right)$ at all positions, and simply multiplying this with our array for $\Psi(x,t)$. Then each entry in Ψ will be multiplied with and only the corresponding value at the same position. Calculating

$$e^{-i\frac{\tau}{2}K_{1/2}}\Psi(x,t), \tag{10}$$

however, is more complicated due to the form of ${K_{1/2}}$[3]. Fortunately, both K_1 and K_2 are composed of 2x2 matrices along the diagonal, while for the K_1 the last entry is zero, and for the K_2, the first entry is zero. This is the reason why we only have to separate between the cases

$$\exp\left(+i\frac{\tau}{4\Delta^2}\begin{pmatrix}0&1\\1&0\end{pmatrix}\right)\begin{pmatrix}\Phi(k,t)\\\Phi(k+1,t)\end{pmatrix} \tag{11}$$

and

$$\exp\left(+i\frac{\tau}{4\Delta^2}\cdot 0\right)\Phi(k,t)=\Phi(k,t). \tag{12}$$

It is easy to show that eq.(11) can be rewritten into

$$\exp\left(+i\frac{\tau}{4\Delta^2}\begin{pmatrix}0&1\\1&0\end{pmatrix}\right)\begin{pmatrix}\Phi(k,t)\\\Phi(k+1,t)\end{pmatrix}=\underbrace{\begin{pmatrix}\cos(a)&i\sin(a)\\i\sin(a)&\cos(a)\end{pmatrix}}_{\texttt{eK12}}\begin{pmatrix}\Phi(k,t)\\\Phi(k+1,t)\end{pmatrix}, \tag{13}$$

with $a=\frac{\tau}{4\Delta^2}$. For K_1, k goes from 0 to $L-2$ (only the even numbers), and for K_2 from 1 to $L-1$ (only the odd numbers)[4]. In any case, this is a simple multiplication of a 2x2 matrix with a vector. This can be realized in python using `np.dot(a,b)`, which yields the product of `a` and `b`. Since k does not take all the numbers, but only the even ones for K_1 and the odd ones for K_2, we use `for-loops` with the range defined in the functions below

```python
eK12     = np.array([[np.cos(a), 1j*np.sin(a)], [1j*np.sin(a), np.cos(a)]], dtype=np.
    complex128)

#Phi_in is the function that needs to be multiplied with eK12 and 1
def Phi_K1(Phi_in):
    Phi = np.zeros(L, dtype=np.complex128)
    for i in range(0, L-1, 2):
        Phi[i:i+2]  = np.dot(eK12, Phi_in[i:i+2])
    Phi[-1] = Phi_in[-1]
    return Phi

def Phi_K2(Phi_in):
    Phi = np.zeros(L, dtype=np.complex128)
    for i in range(0,L-1,2):
        Phi[i+1:i+3] = np.dot(eK12, Phi_in[i+1:i+3])
    Phi[0] = Phi_in[0]
    return Phi
```

[3] $K_{1/2}$, means either K_1 or K_2.

[4] here, with $K_{1/2}$ we are referring to $e^{-i\tau\frac{K_{1/2}}{2}}$

Here, we can see that by choosing `for i in range(0,L-1,2)`, the counter `i` gets incremented by 2 in each step. To obtain even numbers, we directly use the i. For the odd numbers we add 1 to i. These functions also incorporate the eq.(12) cases (in line 8 and 15). With the necessary functions discussed, we can calculate eq.(7) using eq.(8). We cannot save the Φ after time iteration separately, as it would take too much time. Hence, the idea behind the algorithm is to solve is to save the values of $\Phi(x, t+\tau)$ by updating $\Phi(x, t)$. This has the benefit that it is faster, but at the same time, we have to restrict ourselves to save the times we are interested in separately. Updating the $\Phi(x, t = 0)$ m times can be done iteratively with

```python
for i in range(0,m):
        Phi = Phi_K1(Phi)
        Phi = Phi_K2(Phi)
        Phi = Phi*eVx
        Phi = Phi_K2(Phi)
        Phi = Phi_K1(Phi)
```

This `for-loop` is the implementation of eq.(8) in eq.(7), if `Phi` was initially the $\Phi(x, t = 0)$. `eVx` is the implementation of eq.(9) After saving the wavefunction at the times we are interested in, we want to plot the probability $P(x, t)$ as a function of the position x at time t. The probability density is defined as

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

For the positions, the interval size is given by Δ. Therefore, to obtain the probability $P(x, t)$ we have to multiply by that factor, such that

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

The absolute value of can be calculated using `np.absolute()`.

In the second part of the exercise, we want to solve the TDSE for a particle impinging on a potential barrier and investigate the tunneling probability. From the fact that the tunneling probability compared to the reflection is lower, we intend to normalize the values on the right side of the potential. For that, we calculate the total transmitted probability $P(x > 50.5, t)$ for each time t. From those probabilities, we find the maximum value P_{max} and use this to normalize all the values on the other side of the barrier.

3 Results

Based on the discussed algorithm, we now present the results for two potentials. The system was discretized by the number of grid points $L = 1001$, the spatial resolution $\Delta = 0.1$, the small time step $\tau = 0.001$, and the maximal amount of time steps $m = 50000$. Furthermore, the wavepacket is described by its standard deviation $\sigma = 3$, the initial position of the mean $x_0 = 20$, and the momentum $q = 1$.

3.1 System 1: No Potential Barrier

In the first system, we do not consider a potential barrier, which implies

$$V(x) = 0. \tag{14}$$

For this system, we present the simulation results at 6 different times $t = 0, 5, 10, 40, 45, 50$, all given in one plot, presented in fig.(1).

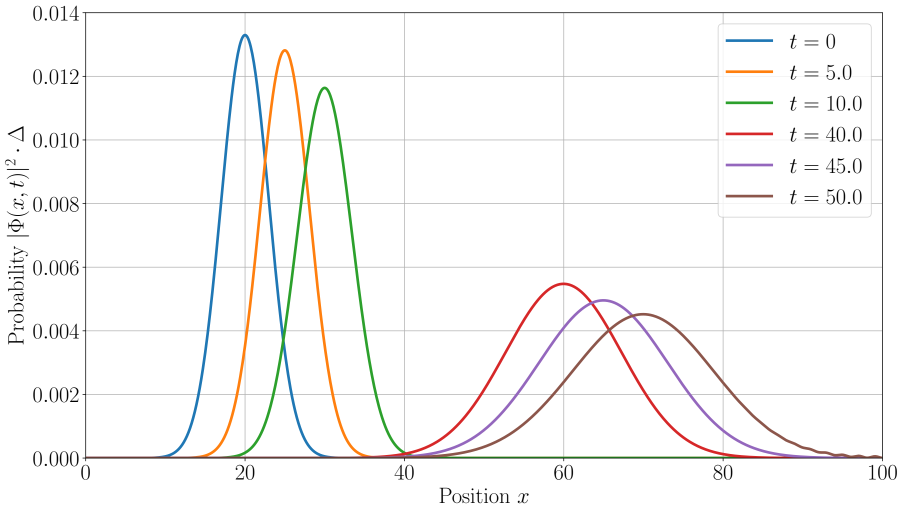

Figure 1: Probability $P(x,t) = |\Phi(x,t)|^2 \cdot \Delta$ versus position x. The simulation results are presented for the five different times $t = 0, 5, 10, 40, 45, 50$ corresponding to the discretization $n = 0, 5000, 10000, 40000, 45000, 50000$, respectively.

3.2 System 2: Square Potential Barrier

Now, we consider a square potential barrier, given by

$$V(x) = \begin{cases} 2, & \text{if} \quad 50 \leq x \leq 50.5 \\ 0, & \text{otherwise.} \end{cases} \tag{15}$$

As discussed in the previous section, the wave transmitted through the potential barrier is normalized by the maximal total transmitted probability. The total transmitted probability is maximal at the time $t = 50$ and has the value

$$P_{max} \approx 0.3283.^5 \tag{16}$$

We present our simulation results for 7 different times in total. Each figure plots the probability $P(x,t) = |\Phi(x,t)|^2 \cdot \Delta$ at a given time t versus the position x. Further, the legends of the plots give the total probability for $x > 50.5$ which is not yet normalized to P_{max}.[6] Fig.(2) contains the results at the times $t = 0, 5, 10$, while fig.(3) shows the result at $t = 25$. Finally, fig.(4) presents the results for $t = 40, 45, 50$.

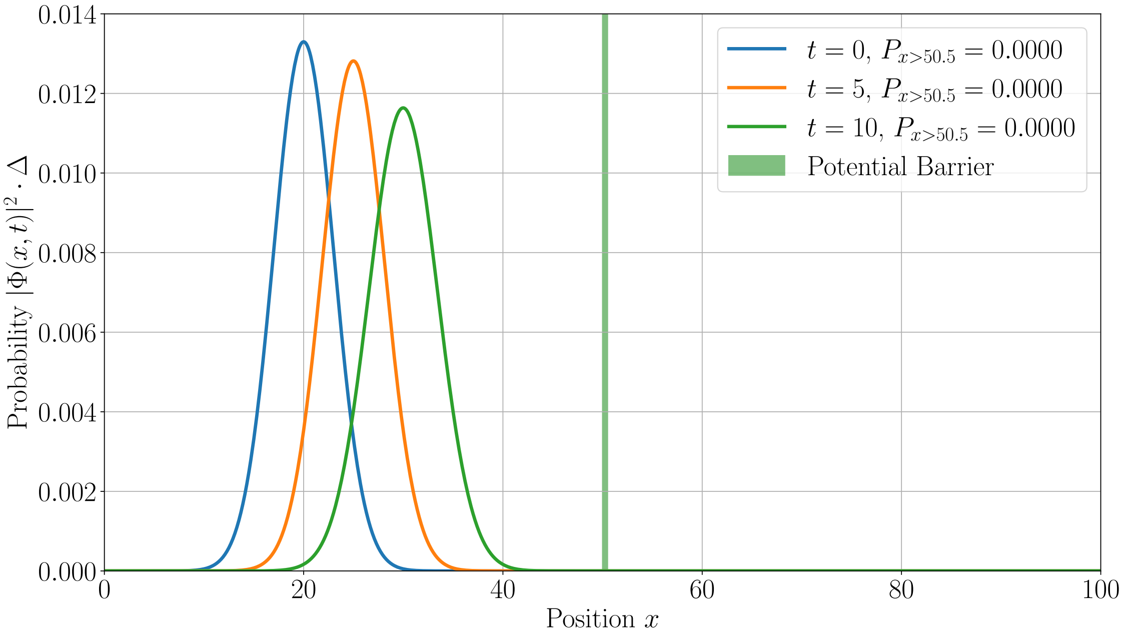

Figure 2: Probability $P(x,t) = |\Phi(x,t)|^2 \cdot \Delta$ versus position x. The simulation results are presented for 3 different times $t = 0, 5, 10$ corresponding to the discretization $n = 0, 5000, 10000$, respectively. (according to $t = m\tau$)

[5]The result is approximate because the true value obtained by the code has more digits

[6]Note that the legend gives the not normalized total probability for $x > 50.5$, while the curves in the plot are normalized by P_{max} for $x > 50.5$.

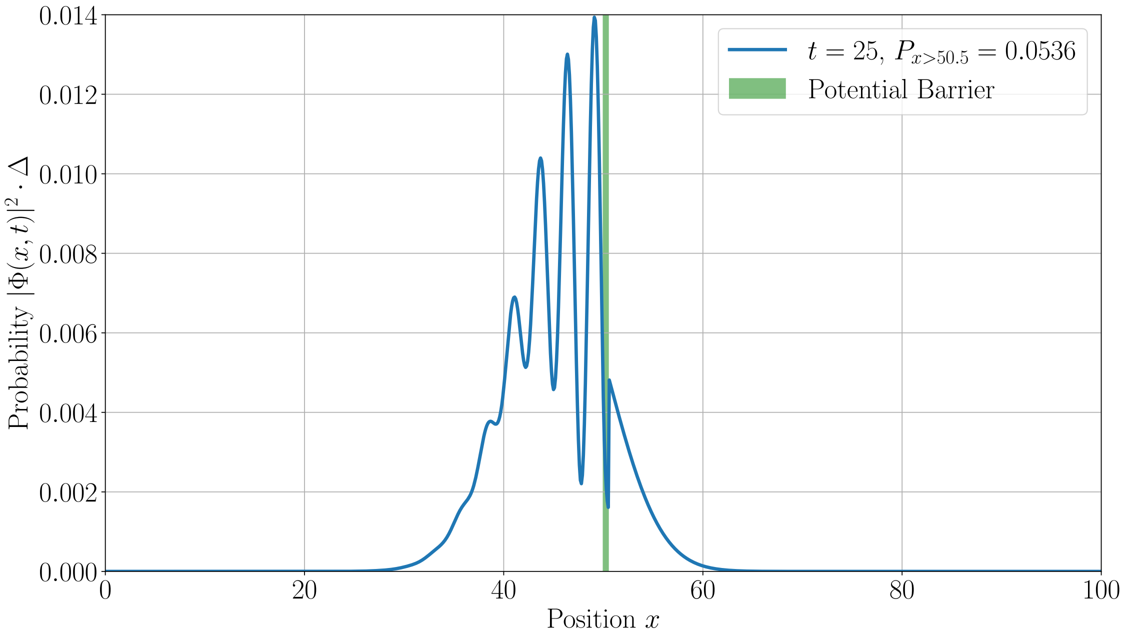

Figure 3: Probability $P(x,t) = |\Phi(x,t)|^2 \cdot \Delta$ versus position x. The simulation results are presented for one time $t = 25$ corresponding to the discretization $n = 25000$. (according to $t = m\tau$)

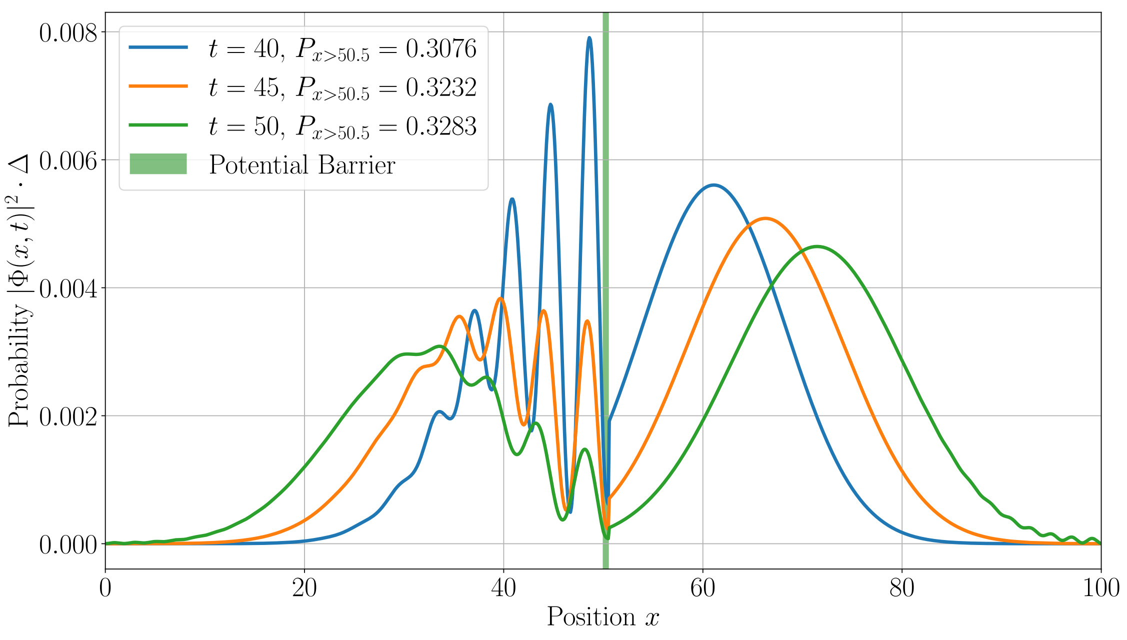

Figure 4: Probability $P(x,t) = |\Phi(x,t)|^2 \cdot \Delta$ versus position x. The simulation results are presented for 3 different times. $t = 40, 45, 50$ corresponding to the discretization $n = 40000, 45000, 50000$, respectively. (according to $t = m\tau$)

3.3 Comparison of the Systems

Finally, we compare the speed of the waves of the two different systems. Therefore, fig.(5) presents two wavepackets. The red wavepacket is the simulation result of system 1 at the time $t = 50$. The maximum of this wavepacket is emphasized by the vertical red dashed line. The green wavepacket is the simulation result for the second system also at $t = 50$. Again, we marked the maximum of the transmitted wave by a vertical dashed line, in green. Note, that the green wavepacket is normalized by P_{max} since it is the transmitted wave.

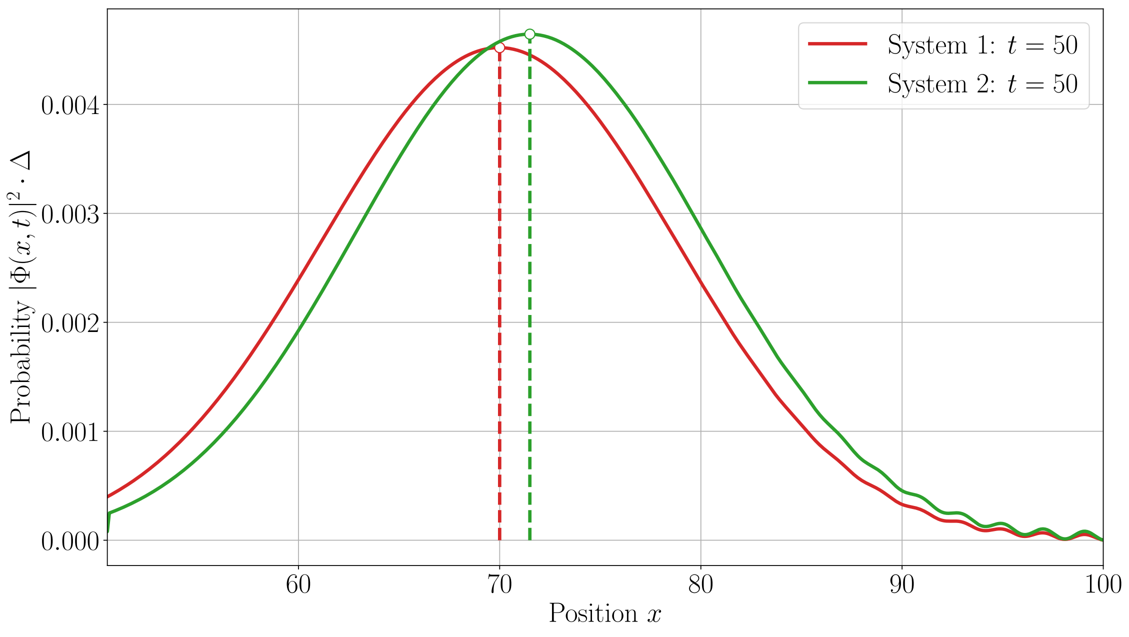

Figure 5: Probability $P(x,t) = |\Phi(x,t)|^2 \cdot \Delta$ versus position x. red: wavepacket of system 1 at $t = 50$. green: wavepacket of system 2 at $t = 50$. The red and green dashed lines are markers for the maximum of the wavepackets of systems 1 and 2, respectively.

4 Discussion

First, we discuss the simplest system, which contains no potential barrier. In fig.(1) the probability of the wavepacket is presented at six different times. The initial wavefunction is described by eq.(1). Here, the parameter $q = 1$ describes the wavenumber and, therefore, gives the average momentum of the particle. Since $q > 0$, we observe that the wavepacket propagates to the right. In the presented results we find that the amplitude of the probability decreases and the width[7] increases. This was very much expected as it is predicted by quantum mechanics. Physically we can interpret it as follows. The wavefunction gives a probability to measure a particle in a certain region of space. However, not only the position of the particle is uncertain, but also the momentum. In momentum-space[8] the wavepacket is also Gaussian distributed, which implies that we also have different probabilities to measure different momenta of the particle. Now, if we consider the particle to be a superposition of wavefunctions that describe different momenta[9], we conclude that wavefunctions with a higher momentum propagate faster than the mean and the ones with lower momenta propagate slower. Therefore, the superposition of all possible momenta results in a wavefunction that spreads out[10] over time since the different contributions propagate at different speeds. Therefore, if the wavepacket spreads out over time the probability also spreads out. Furthermore, besides the effect that the wavepacket widens over time, we can see some interference effects for $t = 50$ at the right boundary of the system due to the reflected wave. Since the simulation considers a limited spatial region, the boundaries of the system reflect the incoming wavepackets.

The second system we present contains a potential barrier that is positioned in the center of the simulated spatial region. The potential barrier is described by eq.(15), which has a "height" of $V(x) = 2$ for $50 \leq x \leq 50.5$. Meanwhile, the average kinetic energy of the particle is given by $K = q^2/2 = 1/2$. Hence, the kinetic energy of the particle is lower than the potential barrier.

First, we discuss the results at early times[11] $t = 0, 5, 10$ (see fig.(2)) where the probability behaves very similarly to the first system. This has the simple reason that the wavepacket is not close enough to the barrier to see noticeable effects. At the time $t = 25$ the wavepacket has reached the potential barrier and we can see the transmitted probability on the right of the barrier.[12] The transmission of the probability is a quantum effect called tunneling, which is from a classical point of view unexpected since the kinetic energy of the particle is lower than the potential barrier. Further, we can clearly observe an interference pattern at the left of the barrier due to reflections of the wavefunction. At last, the simulation results for late times $t = 40, 45, 50$ are presented in fig.(4). Here, we still have the characteristic interference pattern on the left of the barrier and the transmitted wave on the right. We can see that for $t = 50$ the reflected probability has a Gaussian-like shape, but we still see small interference effects. Similarly, we observe that the transmitted wavepacket probability has seemingly the form of a Gaussian wavepacket. Furthermore, for $t = 50$, we notice some wiggles in the probability at the far right which is a result of the reflection of the probability at the right boundary of the simulated spatial region.

Finally, we compare the results of the two systems. To this end, fig.(5) compares the wavepacket of the two different systems at time $t = 50$. Firstly, we notice that the probability of system two (green) has a larger amplitude than the probability of system 1 (red). This is a result of the normalization. We found that the total probability of the transmitted wavefunction is maximal at time $t = 50$, which was

[7]described by the standard deviation
[8]obtained by performing a Fourier tansform
[9]each wavefunction of this superposition is a plane wave with fixed momentum
[10]spread out means that the standard deviation of the probability distribution increases
[11]Early means that the probability distribution has not reached the barrier yet.
[12]Note that there is a jump at $x = 50.5$ since the wavepacket beyond the barrier is normalized for visual purposes.

then used to normalize the transmitted probability. Since we consider both systems at this time, we can conclude that the sum of the total probability of the green curve is equal to one. However, this is not the case for system 1 since the red curve is not normalized for the region $x > 50.5$, it is normalized to the entire spatial region. Therefore, the sum of all probabilities of the red curve, in the presented spatial region, is less than one. Furthermore, the dashed lines represent the maxima of the probabilities and are therefore a measure of how much the wavepackets propagated. We can observe that the transmitted wave (green) propagated farther than the wavepacket which did not transmit through a potential barrier (red). This is a result of the following phenomenon. The fraction of the probability that transmits through the potential barrier is strongly dependent on the energy and, therefore, on the momentum. More accurately, higher momentum has a higher probability to tunnel through the barrier. As the Gaussian wavepacket is a superposition of plane waves with different momenta, the plane waves of higher momentum have a larger contribution to the tunneled wavefunction. Therefore, the transmitted wavefunction has a higher average momentum than the incident wavefunction. Hence, the potential barrier acts like a highpass filter for momenta. We conclude that the transmitted wavepacket propagates faster than a wavepacket that did not transmit through a potential barrier.

Appendix A Code

```python
import numpy as np
import matplotlib.pyplot as plt
from numba import njit

# For fancy plots :)
plt.rcParams["text.usetex"] = True
plt.rcParams["font.family"] = "times new roman"


def setup(xlim_left, xlabel, ylabel, fs, name):
    plt.rcParams["font.size"]   = fs

    plt.figure(figsize=(16, 9))
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    if not name==3:
        plt.ylim(0, 0.014)
    plt.xlim(xlim_left, 100)
    plt.locator_params(nbins=8)
    plt.grid()

pi = np.pi

# discretization parameters
Delta   = 0.1
L       = 1001
tau     = 0.001
m       = 50000

# wavepacket parameters
q       = 1
sigma   = 3
x0      = 20


# barrier paramters
leftcond    = 50
rightcond   = 50.5

# Define the position array
x = np.linspace(0, (L-1)*Delta, L)

# Exponteial block matrix
a = tau/(4*Delta**2)
eK12    = np.array([[np.cos(a), 1j*np.sin(a)], [1j*np.sin(a), np.cos(a)]], dtype=np.
    complex128)             #eK1 and eK2 have the same block matrices


# Initial wavefunction at t=0
@njit
def Phi_ini(x):
    return ((2*pi*sigma**2)**(-1/4) * np.exp( 1j*q*(x-x0) ) * np.exp( -(x-x0)**2/(4*
    sigma**2) )).astype(np.complex128)

# Potential barrier for task 1 (no barrier) and task 2 (with barrier)
def V(x, task=1):
    if task == 1:
        return 0
    if task == 2:
        def temp(x):
            if 50<= x<= 50.5:
                return 2
            else:
                return 0
        return np.vectorize(temp)(x)

# Do the matrix vector multiplication
# Phi_in is the function that needs to be multiplied with eK12 or 1
@njit
def Phi_K1(Phi_in):
    Phi = np.zeros(L, dtype=np.complex128)
    for i in range(0, L-1, 2):
        Phi[i:i+2]  = np.dot(eK12, Phi_in[i:i+2])
    Phi[-1] = Phi_in[-1]
    return Phi

@njit
def Phi_K2(Phi_in):
    Phi = np.zeros(L, dtype=np.complex128)
    for i in range(0,L-1,2):
        Phi[i+1:i+3] = np.dot(eK12, Phi_in[i+1:i+3])
    Phi[0] = Phi_in[0]
    return Phi


# second order product formula algorithm implementation
# Do the matrix vector multiplication iteratively
def solve_product(m, times, task):
    print("\nCheck Normalization:")

    # Define initian condition
    x           = np.linspace(0, (L-1)*Delta, L)
    temp        = Phi_ini(x)

    eVx = np.exp(-1j*tau*(Delta**(-2) + V(x, task)))

    # Do all matrix vector multiplications
    P_max = 0
    for i in range(1,m+1):
        temp = Phi_K1(temp)
        temp = Phi_K2(temp)
        temp = temp*eVx
        temp = Phi_K2(temp)
        temp = Phi_K1(temp)

        # Calculate the maximal total transmitted probability for task 2
        if task==2:
            P_temp = np.sum(np.absolute(temp[506:])**2) * Delta
            if P_max < P_temp:
                P_max = P_temp
                t_max = i*tau

        # Save the data at the times of interest
        if (i==(times/tau).astype(int)).any():
            P = np.absolute(temp)**2*Delta
            np.save(f"Data/TDSE_task{task}_times{int(i*tau)}.npy", P)
            print(f"t={int(i*tau)}, task {task}, sum(P)={sum(P)}")

    # save and print the maximal total transmitted probability for task 2
    if task==2:
        print(f"time at which the total transmitted probability is maximal: t_max={
    t_max}")
```

```
        P_max_arr = np.array([P_max])
        np.save(f"Data/TDSE_task{task}_Pmax.npy", P_max_arr)


# Define all times that we want to investigate
times1      = np.array([0, 5, 10, 40, 45, 50])
times2      = np.array([0, 5, 10, 25, 40, 45, 50])

# Generate the data
#solve_product(m, times1, 1)
#solve_product(m, times2, 2)

# Seperate the times for the plots
times2_1    = np.array([0, 5, 10])
times2_2    = np.array([25])
times2_3    = np.array([40, 45, 50])


# These two functions create the plots
def gen_plots(times, task, name):
    lw = 3
    setup(0, "Position $x$", "Probability $|\Phi(x,t)|^2 \cdot \Delta$", "25", name)
    for i in times:
        if i == 0:
            P = np.absolute(Phi_ini(x))**2 * Delta
        else:
            P = np.load(f"Data/TDSE_task{task}_times{i}.npy")

        if task==2:
            P_max = np.load("Data/TDSE_task2_Pmax.npy")
            P_max_snapshot = np.sum(P[506:])
            P[506:] /= P_max

            subscript   = "x>50.5"
            plt.plot(x, P, lw=lw, label=f"$t={i:.0f}$, $P_{{{subscript}}} = {
    P_max_snapshot:.4f}$")
        else:
            plt.plot(x, P, lw=lw, label=f"$t={i:.0f}$")

    if task == 2:
        plt.axvspan(leftcond, rightcond, alpha=0.5, color="green", label="Potential
    Barrier")

    plt.legend()
    plt.savefig(f"plots/TDSE_task{task}_{name}.pdf")

def gen_plots_compare():
    lw = 3

    P1 = np.load("Data/TDSE_task1_times50.npy")
    P2 = np.load("Data/TDSE_task2_times50.npy")
    P_max = np.load("Data/TDSE_task2_Pmax.npy")

    idx1 = np.argmax(P1)
    idx2 = np.argmax(P2[506:]) + 506

    P2[506:] /= P_max
    print(P_max)

    setup(0, "Position $x$", "Probability $|\Phi(x,t)|^2 \cdot \Delta$", "25", 3)
    plt.xlim(50.5, 100)

```

```python
    plt.vlines(idx1*Delta, 0, P1[idx1], color="tab:red", ls="--", lw=lw)
    plt.plot(x, P1, color="tab:red", lw=lw, label=f"System 1: $t={m*tau:.0f}$")
    plt.plot(x[idx1], P1[idx1], marker="o", markerfacecolor="white", color="tab:red", markersize=9)

    plt.vlines(idx2*Delta, 0, P2[idx2], color="tab:green", ls="--", lw=lw)
    plt.plot(x, P2, color="tab:green" , label=f"System 2: $t={m*tau:.0f}$", lw=lw)
    plt.plot(x[idx2], P2[idx2], marker="o", markerfacecolor="white", color="tab:green", markersize=9)
    plt.legend()
    plt.savefig("plots/TDSE_compare_speed.pdf")




# Generate the plots
gen_plots(times2_1, 2, 1)
gen_plots(times2_2, 2, 2)
gen_plots(times2_3, 2, 3)

gen_plots(times1, 1, 0)

gen_plots_compare()
```