FYS3150 - Computational Physics - Fall 2014

Final project (4 & 5)

Diffusion of neurotransmitters in the synaptic cleft

Candidate number: 22

Date: 8. desember 2014

Introduction

In this project we had a look at different methods for solving diffusion equations numerically, both in one and two dimensions. As a specific problem we considered diffusion of neurotransmitters in the synaptic cleft.

Diffusion equations

A diffusion equation is a partial differential equation on the form

$$\frac{\partial u(r,t)}{\partial t} = D\nabla u(r,t)$$

where D is the diffusion constant. In this project we set D=1 and we consider first the one dimensional problem before we advance to a two dimensional one.

One-dimensional case

The equation we were to solve in the one-dimensional case looked like

$$\frac{\partial u(x,t)}{\partial t} = \frac{\partial^2 u(x,t)}{\partial x^2}.$$

This equations can be solved both analytically and numerically, as we will do later on. A solution to the equation is a function $u(x,t)$ which satisfy the equation above. However in order to do so we must have initial- and boundary conditions. The solution must obey them as well. In this problem $u(x,t)$ represented the distribution of neurotransmitters at a position x within the synaptic cleft at a time t. We defined the length of the synaptic cleft to be L so that the presynaptic and postsynaptic membranes were positioned at $x = 0$ and $x = L$ respectively.

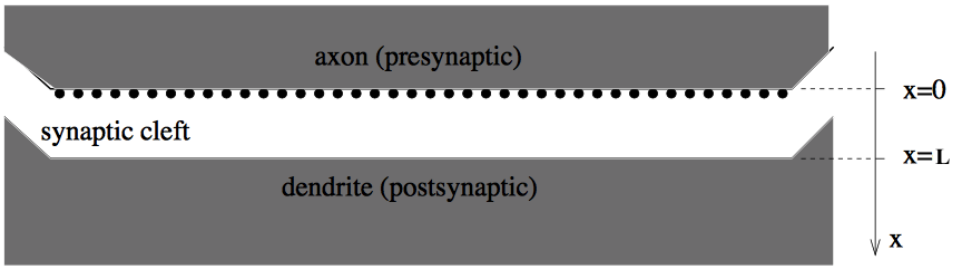

Figure 1: Schematic drawing of the synaptic cleft in our model. The black dots represent neurotransmitter molecules, and the situation shown corresponds to $t = 0$.

The initial condition we were given was that all the neurotransmitters were to be at the presynaptic membrane, as Figure 1 illustrates nicely. We also had boundary conditions at both ends of the spatial interval. At the presynaptic membrane ($X = 0$) the concentration of neurotransmitters should always be equal one. At the other end, at the postsynaptic membrane, it should always equal zero. Written mathematically the conditions were

$$u(x,0) = 0 \quad , x > 0 \tag{1}$$
$$u(0,t) = 1 \tag{2}$$
$$u(L,t) = 0 \tag{3}$$

After a "long time" the distribution function $u(x,t)$ has diverged to a so-called steady state solution. At this point it will no longer change with time. For a system such as here, the steady state solution will take a linear form, so:

$$u_s(x) = Ax + b \ .$$

Boundary condition (2) gives $b = 1$, and (3) gives $A = -1/L$. Thus the steady state is

$$u_s(x) = 1 - \frac{x}{L} \ . \tag{4}$$

We now define another function:

$$v(x,t) = u(x,t) - u_s(x) \ , \tag{5}$$

which is the difference between the state we're in and the steady state. The reason we do this is that we obtain dirichlet boundary conditions, which is easier to work with both analytically and numerically. We obtain the initial- and boundary conditions:

$$v(x,0) = x/L - 1 \tag{6}$$
$$v(0,t) = 0 \tag{7}$$
$$v(L,t) = 0 \tag{8}$$

Analytical solution

We will now try to derive the solution to the diffusion equation of the function, as this also gives the solution of $u(x,t)$. We have the equation:

$$\frac{\partial v(x,t)}{\partial t} = \frac{\partial^2 v(x,t)}{\partial x^2} \ . \tag{9}$$

We use a technique called separation of variables to solve this, meaning we assume that

$$v(x,t) = X(x)T(t)$$

which transforms (9) into:

$$\frac{1}{T(t)}\frac{\partial T(t)}{\partial t} = \frac{1}{X(x)}\frac{\partial^2 X(x)}{\partial x^2} \ .$$

We allow this to equal some constant $-\lambda^2$. The equations we have is thus:

$$\frac{\partial^2 X(x)}{\partial x^2} + X(x)\lambda^2 = 0 \ ,$$

with the solution

$$X(x) = A\sin(\lambda x) + B\cos(\lambda x) \ .$$

The boundary conditions will now restrict the coefficients.

$$(7) \ \Rightarrow \ B = 0$$

$$(8) \ \Rightarrow \ A\sin(\lambda x) = 0 \ \Rightarrow \ \lambda = \frac{\pi n}{L} \quad , n \in \mathbb{Z}$$

We assume $n > 0$ because the negative values are only linear combinations of the positive, and thus the positive values alone span the entire space.
The time dependent function has to obey:

$$\frac{\partial T(t)}{\partial t} + T(t)\lambda^2 = 0 \ ,$$

which has the solution

$$Ce^{-\lambda^2 t} \ .$$

where C is some constant. The solution to the diffusion equation (9) is thus:

$$v(x,t) = \sum_{n=1}^{\infty} A_n \sin\left(\frac{\pi n}{L}x\right) e^{-\left(\frac{\pi n}{L}\right)^2 t} \ ,$$

where the constant C in the time dependent function has been absorbed in A_n. The initial condition (6) gives

$$\frac{x}{L} - 1 = \sum_{n=1}^{\infty} A_n \sin\left(\frac{\pi n}{L}x\right) \ .$$

A_n is nothing but the Fourier coefficients of the function $x/L - 1$, and therefore:

$$\begin{aligned} A_n &= \frac{2}{L}\int_0^L \left(\frac{x}{L} - 1\right)\sin\left(\frac{n\pi}{L}x\right) dx \\ &= \frac{2}{n\pi}\Big((1-L)\cos(n\pi) - 1\Big) \ . \end{aligned}$$

Finally we have the analytical solution

$$v(x,t) = \sum_{n=1}^{\infty} \frac{2}{n\pi}\left[(1-L)\cos(n\pi) - 1\right] \sin\left(\frac{\pi n}{L}x\right) e^{-\left(\frac{\pi n}{L}\right)^2 t}$$

In this project we set $L = 1$, and therefore the analytical solution of the diffusion equation we consider is reduced to:

$$v(x,t) = \sum_{n=1}^{\infty} -\frac{2}{n\pi} \sin(\pi n x)\, e^{-(\pi n)^2 t} \ . \tag{10}$$

Numerical solution

In order to solve this diffusion equation numerically there are several schemes we could use. In this project we focused on three in particular: the explicit, the implicit and the Crank-Nicolson. A fourth option is to use Monte Carlo methods, with the use of random walks, in order to model the evolution of the system. I will come back to this later. First I will present the algorithms for the schemes and their stability properties. I will, for the sake of readability, use the abbreviations:

$$u_t = \frac{\partial u(x,t)}{\partial t} \qquad\qquad u_{xx} = \frac{\partial^2 u(x,t)}{\partial x^2} \ .$$

I also use the discretization:

$$\begin{aligned} x_i &= i\Delta x \\ t_j &= j\Delta t \\ u_{i,j} &= u(x_i, t_j) \ . \end{aligned}$$

The explicit scheme is written as

$$u_t = \frac{u_{i,j+1} - u_{i,j}}{\Delta t} \tag{11}$$

$$u_{xx} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \tag{12}$$

and thus the equation reads

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}$$

Deffining $\alpha = \Delta t/\Delta x^2$ and rearranging a bit we obtain

$$u_{i,j+1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j} \ , \tag{13}$$

which is the algorithm we use for this scheme. It has a truncation error of order $O(\Delta t)$ for the time-derivative and $O(\Delta x^2)$ for the spatial part. The reason for this is that in the approximations for rate of change in time we do not include terms of higher order than Δt, and likewise for the double derivative of x, we do not include terms containing higher order of Δx than Δx^2.

The implicit scheme is written as

$$u_t = \frac{u_{i,j} - u_{i,j+1}}{\Delta t} \tag{14}$$

$$u_{xx} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \tag{15}$$

Rendering our problem to be:

$$\frac{u_{i,j} - u_{i,j+1}}{\Delta t} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \ .$$

We define α as for the explicit scheme and end up with the equation:

$$u_{i,j-1} = -\alpha u_{i-1,j} + (1-2\alpha)u_{i,j} - \alpha u_{i+1,j} \ . \tag{16}$$

If we define vectors

$$\mathbf{v}_j = \begin{pmatrix} u_{0,j} \\ u_{1,j} \\ \vdots \\ u_{n,j} \end{pmatrix} ,$$

the above equation (16) can be expressed as a matrix-vector equation

$$\mathbf{v}_{j-1} = \hat{\mathbf{A}}\mathbf{v_j} \ . \tag{17}$$

With the matrix $\hat{\mathbf{A}}$ being the tridiagonal matrix

$$\mathbf{A} = \begin{pmatrix} 1+2\alpha & -\alpha & & & & \\ -\alpha & 1+2\alpha & -\alpha & & & \\ & \ddots & \ddots & \ddots & & \\ & & -\alpha & 1+2\alpha & -\alpha & \\ & & & -\alpha & 1+2\alpha & \end{pmatrix} . \tag{18}$$

(All unfilled elements of the matrix are zeros.) Since we got the initial conditions of the system in this problem we set $\mathbf{v}_{j-1}$ to be this function (for the one-dimensional case the initial condition is the initial state). Thus the only unknown quantity in equation (17) is $\mathbf{v}_j$. This is a problem we have solved in project 1, and I used the same technique here. I have added my report on project 1 in case you would like to look it up for an explanation of the algorithms, but it is not a part of this project. However, as we find $\mathbf{v}_j$, we have done one iteration in time. Using this new vector as the new $\mathbf{v}_{j-1}$ we can find the function at the next iteration and so on. Also this scheme has a truncation error of order $O(\Delta t)$ for the time-derivative and $O(\Delta x^2)$ for the spatial part.

The Crank-Nicolson scheme is a bit different from the others. This scheme make a better approximation of the time derivative by using the midpoint rule. As a result we evaluate the time derivative at $t' = t + \Delta t/2$. Therefore we must also calculate the double derivative with respect to x at this time. Meaning we must approximate

$$\frac{\partial u(x,t')}{dt} \qquad \text{and} \qquad \frac{\partial^2 u(x,t')}{dx^2}$$

We approximate these terms by using the midpoint rule.

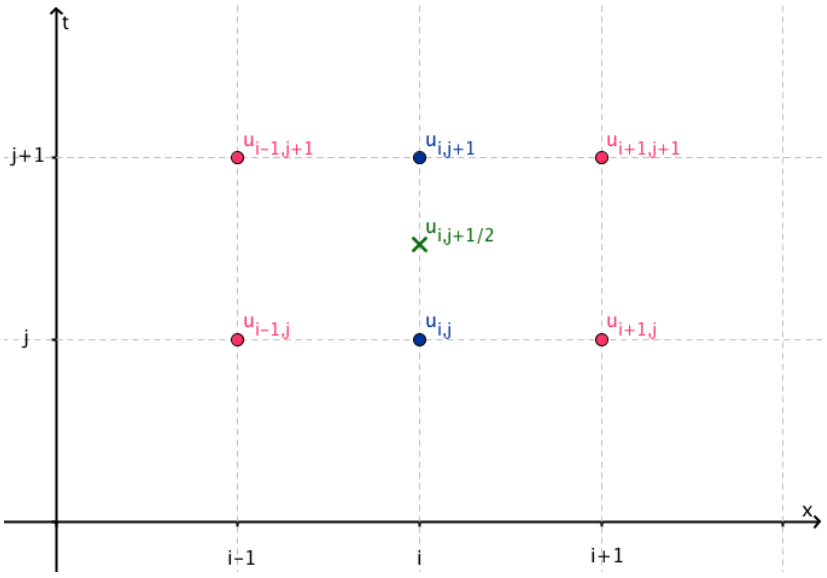

Figure 2: Calculation molecule for the Crank-Nicolson scheme.

Basically we approximate the time derivative of the element $u_{i,j+1/2}$ using the midpoint rule as

$$\frac{\partial u(x,t')}{dt} \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t}$$

And we also approximate the spatial part in the same way.

$$\frac{\partial^2 u(x,t')}{dx^2} \approx \frac{1}{2}\left[\frac{u_{i+1,j+1} - 2u_{i,j+1} + u_{i-1,j+1}}{\Delta x^2} - \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}\right]$$

These have to equal. By using again the $\alpha = \Delta t/\Delta x^2$ and reorganizing, this results in

$$-\alpha u_{i+1,j+1} + (2+2\alpha)u_{i,j+1} - \alpha u_{i-1,j+1} = \alpha u_{i+1,j} + (2-2\alpha)u_{i,j} - \alpha u_{i-1,j} \ . \tag{19}$$

Which is the algorithm for the Crank-Nicolson method. This scheme has a truncation error of order $O(\Delta t^2)$ for the time-derivative and $O(\Delta x^2)$ for the spatial part. The reason for the improved approximation of the time-derivative is that we use the midpoint rule. All the algorithms shown here can be neatly summarized by the so-called theta method

$$\frac{u_{i,j} - u_{i,j-1}}{\Delta t} = \frac{\theta}{\Delta x^2}\left(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\right) + \frac{1-\theta}{\Delta x^2}\left(u_{i+1,j-1} - 2u_{i,j-1} + u_{i-1,j-1}\right)$$

where $\theta = 0$ gives the explicit scheme as in (13), $\theta = 1$ gives the implicit scheme as in (16) and $\theta = 1/2$ gives the Crank-Nicolson scheme as in (19).

The stability of the schemes can be analyzed by inspection of the matrices we get when rewriting the algorithms as matrix-vector equations. We did this for the implicit scheme, and using the same definition of the vectors $\mathbf{v}_j$ we can show that

$$(13) \quad \Rightarrow \quad \mathbf{v}_j = \hat{\mathbf{A}}\mathbf{v}_{j-1} \quad , \hat{\mathbf{A}} = \hat{\mathbf{I}} - \alpha\hat{\mathbf{B}}$$

$$(16) \quad \Rightarrow \quad \mathbf{v}_j = \hat{\mathbf{A}}^{-1}\mathbf{v}_{j-1} \quad , \hat{\mathbf{A}} = \hat{\mathbf{I}} + \alpha\hat{\mathbf{B}}$$

$$(19) \quad \Rightarrow \quad \mathbf{v}_j = \hat{\mathbf{A}}\mathbf{v}_{j-1} \quad , \hat{\mathbf{A}} = \left(2\hat{\mathbf{I}} + 2\alpha\hat{\mathbf{B}}\right)^{-1}\left(2\hat{\mathbf{I}} - 2\alpha\hat{\mathbf{B}}\right)$$

where $\hat{\mathbf{A}}$ is as in (18), and

$$\hat{\mathbf{B}} = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} .$$

It can be shown that the eigenvalues of this matrix ($\hat{\mathbf{B}}$) are $2 - 2\cos\theta$.

The stability of the schemes is given by the spectral radius, $\rho(\hat{A})$ of the matrices $\hat{\mathbf{A}}$ above. The spectral radius is defined as

$$\rho(\hat{A}) = \max\left\{ |\lambda| : det\left(\hat{A} - \lambda\hat{I}\right) = 0 \right\}$$

Meaning that the spectral radius of a matrix, $\hat{A}$, equals the largest (in magnitude) eigenvalue of this matrix. The criteria for the schemes to be stable is:

$$\rho(\hat{A}) < 1 \ .$$

If this criteria is pleased, the resulting vector will converge towards a finite value. In our case toward the so-called steady state solution. However, the three schemes have different matrices in their matrix-vector problems. Thus we check the stability of each one explicitly. The criteria of the spectral radius tells us that for the explicit scheme we have

$$\rho(\hat{I} - \alpha\hat{B}) < 1$$

$$-1 < 1 - \alpha 2(1 - \cos\theta) < 1$$
$$0 < 2\alpha(1 - cos\theta) < 2$$

By definition $\alpha > 0$.

$$\alpha < \frac{1}{1 - \cos\theta}$$
$$\alpha < \frac{1}{2}$$

This means that the explicit scheme is stable as long as $\alpha < 1/2$.
For the implicit case we have

$$\rho((\hat{I} + \alpha\hat{B})^{-1}) < 1$$

$$-1 < \frac{1}{1 + \alpha 2(1 - \cos\theta)} < 1$$

Which obviously is true for all values of α (because $\alpha > 0$), meaning the implicit scheme is stable for any value of α! For the Crank-Nicolson method we have

$$\rho((2\hat{I} + 2\alpha\hat{B})^{-1}(2\hat{I} - 2\alpha\hat{B})) < 1$$

$$-1 < \frac{2 - 2\alpha(1 - \cos\theta)}{2 + 2\alpha(1 - \cos\theta)} < 1$$

Lets look at the criteria separately

$$-(2 + 2\alpha(1 - \cos\theta)) < 2 - 2\alpha(1 - \cos\theta) \qquad 2 - 2\alpha(1 - \cos\theta) < 2 + 2\alpha(1 - \cos\theta)$$
$$-4 < 0 \qquad 4\alpha(1 - \cos\theta) > 0$$

As both of these always are true, again because of the definition of α, the Crank-Nicolson scheme is stable for any value of α as well.

Results

In my script i implemented the schemes in a class in such a way that calling upon it, with a vector as input, it will do one iteration in time and return a new vector. In the main function we can tweak the variables (limits, number of points, time period...). I Set the spatial step length to $\Delta x = 0.1$ and time step length $\Delta t = 0.005$, meaning we are at the limit of the stability criteria for the explicit scheme. I included the situation at two different times bellow. One at an early stage (Figure 3 and 4) and one at a later stage (Figure 5 and 6).

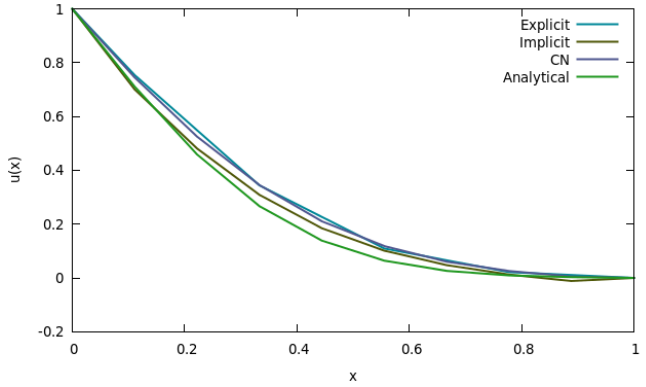

Figure 3: Calculated state of the system in an early stage. ($t = 0.05$)

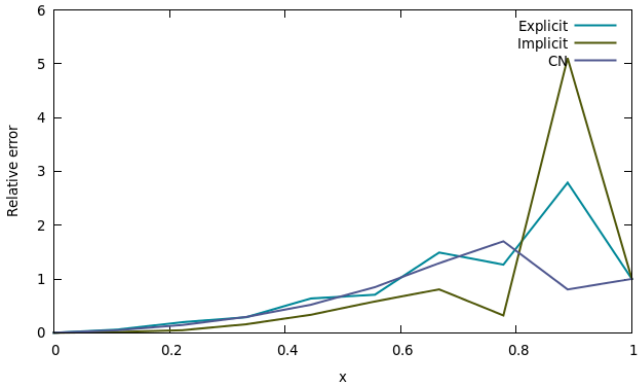

Figure 4: Relative error compared to the analytical solution, in an early stage ($t = 0.05$).

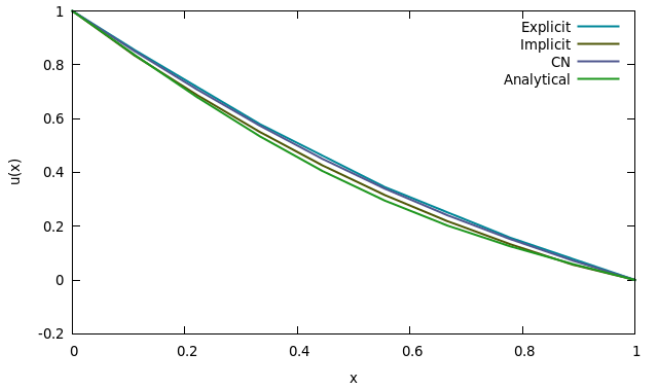

Figure 5: Calculated state of the system at a later stage. ($t = 0.1$)

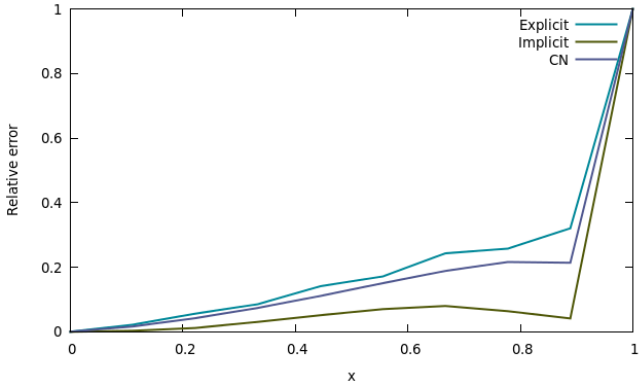

Figure 6: Relative error compared to the analytical solution, at a later stage ($t = 0.1$).

We see that as time develops the state converges toward the steady state (straight line from (0,1) to (1,0)), as expected. I think the figures showing the relative error is of the most interest. We clearly see that the explicit scheme has the largest relative error, however I was surprised that the Crank-Nicolson wasn't far superior to the implicit as well. It does have a better approximation for the time-derivative, and so I expected a better approximation than the

implicit scheme, however for the late stage the implicit is pretty much dead on, and the Crank-Nicolson is not! In the early stage the implicit is also the best approximation in the major part of the interval, but close to the end the relative error escalates. The thing is, the error isn't even that big, but the relative error is. With this as my evidence I would say that the implicit scheme is the best one, though this does not make sense to me. I guess it depend on the problem. Lastly, I think I should explain why the relative error is 1 for all the schemes at $x = 1$. This is simply a consequence of the boundary conditions demanding that the functions has to equal the steady state at this point, and as the relative error between the analytical value, A, and the one for one of scheme, S, is:

$$\text{relative error} = \left| \frac{A - S}{A} \right| .$$

Since A is a calculated number (very close to zero at $x = 1$, but not exactly zero) and S is zero at $x = 1$ by demand, the relative error at $x = 1$ is $A/A = 1$. These point of Figure (4) and Figure (6) should be ignored.

Monte Carlo methods

The same problem can be solved using Monte Carlo methods and random walks. Basically what we do is to create many particles, and at each iteration, we demand that the particles move, either left or right. And we must off course have some boundary- and initial conditions as to where the particles can be. This is really the only tricky part about this method. We must have conditions that make sense, considering the problem we deal with. We allow the system to go through several such iterations, and we observe how it evolves. Some details can make big differences. For instance how far the particles can move or the probabilities of moving left or right, or standing still. We truly have to understand the system in order to make a good Montecarlo simulation. However if one does, the method is fairly simple, since we don't even have to think about the differential equation at all. I will now solve the same problem as above (one dimensional case), but using Monte Carlo methods. First with discrete step-lengths, and then with a continuous randomly (to a certain degree) picked step-length.

Modeling the system

Our system is still as described on page 1. Lets make a simulation of N neurotransmitters (from now on referred to as "particles"). At the initial state all the particles are at position $x = 0$. In my simulation I therefore make a vector containing N elements, set to zero, representing the position of each individual particle. At a each iteration we run through every particle and propose a random move, we then add this to the particles position. The "move" will be discussed later. As a particle moves from $x = 0$ we add a new element to the list with position $x = 0$. The reason for this is that the number of particles at the axon (presynaptic side, $x = 0$) has to be constant. Thus the number of particles in the system increases, at least in the beginning. Now, what happens if a particle moves outside of the system (x<0 or x>1)? Well, what we know is that the concentration of the particles at the dendrite (postsynaptic side, $x = 1$) should be zero, because as they hit the dendrite they are absorbed. Thus we impose that if a particle moves past $x = 1$ we simply delete it. If a particle moves to $x < 0$ it will no longer effect the system, as the concentration of particles at the axon already is 1 (It doesn't effect some probability of adding a particle at $x = 0$ for instance). Thus we delete those as well. In my code I said that if a particle moves form $x = 0$ to $x < 0$, then don't bother doing anything, as this will save some operations. Also for the constant step-length case if a particle moves to $x = 0$, we ignore it as well. This is easiest to understand by just studying the code (attached at the back).

Discrite positions

First of let's look at a Monte Carlo simulation with constant a step-length $l_0 = \sqrt{2D\Delta t}$. This means that each particle (in the system) will be at a position

$$x = i \cdot l_0 \quad \text{, where } i \in 0, 1, 2 \ldots L/l_0.$$

The D int l_0 is the diffusion constant, which we have set to $D = 1$. As we see, Δt effects how far a particle can move, as it should. In the simulation we make one iteration each time-step, and at each iteration we let there be a 50% chance the particle moves l_0 to the left, and a 50% chance the particle moves l_0 to the right. It can not remain at the same position. Doing this simulation we get the results bellow.

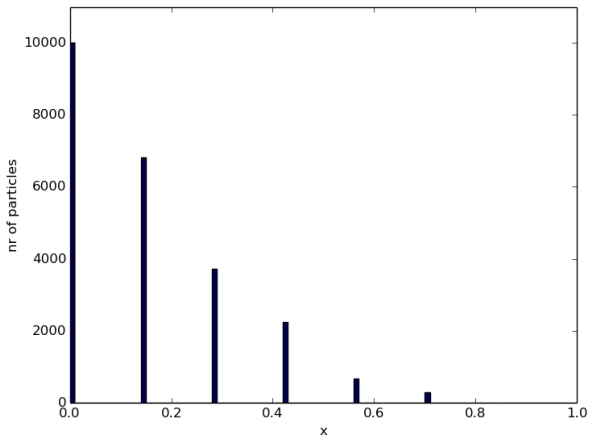

Figure 7: Simulated state of the system in an early stage ($t = 0.05$), using constant step-size.

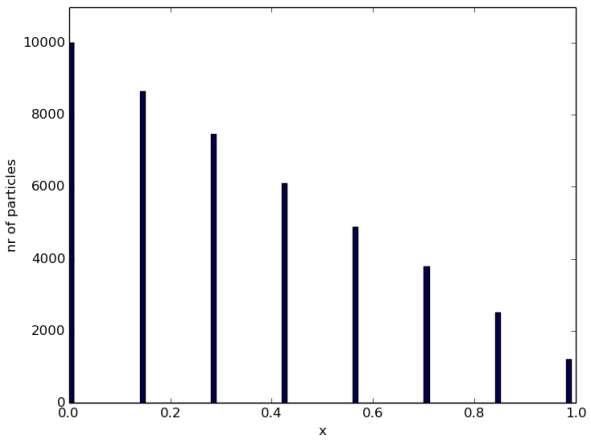

Figure 8: Simulated state of the system at a late stage ($t = 1.00$), using constant step-size.

In the figures I had 100 bins in the histogram. As we see the number of particles at the end points are as they should be, the particles take only discrete values and also the evolution of the system is very accurate. This is pretty awesome considering the few lines of code it required! Also do notice the similarity of Figure 7 and Figure 3. The resolution (number of spikes) in the figures are related to the time-step length. Shorter Δt gives higher resolution (more spikes), but it requires quite a bit more time. I didn't find it worth the wait to include a figure with higher resolution, as the figures above show the concept well enough.

Continueous positions

Now, lets change the situation a bit. Lets allow the particles to be in any position within the interval $x = [0, 1]$, and to move with a gaussian distributed step-length. We define the new step-length $l_0 = \sqrt{2D\Delta t}\xi$, where ξ is a random number chosen from a Gaussian distribution with mean value 0 and standard deviation $1/\sqrt{2}$. This now suggests that any particle at any position, can jump to any other position, and not only some discrete position next to it, as in the constant step-length case. However, the probability of a large jump is small, and the most probable event is that the particle barely moves. A graphical illustration of the probabilities for different values of ξ is shown in Figure 9.

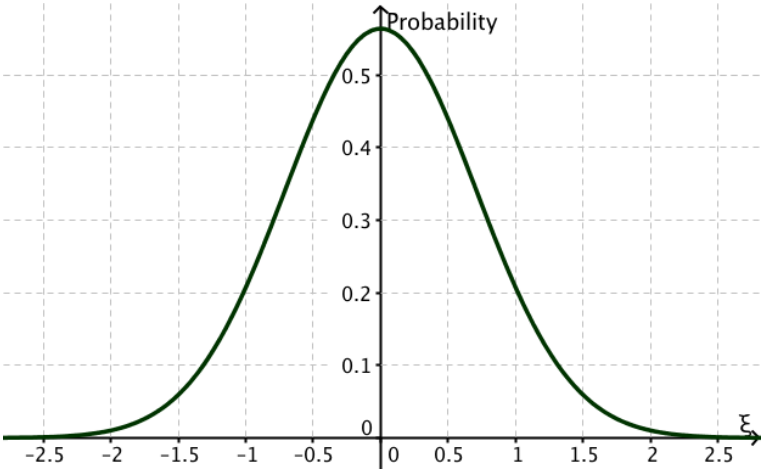

Figure 9: Gaussian probability distribution with mean value 0 and standard deviation $1/\sqrt{2}$.

This setup will cause a more realistic model because in reality all the particles are not moving at the same speed all the time. Maybe they bump into one another, causing some particles to move toward positive x and some toward negative. The only difficulty though is that we can no longer add new particles at $x = 0$ whenever a particle leave $x = 0$. The reason for this is that at every iteration we now call upon the random number generator, which creates a gaussian distributed number and multiplies it with $\sqrt{2D\Delta t}$. This will never be exactly zero, but most likely it will be close. If we define that there should always be N particles at $x = 0$, we would then get a sky high number of particles

at a small interval, dx, which does not match the system we are modeling. In stead we require that the number of particles inside the small interval, dx, should be N at all times. Allowing dx to be small, this is a good approximation. Else, we leave the same conditions such as deleting particles that exit our system and adding new ones inside our interval at $x = 0$. "This should do it" was my thought when I had written the last piece of code using this strategy. The result is shown in Figure 10 bellow.

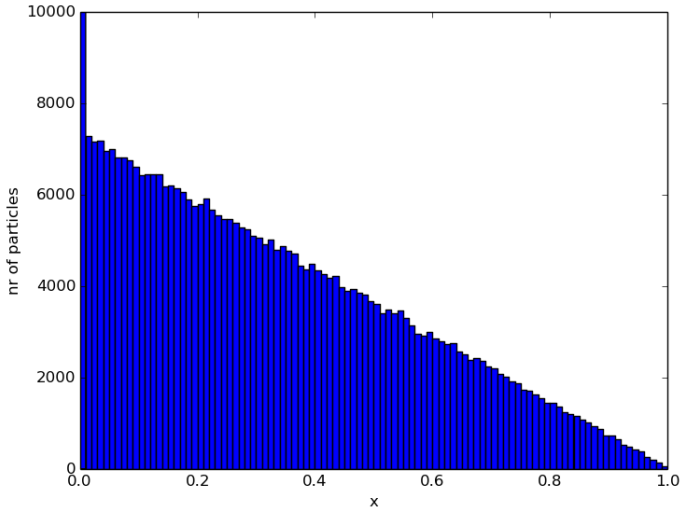

Figure 10: Late stage of a Monte Carlo simulation with continuous positions using 10000 particles. Adding new particles at $x = 0$.

As you can imagine I got mixed feelings from the data I got. Sure, the figure has that nice linear form, but that nasty spike does not fit in. Well, it has the correct value as required. This problem got me thinking allot before I found the solution. It is fairly simple really. The entire problem is that the new particles are added at $x = 0$! First off remember that now particles entering the system are the ones that pass $x = dx$. Since new particles are placed in $x = 0$ and most particles will only slightly move, most of the particles in the dx-interval will be on the left most side, meaning they have a greater chance of jumping out of the interval on the left side and spawn at $x = 0$, than to jump out on the right side. Thus not as many particles will make it into the rest of the system as they should. The solution was simply to change the spawning position to be in the middle of the dx-interval, $x = dx/2$. By doing so we get the results shown bellow.

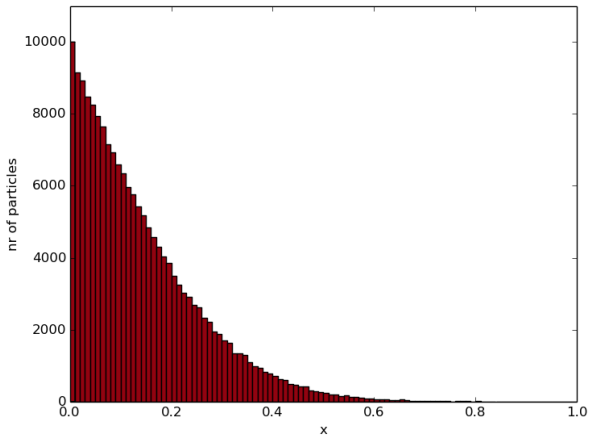

Figure 11: Monte Carlo simulated state of the system in an early stage ($t = 0.05$), using $\Delta t = 0.00005$, $\Delta x = 0.01$ and continuous positions.

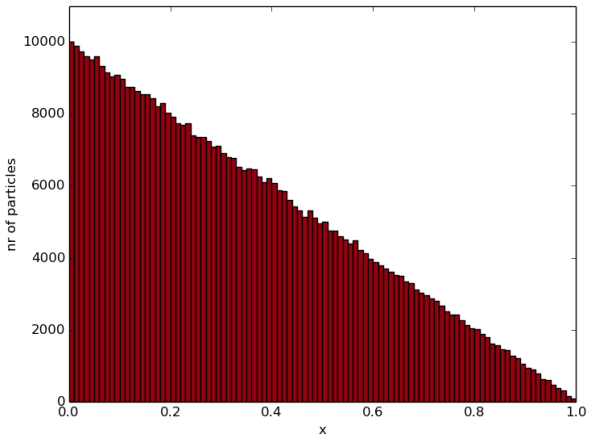

Figure 12: Monte Carlo simulated state of the system in a late stage ($t = 1.00$), using $\Delta t = 0.00005$, $\Delta x = 0.01$ and continuous positions.

Which obviously look great, figure 12 is pretty much exactly what we would expect to see at a late stage. But how

can we know if the way our simulation evolves is accurate? We would have to compare it to something we know to be correct. Therefore, lets use the same conditions as we did when calculating the analytical solution shown in Figure 3 and then compare the results. Setting $\Delta t = 0.005$ and $\Delta x = 0.1$ we get at a time $t = 0.05$ the result in Figure 13.

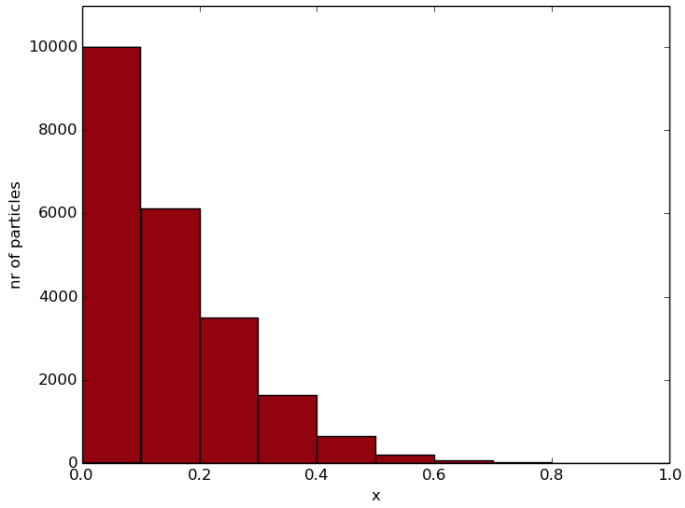

Figure 13: Monte Carlo simulated state of the system in an early stage ($t = 0.05$), using $\Delta t = 0.005$, $\Delta x = 0.1$ and continuous positions.

The reason for the thick bins, is that $\Delta x = 0,1$, meaning each bin must be 0.1 wide. Comparing this to Figure 3 it does look very similar. So much in fact that I dare to say that the simulation is accurate.

Diffusion in two dimensions.

We will now solve the diffusion equaltion in two dimensions. We redefine our function $u = u(x, y, t)$. The particles can now not only move allong a line, but anywhere on the x, y-plane (until we restrict them with boundary conditions). The equation we need to solve is:

$$\frac{\partial u(x,y,t)}{\partial t} = \frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2}. \tag{20}$$

In order to find a solution, again we must define initial and boundary conditions. Remember that we are still modeling the same system, som lets keep $u(0, y, t) = 1$ and $u(L, y, t) = 0$ as before, and at the initial state, let the rest of the system be empty (of particles), meaning

$$u(x, y, 0) = 0 \ , \quad 0 < x \leq L \ \text{ and } \ 0 \leq y \leq H$$

I have defined that the particles can run from 0 to L in the x-direction and from 0 to H in the y-direction. Later we will set these equal to 1, but I find it convinient to make a more general solution and then specify the simple case later, because I then have the machinery to adjust the parameters as I wish later. We have now the same condition as before, but we still have to define conditions for the boundary of y. I figured that the boundaries hsould correspond to the steady state solution, and therefor I simply chose the boundary conditions to for y to equal the steady state $u_s = 1 - x/L$. To summarize the condition we have:

$$\begin{aligned}
u(x, y, 0) &= 0 \ , \quad 0 < x \leq L \ \text{ and } \ 0 \leq y \leq H \\
u(0, y, t) &= 1 \\
u(L, y, t) &= 0 \\
u(x, 0, t) &= 1 - x/L \\
u(x, L, t) &= 1 - x/L
\end{aligned}$$

We now use the same trick as in the one-dimensional case, defining a function $v(x,y,t) = u(x,y,t) - u_s(x)$. This function will then have the conditions:

$$v(x,y,0) = x/L - 1\ , \quad 0 < x \leq L \ \text{ and } \ 0 \leq y \leq H \tag{21}$$
$$v(0,y,t) = 0 \tag{22}$$
$$v(L,y,t) = 0 \tag{23}$$
$$v(x,0,t) = 0 \tag{24}$$
$$v(x,L,t) = 0 \tag{25}$$

Again we have the nice dirichlet boundary conditions. We now have everything we need and will proceed to find an analytical solution to the problem.

Analytical solution (two-dimensional case)

We solve it similarly as we did in the one-dimensional case, the biggest differance is in equation XXX.

We propose again a separation of variables, such that

$$v(x,y,t) = X(x)Y(y)T(t)\ .$$

This implies that equation (20) transformes to

$$\underbrace{\frac{1}{X}\frac{\partial^2 X}{\partial x^2}}_{=-\lambda_x^2} + \underbrace{\frac{1}{Y}\frac{\partial^2 Y}{\partial y^2}}_{=-\lambda_y^2} = \underbrace{\frac{1}{T}\frac{\partial T}{\partial t}}_{=-\lambda_t^2} \tag{26}$$

We allow now each term in this equation to equal a constant ($-\lambda_x^2, -\lambda_y^2$ and $-\lambda_t^2$). Obviously we have the relation

$$\lambda_t^2 = \lambda_x^2 + \lambda_y^2.$$

This means that the separate functions have the solutions:

$$X(x) = A\sin(\lambda_x x) + B\cos(\lambda_x x)$$
$$Y(x) = C\sin(\lambda_y y) + D\cos(\lambda_y y)$$
$$T(t) = Ee^{-\lambda_t^2 t} = Ee^{-(\lambda_x^2+\lambda_y^2)t}$$

Now the boundary condition give us that:

$$(22) \Rightarrow B = 0$$
$$(24) \Rightarrow D = 0$$
$$(23) \Rightarrow \lambda_x = n\pi/L$$
$$(25) \Rightarrow \lambda_y = m\pi/H$$

The initial condition (21) now give us that

$$x - 1 = \sum_{n=1}^{\infty} A_n \sin\left(\frac{n\pi x}{L}\right) \sum_{m=1}^{\infty} C_m \sin\left(\frac{m\pi y}{H}\right)$$

We let $F_{nm} = A_n \cdot C_m \cdot E$ and fourier transform, resulting in

$$F_{nm} = \frac{4}{LH}\int_0^L \int_0^H \left(\frac{x}{L} - 1\right)\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{H}\right)\ dy\ dx$$

$$= \left(\frac{4}{LH}\right)\left(\frac{H}{m\pi}\right)(1 - \cos(m\pi)) \underbrace{\int_0^L \left(\frac{x}{L} - 1\right)\sin\left(\frac{n\pi x}{L}\right)\ dx}_{=-\left(\frac{L}{n\pi}\right)}$$

$$= \frac{4}{nm\pi^2}\left(\cos(m\pi) - 1\right)$$

We can now write the analytical solution as

$$v(x,y,t) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} \frac{4}{nm\pi^2}\left(\cos(m\pi)-1\right)\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{H}\right)e^{-\left(\left(\frac{n\pi}{L}\right)^2+\left(\frac{m\pi}{H}\right)^2\right)t}$$

Setting $L = H = 1$ This reduces to

$$v(x,y,t) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} \frac{4}{nm\pi^2}\left(\cos(m\pi)-1\right)\sin\left(n\pi x\right)\sin\left(m\pi y\right)e^{-\left((n\pi)^2+(m\pi)^2\right)t}$$

As I dont want to do alot of divisions by one, I only used this last expression in my code. Once we have $v(x,y,t)$ we can easily convert it back to $u(x,y,t)$ by adding the steady state solution $u_s(x)$.

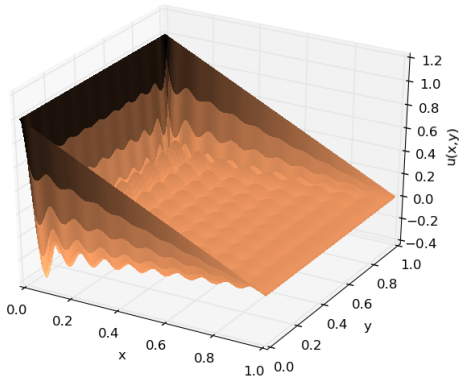

Figure 14: Analytical initial state letting n and m only run up to 20.

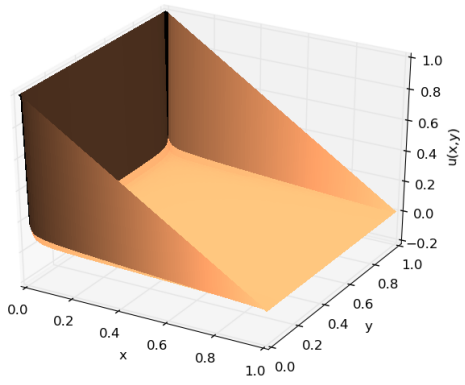

Figure 15: Analytical initial state letting n and m run up to 200.

Figure 14 and Figure 15 illustrate how the limits of n and m affect the solution. As we have to have som limit on them we will never be able to simulate the analytical solution perfectly, however at some point, including more terms will not have any significace, so we can be satisfied if we lower our standards from "perfect" to "good enough". Off course, including more terms means more operations and more patient waiting.

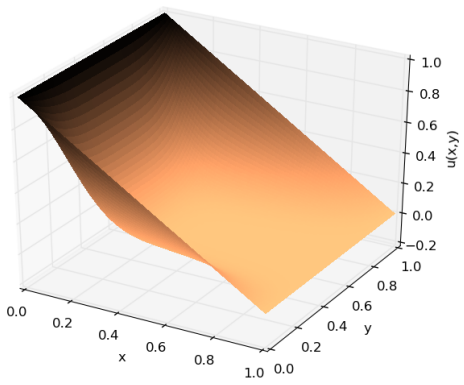

Figure 16: Analytical solution at some intermidiate time.

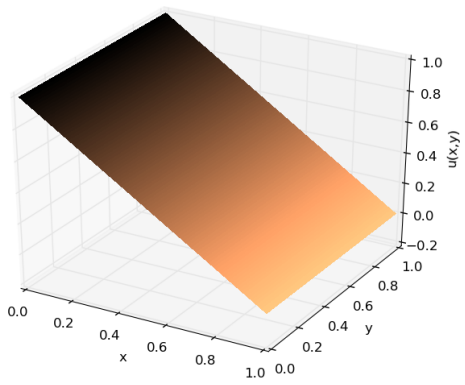

Figure 17: Analytical solution at the steady state.

As we can see from the figures above the system evolves the way we would hope, toward a steady state ($u_s = 1-x$) that is linear and only depend on x.

Numerical solution (two-dimensional case)

In order to solve the two dimensional diffusion equation, we will use two schemes, one Explicit and one Implicit. And afterwards we will make a Monte Carlo simulation as well. We use again the syntax:

$$u_t = \frac{\partial u(x,y,t)}{\partial t} \qquad u_{xx} = \frac{\partial^2 u(x,y,t)}{\partial x^2} \qquad u_{yy} = \frac{\partial^2 u(x,y,t)}{\partial y^2} \ ,$$

and the discretization:

$$\begin{aligned} x_i &= i\Delta x \\ y_j &= j\Delta y \\ t_l &= l\Delta t \\ u^l_{i,j} &= u(x_i, y_j, t_l) \ . \end{aligned}$$

As we are working with a square lattice with equally many mesh points in the x- and y directions, we have $\Delta x = \Delta y = h$.

The Explicit scheme is quite simple, and we derive the algorithms the same way we did for the one-dimensional case (page 3). For the time derivative we use the forward-going Euler formula, which when using the discretization is written as

$$u_t \approx \frac{u^{l+1}_{i,j} - u^l_{i,j}}{\Delta t}.$$

For the double derivatives with respect to x and y, we have:

$$u_{xx} \approx \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2} \qquad u_{yy} \approx \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2} \ .$$

Using these approximations our equation (20) reads:

$$\frac{u^{l+1}_{i,j} - u^l_{i,j}}{\Delta t} = \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2} + \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2}$$

Defining $\alpha = \Delta t/h^2$ and organizing by known and unknown terms we end up with

$$u^{l+1}_{i,j} = u^l_{i,j} + \alpha\left(u^l_{i+1,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1} - 4u^l_{i,j}\right). \tag{27}$$

This is the algorithm for the explicit scheme in two dimensions. The left side is the only unknown term, as the right hand side is entirely determined by the boundary and initial condition. Thus we can solve for u^l, and use this as the u^{l-1} in the next step. And so on. As I said, quite simple. Lets try to find the **stability** of this scheme. We do this the same way as we did in the one-dimensional case. First we rewrite equation (27) as a matrix-vector problem.

$$\mathbf{u}_j = \hat{\mathbf{A}}\mathbf{u}_{j-1} \qquad , \hat{\mathbf{A}} = \hat{\mathbf{I}}\,(1-4\alpha) + \alpha\hat{\mathbf{B}}$$

where

$$\hat{\mathbf{B}} = \begin{pmatrix} 0 & 1 & & & \\ 1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 0 & 1 \\ & & & 1 & 0 \end{pmatrix} \ .$$

The eigenvalues of $\hat{\mathbf{A}}$ are $1-4\alpha+\alpha\mu$, mu being the eigenvalues of $\hat{\mathbf{B}}$. In order to find μ, note that the matrix elements of $\hat{\mathbf{B}}$ are

$$b_{i,j} = \delta_{i+1,j} + \delta_{i-1,j}$$

The eigenequation for component i is then

$$\left(\hat{\mathbf{B}}\hat{\mathbf{x}}\right)_i = \sum_{j=1}^{n} \left(\delta_{i+1,j} + \delta_{i-1,j}\right) x_j = x_{i+1} + x_{i-1} = \mu_i x_i$$

Lets now expand x in the basis $x = \{\sin(\theta), \sin(2\theta), \cdots, \sin(n\theta)\}$ We have

$$\sin\left((i+1)\theta\right) + \sin\left((i-1)\theta\right) = \mu_i x_i$$

Following the prosedure in the lecture notes, page 308, I rewrote this as

$$-\underbrace{\Big[2\sin(i\theta)-\sin((i+1)\theta)-\sin((i-1)\theta)\Big]}_{2\,(1-\cos\theta)\sin(i\theta)}+2\sin(i\theta)=\mu_i x_i$$

$$\sin(i\theta)\Big[2\,(cos\theta-1)+1\Big]=\mu_i x_i$$

Leaving us with the eigenvalues of $\hat{\mathbf{B}}$ as

$$\mu_i = 2cos\theta - 1\ .$$

Meaning that the eigenvalues of $\hat{\mathbf{A}}$ are

$$\begin{aligned}\lambda &= 1-4\alpha+\alpha\,(2\cos\theta-1)\\ &= 1-5\alpha+\alpha\,(2\cos\theta)\end{aligned}$$

In order for (27) to converge toward the correct value/be stable the spectral radius of $\hat{\mathbf{A}}$ must be smaller than 1. Thus we have

$$\begin{aligned}-1 &< 1-5\alpha+\alpha\,(2\cos\theta) < 1\\ -2 &< -5\alpha+\alpha\,(2\cos\theta) < 0\\ -2 &< \alpha\,(2\cos\theta-5) < 0\end{aligned}$$

As $\alpha > 0$, this causes the restraint

$$\alpha < \frac{2}{7}$$

Thus the explicit scheme in the two-dimensional case is only stable as long as $\Delta t/h^2 < 2/7$.

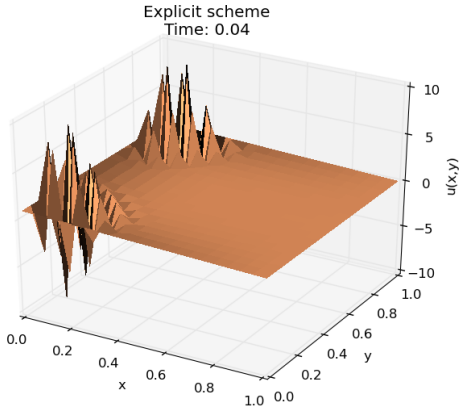

Figure 18: Explicit scheme using $\alpha = 2/5$.

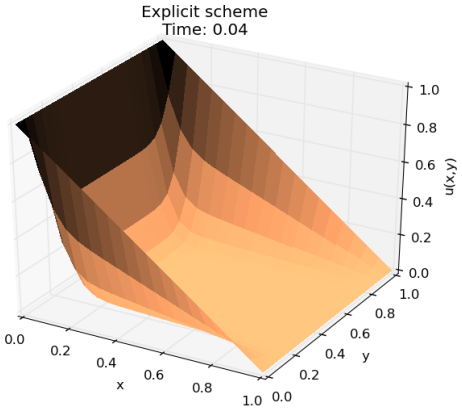

Figure 19: Explicit scheme using $\alpha = 1/10$.

The Implicit scheme is a bit more tricky. The double derivatives with respect to the spatial variables are the same as for the explicit scheme

$$u_{xx} \approx \frac{u^l_{i+1,j}-2u^l_{i,j}+u^l_{i-1,j}}{h^2} \qquad u_{yy} \approx \frac{u^l_{i,j+1}-2u^l_{i,j}+u^l_{i,j-1}}{h^2}\ .$$

The difference is in the approximation of the time derivative, wher fore the implcit scheme we now use the backward Euler formula

$$u_t \approx \frac{u^l_{i,j}-u^{l-1}_{i,j}}{\Delta t}.$$

Using the α defined the same way as for the explicit scheme, reorganizing a bit we get

$$u_{i,j}^{l} = \frac{1}{1+4\alpha}\left[\alpha\left(u_{i+1,j}^{l} + u_{i-1,j}^{l} + u_{i,j+1}^{l} + u_{i,j-1}^{l}\right) + u_{i,j}^{l-1}\right]. \tag{28}$$

The difficulty solving this equation is that the only known term is $u_{i,j}^{l-1}$! The way we go about solving this though is quite simple. We make a guess for all elements in the matrix u^l lets call this guess $u^{l,1}$. We then try to make an improved guess $u^{l,2}$ by solving (28). What I try to express is

$$u_{i,j}^{l,k+1} = \frac{1}{1+4\alpha}\left[\alpha\left(u_{i+1,j}^{l,k} + u_{i-1,j}^{l,k} + u_{i,j+1}^{l,k} + u_{i,j-1}^{l,k}\right) + u_{i,j}^{l-1}\right]. \tag{29}$$

We see that our new guess is based on our prevous guess. Most likely $k+1$ is a better guess than k. Equation (29) could be written as a matrix-vector problem, as we did for the one-dimensional case. In which case we would have the matrix $\hat{A}$ to have diagonal elements $1+4\alpha$ and non-diagonal elements α. Thus it would be positive definite, and diagonal dominant. This means that the spectral radius of the matrix is less than 1, and thus an iterative method such as (29) converges. I belive this means that the implicit scheme is stable for all values of α (Δt and h).

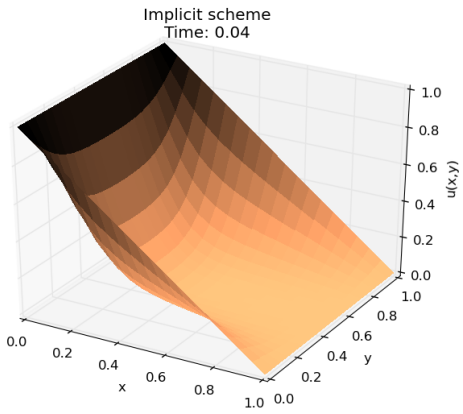

Figure 20: Implicit scheme using $\alpha = 2/5$.

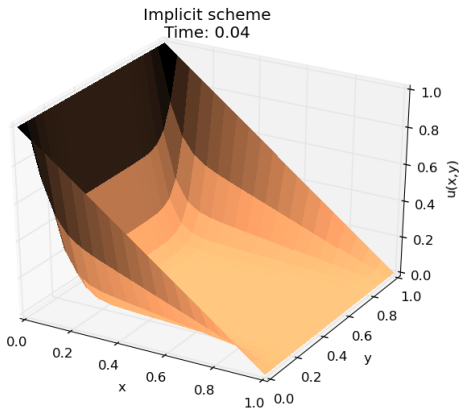

Figure 21: Implicit scheme using $\alpha = 1/10$.

The above figures are just to show that where the explicit scheme failed (Figure 18) the implicit did not. And where the explicit scheme was stable, we got seemingly the same result.

The implicit scheme gain accuracy at each iteration, however at some point a new iteration/guess wouldn't make any significant difference. There are probably many ways choose how many iterations we shall use. We could do just a fixed number of iterations at each time step, or we could have an test that checks if the difference between two guesses is small enough we to stop iterating. I couldn't see how important this is for the results, and I figured storing every guessed matrix and compairing them would slow down the program a whole lot. Therefor I simply had a fixed number of iterations, 20 I believe, in my code. As the first guess I always guess a matrix where every element is zero.

Comparing the two

It isn't as easy to compair the two schemes as it was for the one-dimensional case. I mean, I cant show three surface-plots in one figure. I chose to compare the sum of every point in the matrices of the schemes with the sum of all points in the analytical matrix. I did this at several time steps, so that we can see how the schemes differ from the analytical solution. We can off cours not see where, or how they differ from this plot, but how mush in total they do.

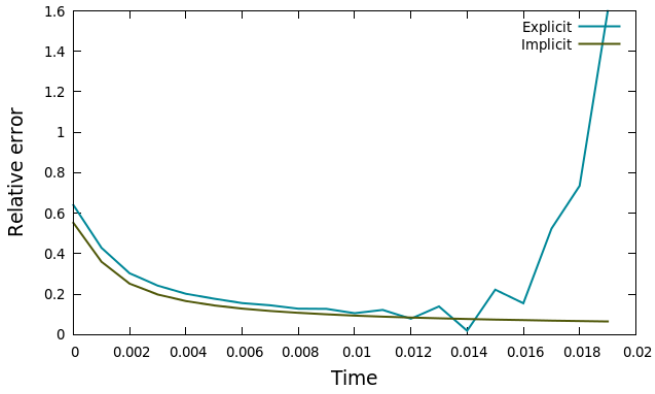

Figure 22: Comparison of the two schemes and the analytical solution with $\alpha = 2/5$.

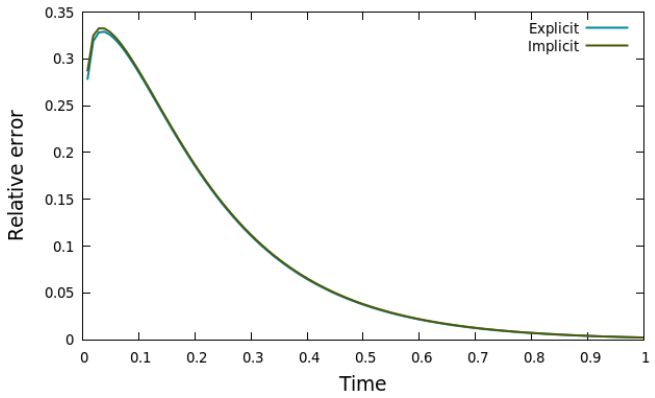

Figure 23: Comparison of the two schemes and the analytical solution with $\alpha = 1/10$.

The comparisons show that the explicit scheme is unstable at $\alpha = 2/5$, while the implicit converges toward the analytical solution. At $\alpha = 1/10$ we see that both converges toward the analytical solution. The reason for the huge error in the beginning of Figure 23 I belive is caused by the fact that the schemes converges at different rates depending on α, and therefor they both change more rapidly than the analytical solution. In the beginning, when all points but the boundary are zero (see Figure 15) , this will have a big effect as we see.

Monte Carlo simulation the two dimensional system

Extending the code made for the one-dimensional case, was fairly simple. We still model the same system, only now particles are given a y-position and have the same movement properties in the y-direction as in the x-direction, which are the same as before (gaussian distributed step length).

I kept the boundary conditions in x, meaning that there should be a constant number of particles inside the small interval $x = [0, dx]$, and none at $x = l$. However, we must also define the boundary conditions at $y = 0$ and $y = 1$. I decided to have periodic boundary conditions at the two new boundaries $u(x, 0, t)$ and $u(x, L, t)$. Meaning that a particle moving from inside the system and across the bottom border, it would keep its x-position, but the new y-position would be the distance it had passed the bottom boarder bellow the top border, if that makes any sence. The thing is by doing so we achieve a situation equivalent to having y be infinatly long, but x still fixed to be 1 wide. Which again means that our system is equivalent to the one-dimentional system. This is the reason I chose these condition, because I then knew what I should expect.

In order to keep track of the particles y-positions I added a new vector. I made sure there was a correlation between the two position vectors so that the i'th element in the x-position vector and the i'th element in the y-position vector represented the same particle. If a particle exits the system ($x < 0$ or $x > 1$) we then have to remove both these elements. The way I went about adding and removing particles, was by using two aditional vectors, which would reprecent the "next" positions in x and y. I tested every particle, as they moved, if they were inside the system after the move. If they were, I add them to the updated vectors, and if they aren't I didn't. That way we don't have to bother "deleting" them, and the code is simpler to understand.

In order to keep the number of particles inside the dx-interval constant, I implemented a counter (cc in the code) which counter how many particles left the interval, and how many particles entered. Then we add new particles after all the movements are done, and they are placed uniformaly along the line $x = dx/2$. This seemed to be a reasonable choise. The reason we add them at the x-position $x = dx/2$ is described at page 9.

In order to make a figure out of this system i sliced the system into a NxN grid. I then checked every single particle to find which interval in the x-direction it was in, and then which interval in the y-direction it was in. I then summed up all the particles that was inside a square, and divided by the total amount of particles in the system, to get the concentration at that specific square. I stored this value in a (N-1)x(N-1) matrix, which had one element for each square.

The results from the simulations can be seen bellow.

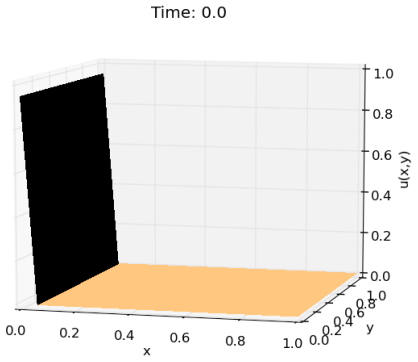

Figure 24: Initial state of the Monte Carlo simulation in two dimentions.

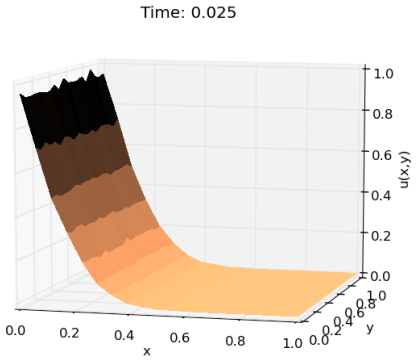

Figure 25: Monte Carlo simulation in two dimentions at an early stage.

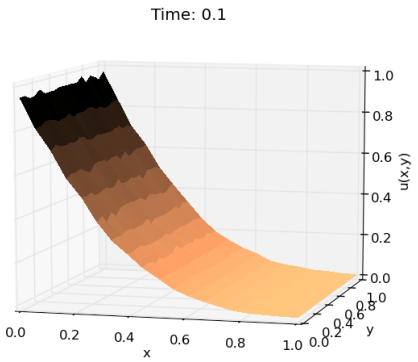

Figure 26: Monte Carlo simulation in two dimentions at a state close to the steady state.

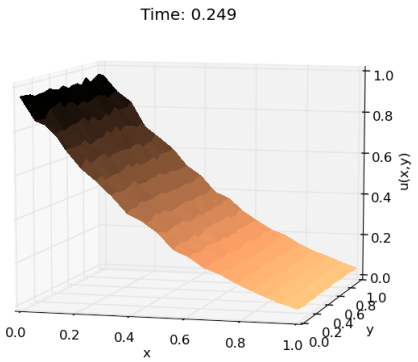

Figure 27: Steady state of the Monte Carlo simulation in two dimentions.

We see here that the system evolves just as it did for the one-dimensional system, as it should. You may notice the small bumps and lumps on the figures. These are a consequense of the random walks. However, if we increase the number of particles these effects will decrease. In this simulation I used 100000 particles.

Project 4 main.cpp

```
#include <iostream>
#include<armadillo>
#include <stdlib.h> /* atof function */
#include <math.h>   /* sine function */
#include <stdio.h>  /* printf function */
#include <fstream>
#include <iomanip>
#include <plot.h>
#include <schemes.h>
using namespace std;
using namespace arma;

vec Analytical(double t, double L , vec x)
{
    vec A = zeros(x.n_elem);
    for (int n=1; n<50; n++)
    {
        A +=  (2/(n*M_PI)) * (cos(n*M_PI)*(1-L) - 1) * sin(M_PI*n*x/L) * exp(-pow((M_PI*n/L)
    }
    return A;
}

int main()
{

    int N = 10;
    double dx = 1./N;
    double dt = 0.5*dx*dx; //Limit of the explicit schemes stability.
    double L = 1;  //Can be chosen freely.
    vec x = linspace(0,L,N);
    vec v = x/L -1;      //Initial condition/state
    vec Anal;    //haha...
    vec CN, E, I, CNu, Eu, Iu;
    Schemes One(dx, dt);     //Just some object. Random name.
    int c = 0;
    E = One.Explicit(v);
    I = One.Implicit(v);
    CN = One.Crank_Nicolson(v);
    for (double t=0; t<0.5;t+=dt)
    {
        E = One.Explicit(E);
        I = One.Implicit(I);
        CN = One.Crank_Nicolson(CN);
        //Transforming back v to u
        Eu = E+(1-x/L);
        Iu = I+(1-x/L);
        CNu = CN +(1-x/L);
        fstream plotdata4c;
        char str[100];
        c+=1;
        if (c == 10 || c== 20|| c==30)  //Writes some cases to files. Used to make plots.
        {
            Anal = Analytical(t,L,x) + (1-x/L);    // Calculates the analytical solution.
            sprintf(str, "/home/filiphl/Desktop/FYS3150/project4/project4/plotdata/plotdata4c
            plotdata4c.open(str, ios::out);
            plotdata4c << setw(20) <<setprecision(10)<<"x"
```

```
                    << setw(20) <<setprecision(10)<< "Explicit"
                    << setw(20) <<setprecision(10)<< "Implicit"
                    << setw(20) <<setprecision(10)<< "Crank Nicolson"
                    << setw(20) <<setprecision(10)<< "Analytical"
                    << setw(20) <<setprecision(10)<< "Diff. Explicit"
                    << setw(20) <<setprecision(10)<< "Diff. Implicit"
                    << setw(20) <<setprecision(10)<< "Diff. Crank Nicolson"<<endl;
            for (int i=0; i<N; i++)
            {
                plotdata4c << setw(20) <<setprecision(10)<<x(i)
                           << setw(20) <<setprecision(10)<<Eu(i)
                           << setw(20) <<setprecision(10)<<Iu(i)
                           << setw(20) <<setprecision(10)<<CNu(i)
                           << setw(20) <<setprecision(10)<<Anal(i)
                           << setw(20) <<setprecision(10)<<abs( (Anal(i)-Eu(i) ) / Anal(i))
                           << setw(20) <<setprecision(10)<<abs( (Anal(i)-Iu(i) ) / Anal(i))
                           << setw(20) <<setprecision(10)<<abs( (Anal(i)-CNu(i) ) / Anal(i))
            }
            plotdata4c.close();
        }
    }

    return 0;
}
```

Project 4 schemes.cpp

```
/*
 *  These functions will only do one single iteration
*/

#include "schemes.h"
using namespace arma;

Schemes::Schemes(double dx, double dt)
{
    alpha = dt/(dx*dx);
}

vec Schemes::Explicit(vec u)
{
    int n = u.n_elem;
    double a = alpha;
    double b = 1-2*alpha;
    vec y(n);
    for (int i=1; i<n-1; i++)
    {
        y(i) = u(i-1)*a + u(i)*b + u(i+1)*a;
    }
    y(0) = 0;       //Boundary condition
    y(n-1) = 0;   //Boundary condition

    return y;
}

vec Schemes::Implicit(vec y)
{
    int n = y.n_elem;
    // Au=y    , A is a matrix and u and y are vectors.
    // Elements of A are all constants so there is no need to construct a matrix.
    double a = -alpha;
    double b = 1+2*alpha;
    vec u = Tridiag(y, a, b, a);     //Solves the tridiagonal matrix equation. Returning the
    u(0) = 0;       //Boundary condition
    u(n-1) = 0;    //Boundary condition

    return u;
}

vec Schemes::Crank_Nicolson(vec u)
{
    // Setting up the output vector
    int n = u.n_elem;
    vec y(n);

    // Calculate v0' in v0'=(2I-aB)v0
    double a = alpha;
    double b = 2-2*alpha;
    for (int i=1; i<n-1; i++)
    {
        y(i) = u(i-1)*a + u(i)*b + u(i+1)*a;
    }
    y(0) = 0;       //Boundary conditions
```

```
    y(n-1) = 0;

    //Calculate v1 in (2I+aB)v1=v0'
    y = Tridiag(y, -alpha, 2+2*alpha, -alpha);
    y(0)=0;         //Boundary conditions
    y(n-1) =0;

    return y;
}

vec Schemes::Tridiag(vec x, double a, double b, double c)
{
    int N = x.n_elem;
    vec v = vec(N); // Making bunch of vectors...
    vec g = v;
    vec p = v;

    g(0) = c/b; //a(0) = 0
    p(0) = x(0)/b;

    for (int i=1; i<N-1; i++){      //Forward substitution.
        g(i) = c/(b-(a*g(i-1)));
        p(i) = (x(i)-a*p(i-1))/(b-a*g(i-1));
    }
    g(N-1) = 0; //c(N-1)=0
    p(N-1) = (x(N-1)-a*p(N-2))/(b-a*g(N-2));


    //v(0)=0; //Initial condition
    v(N-1) = p(N-1);
    for (int i = N-2; i>=0; i--){    //Backward substitution.
        v(i) = p(i)-g(i)*v(i+1);
    }
    return v;
}
```

Project 4 schemes.h

```
#ifndef SCHEMES_H
#define SCHEMES_H
#include <armadillo>

//using namespace arma;
class Schemes
{
public:
    Schemes(){}
    Schemes(double dx, double dt);
    arma::vec Explicit(arma::vec u);
    arma::vec Implicit(arma::vec y);
    arma::vec Crank_Nicolson(arma::vec u);
    arma::vec Tridiag(arma::vec x, double a, double b, double c);
    double alpha;
};

#endif // SCHEMES_H
```

Project 5 main.cpp

```
#include <iostream>
#include<armadillo>
#include <stdlib.h> /* atof function */
#include <math.h>   /* sine function */
#include <stdio.h>  /* printf function */
#include <fstream>
#include <iomanip>
#include <plot.h>
#include <schemes.h>
#include <schemes2d.h>
#include "cppLibrary/lib.h"
#include "random.h"
#include <algorithm>

using namespace std;
using namespace arma;

mat Analytical(double t , vec x, vec y)
{
    int N = x.n_elem;
    double Fnm;
    mat Z = zeros<mat>(N,N);
    for (int i=0;i<N;i++)
    {
        for (int j=0; j<N;j++)
        {
            for (int n=1; n<50; n++)
            {
                for (int m=1; m<50; m++)
                {
                    Fnm = (4/(M_PI*M_PI*n*m)) * (cos(M_PI*m) - 1);
                    Z(j,i) +=  Fnm * sin(n*M_PI*x(i)) * sin(m*M_PI*y(j)) * exp(-(n*n*M_PI*M_
                }
            }
            Z(j,i) += 1-x(i);
        }
    }
    return Z;
}

int main()
{


    /*
    // Monte Carlo a)

    int L = 1;
    double dt = 0.01;
    double l0= sqrt(2*dt);
    int npart = 10000;
    long int idum = -1;
    double r;
    vector<double>dpos(npart);
    vector<double>new_dpos;
    double pastdpos;
```

```
    double nextdpos;

    fstream hista;
    hista.open("/home/filiphl/Desktop/figs/hist_a.txt", ios::out);

    for (double t = 0; t<1; t+=dt)
    {
        cout << t<<endl;
        //Write to file
        for (int k = 0; k<dpos.size(); k++)   {hista << setw(15) <<  dpos[k];}
        hista<<endl;

        for (int i=0; i<dpos.size(); i++)
        {
                pastdpos = dpos[i];
                r = ran0(&idum);
                if (r > 0.5){nextdpos = pastdpos + l0;}
                else {nextdpos = pastdpos - l0;}
                if ((nextdpos>0)&&(nextdpos<L)) {new_dpos.push_back(nextdpos);}
                if (pastdpos==0) {new_dpos.push_back(0);}
        }
        dpos = new_dpos;
        new_dpos.clear();
    }
    hista.close();
    //Run python script for histogram.
    system("python /home/filiphl/Desktop/figs/hist.py hist_a.txt 90");
*/




    //********Monte Carlo b)*********//
    /*
    int L = 1;
    double dt = 0.00005;
    double l0= sqrt(2*dt);
    double dx = 0.01;
    Random r(-1);
    int npart = 10000;

    vector<double> pos;
    vector<double> updatedpos(npart); // FIlled with zeroes as default.
    double sd =1/sqrt(2); //standard deviation
    double pastpos;
    double nextpos;

    fstream hist1;
    hist1.open("/home/filiphl/Desktop/figs/hist1.txt", ios::out);

    int c = 0;
    int cc = 0;
    for (double t=0; t<1; t+=dt)
    {
        //cout<<t<<endl;

        if (c==1000)
        {
            cout << 100*t <<"% done"<<endl;
```

```
        for (int k = 0; k<pos.size(); k++)  //Writes data to file.
        {
            hist1 << setw(25) << setprecision(8) <<  pos[k];
        }
        hist1 << endl;
        c = 0;
    }
    for (int i=0; i<pos.size(); i++)
    {
        pastpos = pos[i];
        pos[i] += r.nextGauss(0.0, sd)*l0;
        nextpos = pos[i];

        //Includes relevant values.
        if ( (nextpos>=0)&&(nextpos<L) ) { updatedpos.push_back(nextpos); }

        // Partical moving from inside the dx-interval to outside
        if ( (0<=pastpos) && (pastpos<dx) )
        {
            if ( (nextpos<0) || (nextpos>dx) ) {cc--;}
        }

        //Partical moving from outside the dx-interval to inside
        if ( (pastpos>dx) && (nextpos<dx) && (nextpos>0) ) {cc++;}
    }

    while (cc < 0)
    {
        updatedpos.push_back(dx/2); // Notice the position!
        cc++;
    }
    pos = updatedpos;
    updatedpos.clear();
    c++;
}
system("python /home/filiphl/Desktop/figs/hist.py hist1.txt 24");
*/

/*
// Monte Carlo 2D

int L = 1;
double dt = 0.001;
double l0= sqrt(2*dt);
double dx = 0.05;   // dy=dx
Random r(-1);
int npart = 10000;

vector<float> xpos;  // FIlled with zeroes as default.
vector<float> ypos;
for (float j=0; j<npart; j++) { xpos.push_back(dx/2); ypos.push_back(j/npart); }
vector<float> new_xpos;
vector<float>new_ypos;
float sd =1/sqrt(2); //standard deviation
float pastxpos, pastypos, nextxpos, nextypos;
long int idum = -1;
int nbins = L/dx-1;
float norm = npart/nbins;
mat M;
int c = 1;
```

```
int a = 0;
int cc = 0;
float y;
for (float t=0; t<0.25; t+=dt)
{
    if (c==1)
    {
        cout << 100*t/0.25 <<"% done"<<endl;

        M = zeros<mat>(nbins, nbins);
        for (int i=0; i<xpos.size(); i++)
        {
            for (int row=0; row<nbins; row++)
            {
                for (int col=0; col<nbins; col++)
                {
                    if ( (xpos[i]>=row*dx) && (xpos[i]<(row+1)*dx) )
                    {
                        if ( (ypos[i]>=col*dx) && (ypos[i]<(col+1)*dx) )  {M(col,row)+=1
                    }
                }
            }
        }
        fstream mc2;
        char str[100];
        sprintf(str, "/home/filiphl/Desktop/figs/mc2d%d.txt", a);
        mc2.open(str, ios::out);

        for (int row=0; row<nbins; row++)
        {
            for (int col=0; col<nbins; col++)
            {
                mc2<< setw(15)<<M(row,col)/norm;
            }
            mc2<<endl;
        }
        a+=c;
        c = 0;
    }

    for (int i=0; i<xpos.size(); i++)   //xpos and ypos should have equal size.
    {
        pastxpos = xpos[i];
        pastypos = ypos[i];
        xpos[i] += r.nextGauss(0.0, sd)*l0;
        ypos[i] += r.nextGauss(0.0, sd)*l0;
        nextxpos = xpos[i];
        nextypos = ypos[i];

        //Includes relevant values.
        if ( (nextxpos>0)&&(nextxpos<L) )
        {
            new_xpos.push_back(nextxpos);
            if ( (nextypos>=0)&&(nextypos<=L) ) { new_ypos.push_back(nextypos); }
            // Particles moving for inside y=[0,1] to outside: periodical boundary condi
            else if ( (pastypos<=L) && (nextypos>L) ) { new_ypos.push_back(nextypos-L); }
            else if ( (pastypos>=0) && (nextypos<0) ) { new_ypos.push_back(nextypos+L); }
        }

        // Partical moving from inside the dx-interval to outside
```

```
            if ( (0<=pastxpos) && (pastxpos<dx) )
            {
                if ( (nextxpos<0) || (nextxpos>dx) ) {cc--;}
            }

            //Partical moving from outside the dx-interval to inside
            if ( (pastxpos>dx) && (nextxpos<dx) && (nextxpos>0) ) {cc++; }
        }

        // Add new elements at x = dx/2 and y = [0,1]
        y = cc;
        while (cc < 0)
        {
            new_xpos.push_back(dx/2.0);
            new_ypos.push_back(cc/y);
            cc++;
        }

        xpos = new_xpos;
        ypos = new_ypos;
        new_xpos.clear();
        new_ypos.clear();

        c++;
    }
*/
    /*

    int N = 20;
    double dt = 0.001;  // 0.001;
    double dx = 0.1; //gives alpha = 0.4
    double t = 0;
    double timelimit = 1;
    int c = 0;
    int inst = 1;
    vec x = linspace(0,1,N);
    vec y = linspace(0,1,N);
    mat U=zeros<mat>(N,N);
    U.col(0) = 1-x;
    U.col(N-1)=1-x;
    U.row(0) = ones<mat>(1,N);
    mat V0(N,N);

    for (int i = 0; i<N; i++)
    {
        V0.col(i)=U.col(i)+x-1;
    }

    Schemes2d Two(dx,dt);
    mat E = ones<mat>(N,N);
    mat EU = ones<mat>(N,N);
    mat I =  ones<mat>(N,N);
    mat IU = ones<mat>(N,N);
    vec relerrorexp = zeros<vec>(timelimit/dt);
    vec relerrorimp = zeros<vec>(timelimit/dt);
    mat Analytic;
    cout << timelimit/dt<<endl;
    E = Two.Explicit2d(V0);
    I = Two.Implicit2d(V0);
```

```
    fstream relerror;
    relerror.open("/home/filiphl/Desktop/figs/relerror.txt", ios::out);

    while ( t<timelimit)
    {
        fstream plotE, plotI;
        char strE[100], strI[100];
        E = Two.Explicit2d(E);
        I = Two.Implicit2d(I);

        for (int i=0; i<N; i++)
        {
            EU.col(i) = E.col(i) + 1-x;       //Converting back to u from v
            IU.col(i) = I.col(i) + 1-x;
        }

        if (inst == 10)
        {
            Analytic = Analytical(t,x,y);
            relerrorexp(c) += abs((accu(Analytic)-accu(EU))/accu(Analytic));
            cout <<c<<endl;
            relerrorimp(c) += abs((accu(Analytic)-accu(IU))/accu(Analytic));
            sprintf(strE, "/home/filiphl/Desktop/figs/plotE%d.txt", c);
            sprintf(strI, "/home/filiphl/Desktop/figs/plotI%d.txt", c);
            plotE.open(strE, ios::out);
            plotI.open(strI, ios::out );

            for (int i=0; i<N; i++)
            {
                for (int j=0; j<N; j++)
                {
                    plotE << setw(20) <<setprecision(10)<<EU(j,i);
                    plotI << setw(20) <<setprecision(10)<<IU(j,i);
                }
                plotE << endl;
                plotI << endl;
            }
            relerror << setw(20) <<setprecision(10)<< t <<setw(20) <<setprecision(10)<<relerr
            relerror <<endl;
            inst = 0;
        }
        inst +=1;
        c+=1;
        t+=dt;
    }
    relerror.close();
*/

    /*      //Makes a figure of the analytical solution
    double t = 0.01;
    int N = 100;
    vec x = linspace(0,1,N);
    vec y = linspace(0,1,N);

    mat A = Analytical(t,x,y);    // Calculates the analytical solution.
    char str[50];
    sprintf(str, "/home/filiphl/Desktop/figs/plotd1.txt");
    fstream plot1;
    plot1.open(str, ios::out);
    for (int i=0; i<N; i++)
```

```
    {
        for (int j=0; j<N; j++)
        {
            plot1 << setw(20) <<setprecision(10)<<A(i,j);
        }
        plot1 << endl;
    }

*/
    return 0;
}
```

Project 5 schemes.cpp

```
/*
 *  These functions will only do one single iteration
*/

#include "schemes.h"
using namespace arma;

Schemes::Schemes(double dx, double dt)
{
    alpha = dt/(dx*dx);
}

vec Schemes::Explicit(vec u)
{
    int n = u.n_elem;
    double a = alpha;
    double b = 1-2*alpha;
    vec y(n);
    for (int i=1; i<n-1; i++)
    {
        y(i) = u(i-1)*a + u(i)*b + u(i+1)*a;
    }
    y(0) = 0;       //Boundary condition
    y(n-1) = 0;   //Boundary condition

    return y;
}

vec Schemes::Implicit(vec y)
{
    int n = y.n_elem;
    // Au=y    , A is a matrix and u and y are vectors.
    // Elements of A are all constants so there is no need to construct a matrix.
    double a = -alpha;
    double b = 1+2*alpha;
    vec u = Tridiag(y, a, b, a);     //Solves the tridiagonal matrix equation. Returning the
    u(0) = 0;       //Boundary condition
    u(n-1) = 0;    //Boundary condition

    return u;
}
```

```
vec Schemes::Crank_Nicolson(vec u)
{
    // Setting up the output vector
    int n = u.n_elem;
    vec y(n);

    // Calculate v0' in v0'=(2I-aB)v0
    double a = alpha;
    double b = 2-2*alpha;
    for (int i=1; i<n-1; i++)
    {
        y(i) = u(i-1)*a + u(i)*b + u(i+1)*a;
    }
    y(0) = 0;       //Boundary conditions
    y(n-1) = 0;

    //Calculate v1 in (2I+aB)v1=v0'
    y = Tridiag(y, -alpha, 2+2*alpha, -alpha);
    y(0)=0;         //Boundary conditions
    y(n-1) =0;

    return y;
}

vec Schemes::Tridiag(vec x, double a, double b, double c)
{
    int N = x.n_elem;
    vec v = vec(N); // Making bunch of vectors...
    vec g = v;
    vec p = v;

    g(0) = c/b; //a(0) = 0
    p(0) = x(0)/b;

    for (int i=1; i<N-1; i++){      //Forward substitution.
        g(i) = c/(b-(a*g(i-1)));
        p(i) = (x(i)-a*p(i-1))/(b-a*g(i-1));
    }
    g(N-1) = 0; //c(N-1)=0
    p(N-1) = (x(N-1)-a*p(N-2))/(b-a*g(N-2));


    //v(0)=0; //Initial condition
    v(N-1) = p(N-1);
    for (int i = N-2; i>=0; i--){    //Backward substitution.
        v(i) = p(i)-g(i)*v(i+1);
    }
    return v;
}
```

Project 5 schemes2d.cpp

```
#include "schemes2d.h"
using namespace arma;

Schemes2d::Schemes2d(double dx, double dt)
{
    alpha = dt/(dx*dx);
    beta = 1/(1+4*alpha);
}

mat Schemes2d::Explicit2d(mat U)
{
    int n = U.n_cols;    //Assuming nxn-matrix
    mat V = zeros<mat>(n,n);
    for (int i=1; i<n-1; i++)
    {
        for (int j=1; j<n-1; j++)
        {
            V(i,j) = U(i,j) + alpha*(  U(i+1,j) + U(i-1,j) + U(i,j+1) + U(i,j-1) - 4*U(i,j)
);
        }
    }
    //The boundary conditions are conserved as they are not changed at all.
    return V;
}

mat Schemes2d::Implicit2d(mat U)
{
    int n = U.n_cols;     //Assuming nxn-matrix
    mat V1 = U;           // Past iteration
    mat V2;                  // Precent iteration
    for (int k=0; k<30; k++)
    {
        V2 = zeros<mat>(n,n);    // Guess of a solution.
        for (int i=1; i<n-1; i++)
        {
            for (int j=1; j<n-1; j++)
            {
                V2(i,j) = beta*( alpha*( V1(i+1,j) + V1(i-1,j) + V1(i,j+1) + V1(i,j-1) ) + U(
);
            }
        }
        V1 = V2;
    }
    return V2;
}
```

Project 5 schemes.h

```
#ifndef SCHEMES_H
#define SCHEMES_H
#include <armadillo>

//using namespace arma;
class Schemes
{
public:
    Schemes(){}
    Schemes(double dx, double dt);
    arma::vec Explicit(arma::vec u);
    arma::vec Implicit(arma::vec y);
    arma::vec Crank_Nicolson(arma::vec u);
    arma::vec Tridiag(arma::vec x, double a, double b, double c);
    double alpha;
};

#endif // SCHEMES_H
```

Project 5 schemes2d.h

```
#ifndef SCHEMES2D_H
#define SCHEMES2D_H

#include <armadillo>

//using namespace arma;
class Schemes2d
{
public:
    Schemes2d(){}
    Schemes2d(double dx, double dt);
    arma::mat Explicit2d(arma::mat U);
    arma::mat Implicit2d(arma::mat U);
    arma::vec Tridiag(arma::vec x, double a, double b, double c);
    double alpha;
    double beta;
};
#endif // SCHEMES2D_H
```