Computational Physics Exercise 7:
Diffusion Equation

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

April 2023

Contents

1 Introduction **2**

2 Task 1: Second Order Product Formula Algorithm **2**

2.1 Simulation Model and Method 2

2.2 Results 5

2.2.1 First Initial Condition 6

2.2.2 Second Initial Condition 8

2.3 Discussion 10

3 Task 2: The Random Walk **11**

3.1 Simulation Model and Method 11

3.2 Results 12

3.3 Discussion 13

A Task 1 Code **14**

B Task 2 Code **17**

[1]fynn.janssen@rwth-aachen.de
[2]tamilarasan.ketheswaran@rwth-aachen.de

1 Introduction

This exercise investigates the diffusion equation with a constant diffusion coefficient and studies the variance of the particle distribution. At first, in task one, the diffusion equation is solved by the second-order product formula algorithm. In exercise two, we calculate the particle distribution by simulating a random walk. The results of exercises one and two are then compared.

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 Task 1: Second Order Product Formula Algorithm

2.1 Simulation Model and Method

In task 1, we investigate the diffusion equation

$$\frac{\partial N(x,t)}{\partial t} = D \cdot \vec{\nabla}^2 N(x,t), \tag{1}$$

where D is the diffusion coefficient. To solve this equation numerically, we first have to discretize it. To this end, we define a very small step in space Δ and in time τ. Then we obtain the following discretization of the partial derivatives

$$\begin{aligned} \frac{\partial N(x,t)}{\partial t} &\longrightarrow \frac{N(x,t+\tau)-N(x,t)}{\tau} \\ \vec{\nabla}^2 N(x,t) &\longrightarrow \frac{N(x+\Delta)-2N(x,t)+N(x-\Delta,t)}{\Delta^2}, \end{aligned} \tag{2}$$

which yields

$$\frac{N(x,t+\tau)-N(x,t)}{\tau} = D\frac{N(x+\Delta)-2N(x,t)+N(x-\Delta,t)}{\Delta^2}. \tag{3}$$

Furthermore, the system is defined on a finite spatial grid, described by

$$x = (\Delta, 2\Delta, ..., (L-1)\Delta, L\Delta), \quad \text{with} \quad L = 1001. \tag{4}$$

To solve eq.(3) efficiently, we define the vector

$$\Phi(t) = \begin{pmatrix} N(1,t) \\ N(2,t) \\ \vdots \\ N(L,t) \end{pmatrix}, \tag{5}$$

which contains the number of particles at a given time t and at all positions. This means that the i^{th} component of $\Phi(t)$ contains the number of particles $N(x_i,t)$ at the position $x_i = i \cdot \Delta$.

[1] abbreviated as `np`

Now, we can rewrite the right-hand side of eq.(3) in matrix form

$$-D\Delta^{-2}\underbrace{\begin{pmatrix} 2 & -1 & 0 & & & 0 \\ -1 & 2 & -1 & & & \\ 0 & -1 & 2 & & & \\ & & & \ddots & & 0 \\ & & & & 2 & -1 \\ 0 & & & 0 & -1 & 2 \end{pmatrix}}_{H}\begin{pmatrix} N(1,t) \\ N(2,t) \\ N(3,t) \\ \vdots \\ \vdots \\ N(L,t) \end{pmatrix} = -D\Delta^{-2}H\Phi(t), \tag{6}$$

for all spatial positions at once. Therefore, this matrix notation yields the differential equation

$$\frac{\partial\Phi(t)}{\partial t} = \alpha H\Phi(t), \tag{7}$$

where we define $\alpha = -D\Delta^{-2}$. The solution is given by

$$\Phi(t+\tau) = e^{\alpha\tau H}\Phi(t). \tag{8}$$

Therefore, $\Phi(t)$ is given by

$$\Phi(t = m\tau) = \left(e^{\alpha\tau H}\right)^m \Phi_0, \tag{9}$$

which is the solution at time $t = m\tau$ using the initial condition $\Phi_0 = \Phi(t=0)$.
Now, it would be very inefficient to calculate the matrix exponential for large matrices. Therefore, we decompose the matrix H and rewrite the matrix exponential using the second-order product formula approximation

$$H = A + B \quad \Rightarrow \quad e^{\alpha\tau H} \approx e^{\alpha\tau A/2}e^{\alpha\tau B}e^{\alpha\tau A/2}, \tag{10}$$

where we define the new matrices A and B as follows:

$$A = -D\Delta^{-2}\left(\begin{array}{cc|cc|cc|c} 1 & -1 & 0 & & & & 0 \\ -1 & 1 & 0 & 0 & 0 & & \\ \hline 0 & 0 & \bullet & \bullet & 0 & & \\ & 0 & \bullet & \bullet & 0 & 0 & 0 \\ \hline & & 0 & 0 & \bullet & \bullet & 0 \\ & & & 0 & \bullet & \bullet & 0 \\ \hline 0 & & & 0 & 0 & 0 & \ddots \end{array}\right) \qquad B = -D\Delta^{-2}\left(\begin{array}{c|cc|cc|cc} 1 & 0 & 0 & 0 & & & 0 \\ \hline 0 & 1 & -1 & 0 & & & \\ 0 & -1 & 1 & 0 & 0 & 0 & \\ \hline 0 & 0 & 0 & \bullet & \bullet & 0 & \\ & & 0 & \bullet & \bullet & 0 & 0 \\ \hline & & & 0 & 0 & \ddots & \cdots \\ 0 & & & & 0 & \vdots & \ddots \end{array}\right)$$

source: Computational Physics – Lecture 16: Diffusion equation II (page 13)

The matrix A consists only of the blocks M along the diagonal

$$M = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \tag{11}$$

except for the last entry, which is one, multiplied by the constant $-D\Delta^{-2}$. Meanwhile, matrix B is also a diagonal block matrix with the same blocks M. However, here, the first component is one, multiplied by the constant $-D\Delta^{-2}$, instead of the last one. These structures of A and B are a result of the structure of H. Therefore, L must be an odd integer[2]. Finally, we can make use of the structure of the matrices A and B. The goal is to calculate

$$e^{\alpha\tau A/2} e^{\alpha\tau B} e^{\alpha\tau A/2} \cdot \Phi. \tag{12}$$

To this end, we start from the right and first determine $e^{\alpha\tau A/2} \cdot \Phi$. Since the matrix exponential of A is also a block matrix we can simply pick pairs of two of Φ and calculate the matrix-vector product of $e^{\alpha\tau M} \cdot \Phi$. The exponential of the block matrix is given by

$$e^{\alpha\tau M} = \frac{1}{2} \begin{pmatrix} 1 + e^{2\alpha\tau} & 1 - e^{2\alpha\tau} \\ 1 - e^{2\alpha\tau} & 1 + e^{2\alpha\tau} \end{pmatrix}. \tag{13}$$

The implementation of these steps looks as follows[3]

```python
def Phi_A2(Phi_in):
    Phi = np.zeros(L)
    for i in range(L//2+1):
        if i-L//2 == 0:
            Phi[L-1] = np.exp(alpha*tau/2)*Phi_in[L-1]
        else:
            temp            = Phi_in[2*i:2*i+2]
            Phi[2*i:2*i+2]  = np.dot(eA2, temp)
    return Phi
```

where `eA2` represents the block matrix of $e^{\alpha\tau A/2}$, given by $e^{\alpha\tau M/2}$. In the code, we have some input array $\texttt{Phi}_{\texttt{in}}$ that is multiplied with $e^{\alpha\tau A/2}$. We iterate from $i = 0$ to $i = L//2$,[4] such that we do $(L-1)/2 + 1 = 501$ steps. Hence, we can multiply 500 pairs of two of Φ with the block matrix exponential. In the last step, we calculate `Phi[L-1]` which is simply given by the multiplication with the constant $e^{\alpha\tau/2}$. After the `for-loop` has finished, the function returns a new array that is the result of the matrix-vector multiplication.

The next step is to multiply this new vector with the matrix exponential $e^{\alpha\tau B}$, which can be achieved in a very similar way, as shown in the next code snippet.

```python
def Phi_B(Phi_in):
    Phi = np.zeros(L)
    for i in range(L//2+1):
        if i == 0:
            Phi[0] = np.exp(alpha*tau)*Phi_in[0]
        else:
            temp = Phi_in[2*i-1:2*i+1]
            Phi[2*i-1:2*i+1] = np.dot(eB, temp)
    return Phi2*i:2*i+2]   = np.dot(eA2, temp)
    return Phi
```

Here, `eB` is of course the block matrix exponential of $e^{\alpha\tau B}$, given by $e^{\alpha\tau M}$. The only difference to the first case is that here the first entry of the diagonal structure is a constant and followed by the 500 block matrices.

Finally, in the last step, we use the resulting array of the function `Phi_B` and use it as an input of the function `Phi_A2`. After these sequential steps, we calculated the matrix-vector product discussed

[2]H has the shape $L \times L$

[3]This is only a small code snippet and would of course not work independently

[4]Note that we start to count from 0 in the code, while in the mathematical description, we start at 1. Further, $L//2$ is equivalent to $(L-1)/2$

in eq.(12).

Based on this algorithm to compute the matrix-vector multiplications efficiently we can now compute $\Phi(t = m\tau)$. By substituting eq.(10) into eq.(9) and obtain

$$\Phi(t = m\tau) \approx \left(e^{\alpha\tau A/2} e^{\alpha\tau B} e^{\alpha\tau A/2}\right)^m \Phi_0. \tag{14}$$

To calculate this expression we have to repeat the previously discussed matrix-vector multiplication (compare eq.(12)) m-times, which is done with a `for-loop`.

Finally, with the obtained array for Φ, we can compute the variance of the position. To this end, we first introduce the mean of $x(t)^p$, given by

$$\langle x(t)^p \rangle = \Delta^p \frac{\sum_{i=1}^{L} (i - i_0)^p \, \Phi_i(t)}{\sum_{i=1}^{L} \Phi_i(t)}. \tag{15}$$

With that, we can calculate the variance divided by the spatial resolution squared

$$\Delta^{-2} var\left(x(t)\right) = \Delta^{-2} \left(\langle x(t)^2 \rangle - \langle x(t) \rangle^2\right), \tag{16}$$

which is the physical quantity of interest.

We notice that the Δ^{-2} cancels with the one in eq.(15). Hence, in the Python implementation, we neglect the Δ^p in eq.(15) to reduce numerical errors. The implementation to obtain the variance divided by Δ^{-2} is presented in the following code snippet.

```python
def x_mean_p(p, phi, i0):
    i_arr   = np.arange(1, L+1)
    return np.sum((i_arr-i0)**int(p)/np.sum(phi) *phi)

def var(phi, i0):
    return x_mean_p(2, phi, i0) - x_mean_p(1, phi, i0)**2
```

Here, we just have to be careful with the indexing of the sum. i and i0 correspond to a position and not indices. i0 plays the role of an initial position of the particles. In this case, we have to use the value for the position, but when we set in initial condition for Phi in the code, we have to use i0 as an index in Python, which differs from mathematical indexing. This may become more clear in the Results section.

Furthermore, the result of the theoretical prediction of the variance as a function of time, is given by

$$\Delta^{-2} var(x(t)) = \frac{2D}{\Delta^{-2}} \cdot t. \tag{17}$$

2.2 Results

In this subsection, we present the results for two different initial conditions. Other than that, both systems are discretized with identical parameters. The diffusion coefficient $D = 1$ which only fixes the scale, the spatial size is characterized by $L = 1001$ and the spatial resolution $\Delta = 0.1$, where L is the number of discrete lattice points. Furthermore, the time discretization is given $\tau = 0.001$ and $m = 10000$, such that the simulation time is given by $t = m\tau = 10$.

Furthermore, for each initial condition, we present plots for $\Phi(t)$ vs. the position x at three different times $t = 0, 0.03, 0.06$. We choose such small times in comparison to $t = 10$ to provide clarity.[5]

[5]At later times, the curves a very flat and you do not see very much if the y-axis ranges from 0 to 1.

2.2.1 First Initial Condition

The first initial condition is given by

$$\Phi_i(t=0) = \begin{cases} 1 & \text{if} \quad i = i_0 \equiv (L+1)/2 = 501 \\ 0 & \text{if} \quad i \neq i_0. \end{cases} \tag{18}$$

Note that the index i in Φ_i uses the mathematical convention and starts counting from one. In the code, we start indexing from zero, such that we set the initial conditions, given by the array `Phi_0`, as follows

```
Phi_0 = np.zeros(L)
Phi_0[500] = 1.
```

Here, the index `500` corresponds to the 501th element of the array.

The calculated array for Φ plotted versus the position x is presented in fig.(1). For clarity, we only present the solution in a small spatial region. Since the array $\Phi_i(t)$ is equal to $N(i \cdot \Delta, t)$, the y-axis of this plot corresponds to the function $N(x,t)$. Moreover, we give the sum over the array Φ, to check whether it varies with time.[6]

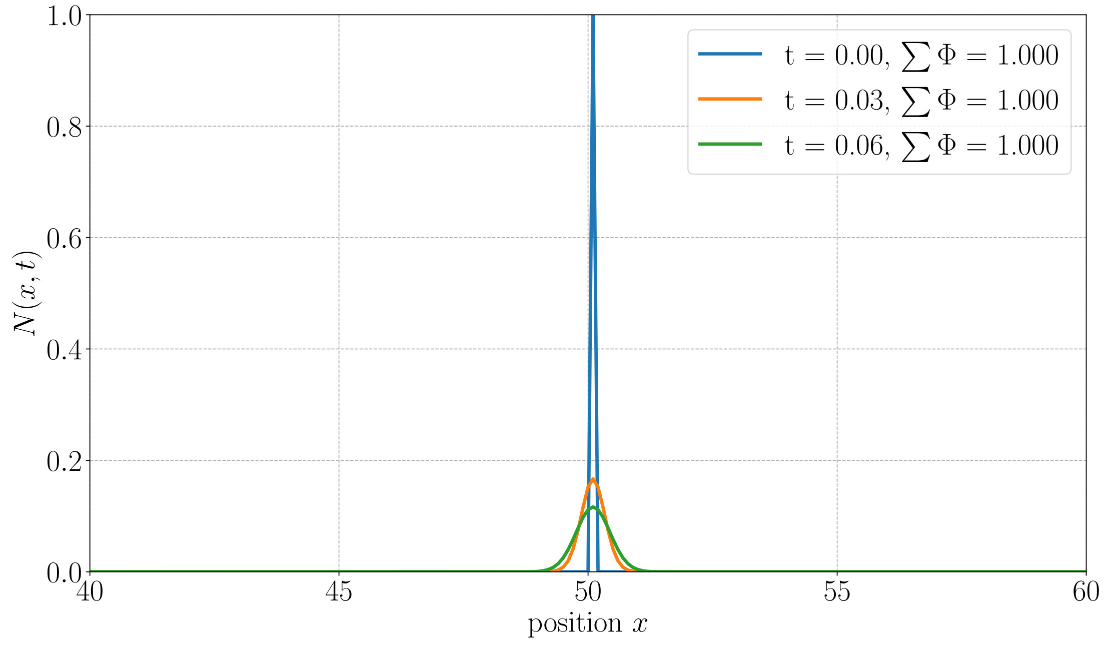

Figure 1: The array Φ versus the position x in the spatial region from $x = 40$ to $x = 60$. The initial distribution of $\Phi_i(t = 0)$ is described by eq.(18). The solution for Φ is presented at the three times $t = 0, 0.03, 0.06$ and additionally, we give the sum over each entry of the array Φ. The y-axis is labeled as $N(x,t)$ since $\Phi_i(t) = N(i \cdot \Delta, t)$.

In fig.(2), we present the results for the variance of the position $x(t)$ divided by Δ^2 versus time t. This plot also contains a black dashed linear curve, given by

$$y = \text{slope} \cdot t, \tag{19}$$

[6]Note that the solution for the array Φ is of course discrete even if it is presented in the plot as a continuous line.

where we consider the slope between the first and last points of the array Φ.

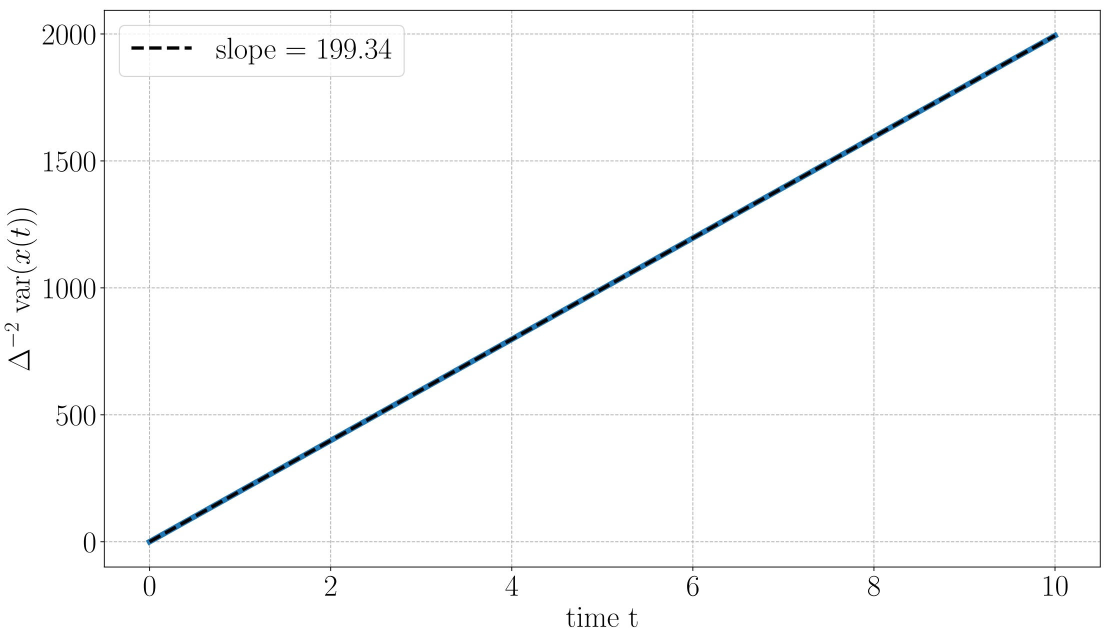

Figure 2: The variance of the position $x(t)$ over Δ^2 versus time. The initial distribution of $\Phi_i(t = 0)$ is described by eq.(18).

2.2.2 Second Initial Condition

In the second case, the system is described by the initial conditions

$$\Phi_i(t=0) = \begin{cases} 1 & \text{if} \quad i = i_0 = 1 \\ 0 & \text{if} \quad i \neq i_0, \end{cases} \tag{20}$$

which corresponds in the code to

```
    Phi_0 = np.zeros(L)
Phi_0[0] = 1
```

due to the different indexing. Hence, in this system, all particles are initially at the left boundary of the system. Here, we obtain the result for the variance of the position, presented in fig.(4).

Again, we first present the calculated array for Φ plotted versus the position x, presented in fig.(3). Therefore, keep in mind that $\Phi_i(t) = N(i \cdot \Delta, t)$, which justifies the y-axis label. Here, the solution is presented in the spatial region from $x = \Delta = 0.1$ up to $x = 2.0$, to provide clarity. We also present the sum over the array Φ in the legend of the plot.

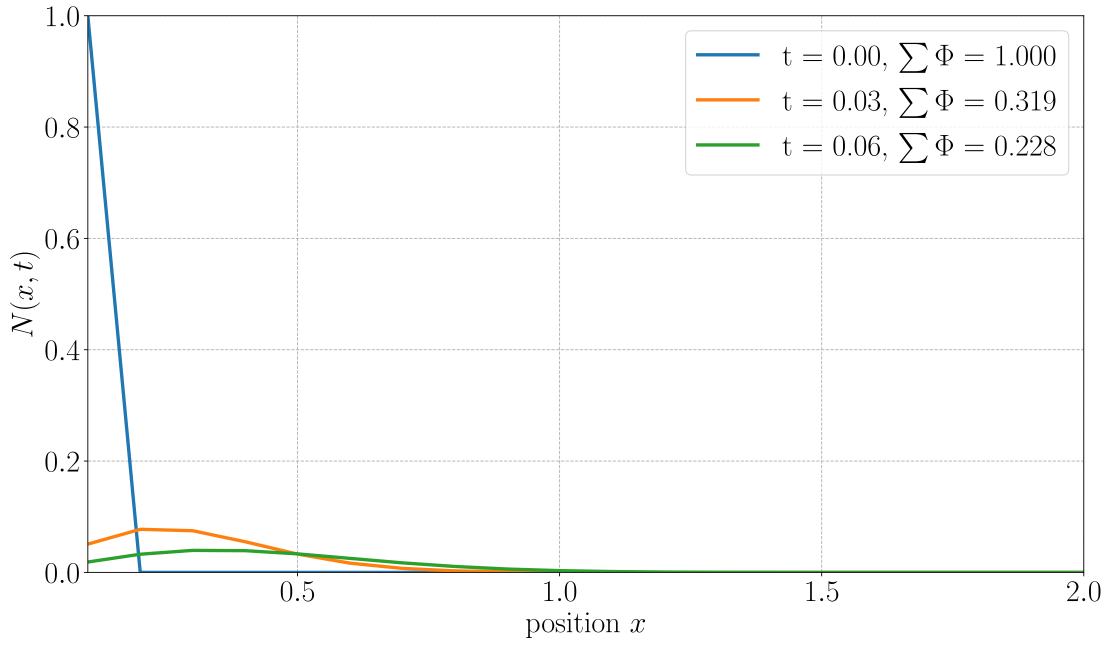

Figure 3: The array Φ versus the position x in the spatial region from $x = 0.1$ to $x = 2.0$. The initial distribution of $\Phi_i(t = 0)$ is described by eq.(20). The solution for Φ is presented at the three times $t = 0, 0.03, 0.06$ and additionally, we give the sum over each entry of the array Φ. The y-axis is labeled as $N(x,t)$ since $\Phi_i(t) = N(i \cdot \Delta, t)$.

In fig.(4), the result for the variance of the position $x(t)$ divided by Δ^2 versus time t is presented. Similarly to the first system, we present a black dashed curve (according to eq.(19)) calculate with the slope between the first and last points.

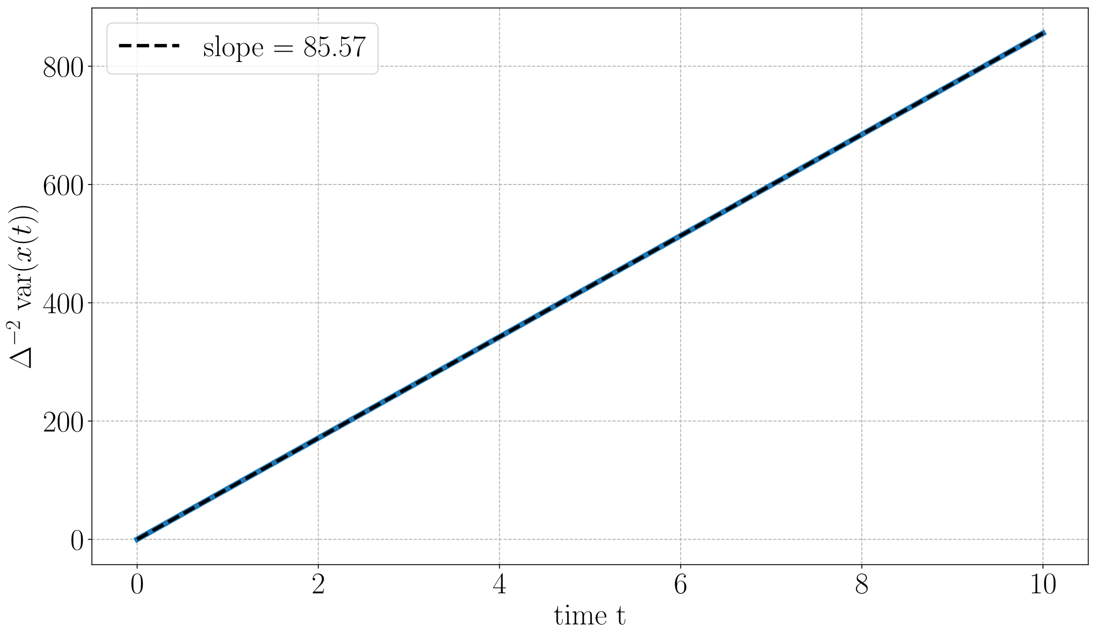

Figure 4: The variance of Φ over Δ^2 versus time. The initial distribution of $\Phi_i(t = 0)$ is described by eq.(20).

2.3 Discussion

First, we discuss the initial conditions. The function $N(x,t)$ and therefore, the calculated array Φ can be interpreted as the number of particles at position x and time t. However, in eqs. (18) and (20), we have set one value of Φ to one, while the other entries of the array remain zero. However, this does not mean that we can just describe one particle. We can multiply the diffusion equation (see eq.(1)) by any factor and absorb this factor in the function N, and the equation still holds. This implies that we can arbitrarily normalize the array Φ. In our case, Φ is normalized to one. Now, $N(x,t)$ can be interpreted as a probability.[7]

To have a conserved number of particles or a valid probability, the sum of the array Φ has to be constant in time, such that

$$\sum_{i=1}^{l} \Phi_i(t) = \sum_{i=1}^{l} N(i \cdot \Delta, t) = 1, \quad \forall t. \tag{21}$$

Furthermore, the amplitude of N does not influence the variance since it is just a rescaling of the y-axis.

Now, we can take a look at the time evolution of the system for the two different initial conditions, presented in figs. (1) and (3). In the case of the first initial condition, in fig.(1), we can see that the distribution $N(x,t)$ widens over time, while the amplitude decreases and the sum over all position remains constant and is equal to one.[8] Furthermore, the increase in the width of the distribution can be perfectly interpreted as diffusion: particles spreading randomly in both directions or the probability to find one particle spreading. This implies that the algorithm yields physical results.

If we compare this to the second system (see fig.(3)), we find different results. In this case, the sum over the distribution $N(x,t)$ decreases with time. This is a result of the initial condition, which was set at the border of the system (compare eq.(20)). In this case, the algorithm does not perform well due to the fixed system size.

This lets us conclude that the algorithm is limited by its size. Hence, we would expect that system one[9] also yields non-physical results if we consider large enough times such that the diffusion reaches the boundaries of the system.

Finally, we discuss the variance of the position, presented in figs. (2) and (4). In both cases, we find a linear dependency and $var(\Phi(t=0)) = 0$. However, the two results have different slopes.

For the first system (see fig.(2)) we find a slope of 199.34, which is approximately equal to $2D/\Delta^2$. This matches the theoretical prediction given in eq.(17) up to an error of less than 0.5%. Hence, the result of this system is in agreement with the theoretical prediction. However, for the second system (see fig.(4)), the variance over Δ^2 has a slope of 85.57, which is a significant deviation from the theoretical prediction. This deviation was expected, as we discussed above.

In summary, the algorithm yields reasonable physical results if the initial distribution of Φ is in the center of the system (compare system one). If this is not given, the algorithm gives not physical results due to the boundary effects that we discussed (compare system two).

[7]Note that the Schroedinger equation of a free particle is a similar differential equation, just with different prefactors.
[8]We of course only consider a small time span, but this is sufficient to compare both systems.
[9]described by the initial conditions in eq.(18)

3 Task 2: The Random Walk

3.1 Simulation Model and Method

For the second task, one has to do a random walk for n particles. All of them start at the position i_0 and after each small step in time τ, each particle goes either to the left or right (± 1). After repeating this process $1/\tau$ times, we say that $\Delta t = 1$ has elapsed. In the end, we want to calculate the variances of the position of the particles at times $t = 0, 1, 2, ..., 10$. The points of interest are initialized via

```
t = np.arange(11).
```

This system itself can be implemented with one array consisting of the position of each particle. The said array is initialized using

```
N = np.ones(n)*i0.
```

In this array of length n, each entry is set to i_0. For the random walk in each iteration, we implement a function returning

```
np.random.choice([-1,1],n).
```

Calling the function returns us an array of length n, consisting of either $+1$ or -1. In each entry of the array, there is a $1/2$ chance between being a $+1$ or -1. Adding this array onto our `N`, changes each value of the array `N` by either $+1$ or -1. For our model, a $+1$ corresponds to the particle going to the right and -1 to the left. Additionally, we define a function, which does the process of adding the array consisting of random values $\frac{1}{\tau}$ times. This corresponds to the time $\Delta t = 1$ being elapsed. From this function, we can save the position of each particle at each time t and are left with the task to calculate the distribution of the particles. Counting the number of particles at each position can be done using[10]

```
np.histogram(N[t], bins = L, range=(0,L))[0].
```

As the last part, we need to calculate the variances of the position, in general, the variance is defined as

$$\mathrm{Var}(x) = \langle x^2(t) \rangle - \langle x(t) \rangle^2, \tag{22}$$

here the averages of x^p are calculated using

$$\langle x^p(t) \rangle = \frac{\sum_{i=1}^{L} (i - i_0)^p N_i(t)}{\sum_{i=1}^{L} N_i(t)}, \qquad p = 1, 2.$$

Our results will be compared to the results of the diffusion equation. From the second exercise, dealing with the random walk, we have shown that $\mathrm{Var}(x_m) = m(\Delta x)^2$. Here, in our case $(\Delta x)^2 = 1$, such that

$$\mathrm{Var}(x_m) = m$$

additionally we know

$$D = \frac{\Delta^2}{2\tau}$$

[10]Important to note is that it is possible that in the rare case that the particles overshoot the length of the array on either side (total length is L). Here, they must be let out of the analysis. But as we checked that non of the particles reach the vicinity of either border, no special treatment was needed

and with $t = m\tau$, we are left with

$$\Rightarrow m = \frac{2D}{\Delta^2} t$$

$$\mathrm{Var}(x(t)) = \frac{2D}{\Delta^2} t \tag{23}$$

$$\tag{24}$$

This result can be plotted against our result from the simulation model to compare the theoretical expectation with the implementation.

3.2 Results

In this subsection, we present the results for the initial condition $i_0 = (L+1)/2 = 501$ with $L = 1001$. Furthermore the number of particles is chosen to be $n = 10000$. The time discretization is chosen to be $\tau = 0.001$, spatial resolution $\Delta = 0.1$, and the diffusion coefficient $D = 5$.

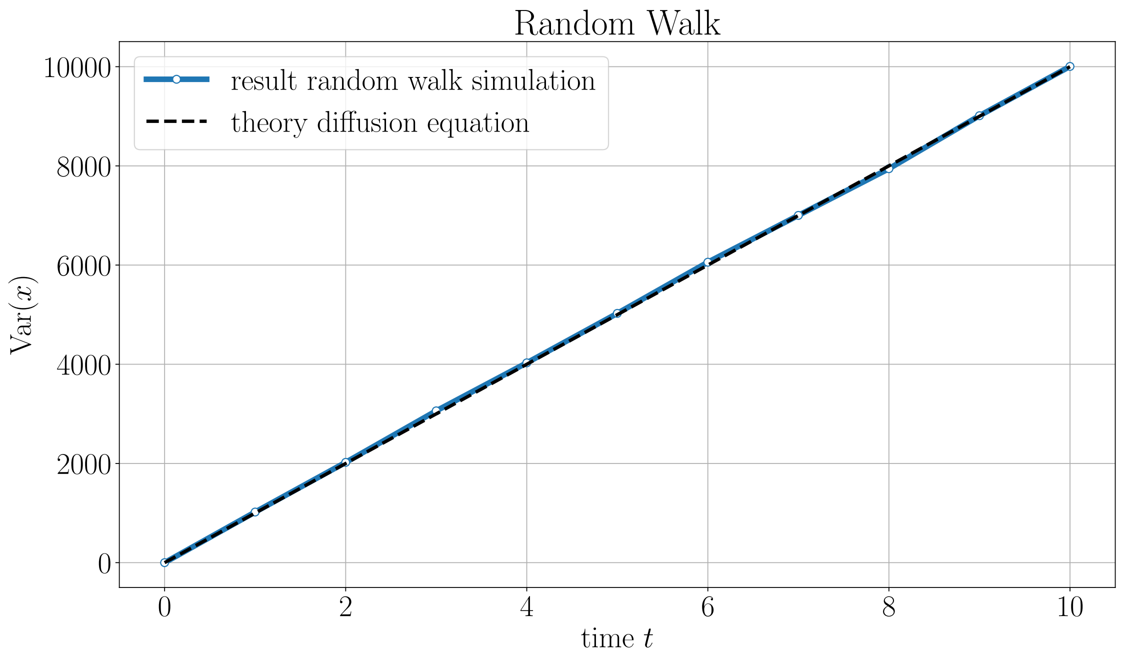

Figure 5: The variance on the position x versus time t. The dashed black line shows the theory according to eq.(24) and the blue line shows the simulation according to the model described in the section before. The slope of the black curve is equal to 1000

3.3 Discussion

In fig.(5), we can clearly see how the results of the diffusion equation with $D = 5$, match our random walk simulation well. Nevertheless, we can see some fluctuations in the blue curve compared to the black line. In exercise 2, a thorough explanation of said deviations was done. The main aspect is the fact that these deviations are mainly of statistical nature. To understand the implications of the curves matching, one has to keep in mind how the diffusion constant plays a role in the slope. In eq.(24), we have seen that we expect linear behavior according to $\mathrm{Var}(x(t)) = \frac{2D}{\Delta^2}t$. This linear function has a slope defined as slope $= \frac{2D}{\Delta^2}$. Hence, under the condition that the Δ is the same in both, simulation and theory, the diffusion constant D must be the same too. And indeed from the same slope, we can derive that also for our random walk simulation the diffusion constant must then equal to $D = 5$.

Appendix A Task 1 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"
plt.rcParams["font.size"]   = "27"

def setup(title, xlabel, ylabel):
    plt.figure(figsize=(16, 9))
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.xticks()
    plt.yticks()
    plt.locator_params(nbins=6)
    plt.grid(ls="--")

# Define Parameters
L       = 1001
Delta   = 0.1
tau     = 0.001

D       = 1
alpha   = -D/Delta**2

# Calculate the mean of x**p/Delta**2
@njit
def x_mean_p(p, phi, i0):
    i_arr   = np.arange(1, L+1)
    return np.sum((i_arr-i0)**int(p)/np.sum(phi) *phi)

# Calculate variance/Delta**2
@njit
def var(phi, i0):
    return x_mean_p(2, phi, i0) - x_mean_p(1, phi, i0)**2

# Define the exponential Block matrices
eA2 = np.array([[1+np.exp(alpha*tau), 1-np.exp(alpha*tau)],[1-np.exp(alpha*tau), 1+np
    .exp(alpha*tau)]])/2
eB  = np.array([[1+np.exp(2*alpha*tau), 1-np.exp(2*alpha*tau)],[1-np.exp(2*alpha*tau)
    , 1+np.exp(2*alpha*tau)]])/2

# Calculate the matrix-vector multiplication eA2*Phi_in
@njit
def Phi_A2(Phi_in):
    Phi = np.zeros(L)
    for i in range(L//2+1):
        if i-L//2 == 0:
            Phi[L-1] = np.exp(alpha*tau/2)*Phi_in[L-1]
        else:
            temp            = Phi_in[2*i:2*i+2]
            Phi[2*i:2*i+2]  = np.dot(eA2, temp)
    return Phi

# Calculate the matrix-vector multiplication eB*Phi_in
```

```python
@njit
def Phi_B(Phi_in):
    Phi = np.zeros(L)
    for i in range(L//2+1):
        if i == 0:
            Phi[0] = np.exp(alpha*tau)*Phi_in[0]
        else:
            temp = Phi_in[2*i-1:2*i+1]
            Phi[2*i-1:2*i+1] = np.dot(eB, temp)
    return Phi

# do the three matrix-vector multiplications subsequently 3 times and iterate m times
# to obtain the soluztion at time t = m*tau
# After each steps calculate the variance
@njit
def solve_product(m, i0):
    # Define initian condition
    Phi0         = np.zeros(L)
    Phi0[i0-1]   = 1

    var_arr = np.zeros(m+1)
    temp    = Phi0
    for i in range(m):
        temp = Phi_A2(temp)
        temp = Phi_B(temp)
        temp = Phi_A2(temp)
        var_arr[i+1] = var(temp, i0)
    return var_arr


# Do the plots
def plot_var(i0, m):
    t       = np.linspace(0, m*tau, m+1)
    var_arr = solve_product(m, i0)

    slope = (var_arr[-1]-var_arr[0])/(t[-1]-t[0])

    np.save(f"var_{i0}.npy", var_arr)
    setup(" ", "time t", r"$\Delta^{-2}$ var($x(t)$)")
    plt.plot(t, var_arr, lw=5)
    plt.plot(t, slope*t, lw=3, ls="--", color="black", label=f"slope = {slope:.2f}")
    plt.legend()
    plt.savefig(f"plots/var_{i0}.pdf")

# Similar function to solve_product(m, i0)
# Here we do plots of the array Phi vs. x
def plot_N(m, i0):
    x            = np.arange(L)+1
    Phi0         = np.zeros(L)
    Phi0[i0-1]   = 1

    setup(" ", "position $x$", r"$N(x, t)$")
    plt.ylim(0,1)
    temp = Phi0
    for i in range(m+1):
        if (i)%(m//2) == 0:
            plt.plot(x, temp, label=rf"t = {i*tau:.2f}, $\sum \Phi$ = {np.sum(temp)
:.3f}")
        temp = Phi_A2(temp)
        temp = Phi_B(temp)
        temp = Phi_A2(temp)

    if i0==501:
        plt.xlim(400, 600)
    else:
        plt.xlim(1, 20)

    plt.legend()
    plt.savefig(f"plots/Phi_{i0}.pdf")
    plt.close()

# Define initial condition
i01     = int(L+1)//2
i02     = 1

# set time: t = m*tau
m = 10000

# time for the other plots
m2 = 60

# Generate plots
plot_var(i01, m)
plot_var(i02, m)

plot_N(m2, i01)
plot_N(m2, i02)
```

Appendix B Task 2 Code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Jun 18 16:23:33 2023

@author: fynn@tamilarasan
"""
import numpy as np
import matplotlib.pyplot as plt
import csv
from numba import njit, vectorize, float64
import os
os.environ["PATH"] += os.pathsep + '/Library/TeX/texbin'

plt.rcParams["text.usetex"] = True
plt.rcParams["font.family"] = "times new roman"
plt.rcParams["font.size"] = "27"


pi      = np.pi
rand    = np.random
#set the seed
rand.seed(1069)
#the constants given by the exercise
L = 1001
n = 10000
Delta = 0.1
D = 5
#inital conditions
i0  = (L+1)//2
#array consisting the current position of each particle
N = np.ones(n)*i0
tau = 0.001
#time array
t = np.arange(11)


#a function returning the direction each particle will be moved in
def walk(n):
    return rand.choice([-1,1],n)

#function that does one iteration of t = t+1
def onet(N):
    F = N.copy()
    for i in range(int(1/tau)):
        F = F+walk(len(F))
    return F

#array below will contain the position of the particles at each time
N_total = [N]
#loops over all times
for T in t:
    N_total +=[onet(N_total[T])]

#function calculating the averages of x^p
def x_mean_p(p, N):
    i_arr   = np.arange(1, L+1)
    return np.sum((i_arr-i0)**int(p) *N)/np.sum(N)

# Calculate the variance/Delta^
def var(N):
    return x_mean_p(2, N) - x_mean_p(1, N)

#array consisting the variances at each time t
Var = []
for i in range(11):
    #histogram is used to calculate the distribution of the particles at each
   position
    Var +=[var(np.histogram(N_total[i], bins = 1001, range=(0,1001))[0])]

#plots
plt.figure(figsize =(16 , 9))
plt.plot(t,Var, lw=5, marker="o", markersize=7, markerfacecolor="white", label= "
   result random walk simulation")
#theoretical prediction
plt.plot (t , 2*D/(Delta **2) * t , lw =3 , color = "black" , ls = "--", label= "
   theory diffusion equation" )
plt.title("Random Walk")
plt.xlabel(r"time $t$")
plt.grid()
plt.legend()
plt.ylabel(r"Var$(x)$")
plt.savefig("task2.pdf")
```