Diffusion in 1 and 2 dimensions

Kandidat nr. 114

University of Oslo

Abstract

In this project we will model the diffusion of neurotransmitters in the synaptic cleft. This is the dominant mechanism for transporting signals between neurons. It is a continuation of Project 4, where we solve the problem using the Monte Carlo Random Walk scheme in one and two dimensions. We use both a constant step length and a variable step length weighted with the Gaussian distribution and compare these solution with the explicit results from Project 4. In two dimensions we also model the system using an explicit scheme based on the forward Euler method, and using an implicit scheme based on the Jacobi algorithm.

Contents

I. Introduction

In this section we give some information about the biological system we are investigating.

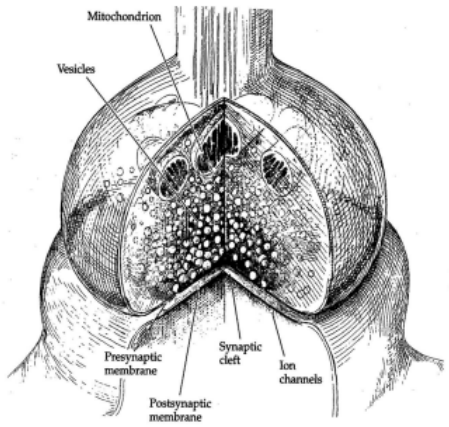

Figure 1: *A sketch of the internal components of a synapse. The top structure is the axon of a neuron and the bottom knob-like structure is the receiving neuron. Image from [3]*

The synapse is the junction through which neurons signal each other and to non-neuronal cells like muscles or glands. A sketch of a synapse is found in Fig. 1. Synapses allow neurons to form connected circuits within the body and are crucial biological components that form the basis of thought and perception.

An electrochemical wave (action potential) travels along the axon of a neuron. Once the action potential reaches the synaptic cleft, which separates the sending and receiving cells, neurotransmitters are released into it. The neurotransmitters diffuse across the synaptic cleft. Once they reach the other side of the cleft (the post-synaptic membrane) they bind to receptors and are transported across. In our model we assume the following:

- The influx of neurotransmitters from the presynaptic side is constant, thus the concentration is at a maximum there.
- Neurotransmitters reaching the receptors at the postsynaptic side is instantly absorbed through the membrane, thus the concentration there is zero.
- There are no neurotransmitters in the cleft for t<0.

When we extend the model to two dimensions we assume the neurotransmitters exits the system at the boundaries on the y axis. Mathematically this condition is given as

$$u(x,0,t) = u(x,1,t) = 0$$

II. Theory

Here we solve the diffusion equation in one and two dimensions analytically. Later we can compare these solutions to our results from the numerical methods.

I. Closed Form Solution in 1D

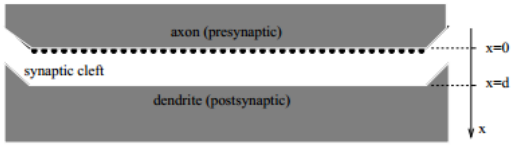

Figure 2: *A one-dimensional model of the synaptic cleft, shown here at t=0. Image from [1]*

In Fig.2 a 1D model of the synaptic cleft is shown. The distance between the pre and postsynaptic membrane is d. We now consider the diffusion of the neurotransmitters across this area. Fick's second law of diffusion in one dimension is mathematically described by

$$\frac{\partial u(x,t)}{\partial t} = D\frac{\partial^2 u(x,t)}{\partial x^2} \tag{1}$$

Here $u(x,t)$ is the concentration of particles at position x and time t, and D is the diffusion constant. The assumptions listed above is mathematically expressed as

- $u(0,t) = u_0 = 1$, for $t \geq 0$
- $u(d,t) = u(1,t) = 0$, for $t \geq 0$
- $u(x,0) = 0$, for $0 < x < 1$

Here we have set $d = 1$ (the length of the cleft) and $u_0 = 1$ (the maximum concentration). We begin by redefining the steady state solution. It can easily be found to be

$$u_s(x) = 1 - x$$

We define a new function, $v(x)$, with the properties

$$v(x) = u(x) - u_s(x)$$

and boundary conditions $v(0) = v(1) = 0$. To solve this we assume that the solution is separable and we can write.

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

Inserting this expression into Eq.1 and dividing by $v(x)v(t)$ on both sides we obtain

$$D\frac{v''(x)}{v(x)} = \frac{v'(t)}{v(t)}$$

Since the right side has to be equal to the left side we can define a constant of separation λ

$$D\frac{v''(x)}{v(x)} = \frac{v'(t)}{v(t)} = -\lambda^2$$

Thus we have two equations of two different variables.

$$v''(x) + \lambda^2 v(x) = 0$$

and

$$v'(t) + D\lambda^2 v(t) = 0$$

The solution of these equations are

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

$$v(t) = Ce^{-\lambda^2 Dt}$$

With the boundary conditions previously mentioned we find that $B = 0$ and $\lambda = n\pi$ where $n = \pm 0, \pm 1, \pm 2...$ By connecting the two solutions again we obtain

$$v(x,t) = v(x)v(t) = A_n sin(n\pi x)e^{-D(n\pi)^2 t}$$

This solution is valid for any n. Since the diffusion equation is linear, a superposition of solutions for various n will also be a solution. Thus we write the solution as a sum

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

Invoking the last boundary condition $u(x,0) = 0$ or similarly $v(x,0) = x - 1$ we obtain the following equation

$$u(x,0) = x - 1 = \sum_{n=1}^{\infty} A_n \sin(n\pi x)$$

The constant A_n can now be found from the theory of Fourier series, and it is defined as

$$A_n = 2\int_0^1 (x-1)sin(n\pi x)dx = -\frac{2}{n\pi}$$

Inserting this into the previous solution we have

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

or

$$u(x,t) = 1 - x - \frac{2}{n\pi}\sum_{n=1}^{\infty} \sin(n\pi x)e^{-D(n\pi)^2 t} \quad (3)$$

II. Spectral Radius and Stability

To require that the solution of the different schemes approaches a definite value, we need to require that the spectral radius $\rho(\hat{A})$ satisfy

$$\rho(\hat{A}) = max\left\{|\lambda| : det(\hat{A} - \lambda\hat{I})\right\} < 1$$

A general tridiagonal matrix

$$\hat{A} = \begin{bmatrix} a & b & 0 & ... & 0 \\ c & a & b & ... & 0 \\ 0 & c & a & ... & 0 \\ 0 & ... & ... & ... & b \\ 0 & ... & ... & c & a \end{bmatrix}$$

has eigenvalues $\lambda_i = a + s\sqrt{bc}\cos(i\pi/n + 1)$ where i=1:n. If all eigenvalues are lager than zero, stability is guaranteed, since the spectral radius is smaller than 1.

III. Methods

In this section we give a description of the numerical algorithms used. We also discuss convergence criteria and the numerical stability of the explicit and implicit schemes in 2D.

I. Random Walks in 1D and 2D

A random walker is an object whose movement through a generalized phase space is determined from a probability function. A random walk is a mathematical formalization of a path that consists of a succession of random steps. For example, the path traced by a molecule as it travels in a liquid or a gas, the search path of a foraging animal, the price of a fluctuating stock and the financial status of a gambler can all be modelled as random walks.

In physics, random walks are used as simplified models of physical Brownian motion and diffusion such as the random movement of molecules in liquids and gases. In our model the diffusion of neurotransmitters can be modelled by random walks by considering the following assumptions:

- Each neurotransmitter is random walker.
- The walkers have an equal probability of moving in any direction.
- At u(0,t) the walker can only move away from the pre synaptic wall.
- When a walker moves away from the wall at u(0,t), we add another one to keep the concentration there u(0,t)=1.
- If a walker previously at u(x,t) moves into u(0,t), we remove it to keep the concentration there u(0,t)=1.
- When a walker hits u(1,t) it is removed from the system.

We track the movement of the walkers over a given time and store their final position in a histogram. Repeating this process over many Monte Carlo cycles will fill the histogram and by normalizing the results from all the cycles we obtain a density distribution of the system for the given time. In our model we use the step length

$$l = \sqrt{2Ddt}$$

as suggested by Farnell and Gibson [2].

This scheme is easily extended to two dimensions by allowing the walkers to move in both the x and y direction. Naturally the probability of moving in either direction is then reduced to 25%. The boundary conditions are the same as discussed in section II-II. We will also investigate the effect of using a Gaussian distributed step length. It is intuitive that by increasing the number of walkers and/or Monte Carlo cycles the result will become more smooth as the local density fluctuation in the probability distribution will even out as the number of cycles goes to infinity.

II. Explicit scheme in 2D

The 2D diffusion equation in Eq.5 can be solved explicitly by discretizing position and time like we did in project 4. The method is quite similar only now we have to discretize space in both the x and y direction, which results in a 2D grid with cells of size dxdy. Using the Euler definition derivatives we obtain

$$u_{xx} = \frac{u(x+h,y,t) - 2u(x,y,t) + u(x-h,y,t)}{(dx)^2}$$

or in shorthand notation

$$u_{xx} = \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{dx^2}$$

here i,j and k denotes $x_i = x_0 + ih$, $y_j = y_0 + jh$ and $t_k = t_0 + kdt$. The expression for the u_{yy} is identically given as

$$u_{yy} = \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j,k-1}}{dy^2}$$

Finally, u_t is obtained from the forward Euler formula and discretized as

$$u_t = \frac{u_{i,j,k+1} - u_{i,j,k}}{dt}$$

Now by setting $u_{xx} + u_{yy} = u_t$ and solving for $u_{i,j,k+1}$ we obtain the final result

$$u_{i,j,k+1} = (1 - 4\alpha)\, u_{i,j,k} + \alpha \Delta_{i,j,k} \qquad (4)$$

where $\Delta_{i,j,k}$ is defined as

$$\Delta_{i,j,k} = u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k}$$

Since $dx = dy$ in this model, $\alpha = dt/dx^2$ and Eq.5 can be rewritten as a matrix vector multiplication. By defining the matrix $\hat{A}$ as

$$\hat{A} = \begin{bmatrix} 1-4\alpha & \alpha & 0 & ... & 0 \\ \alpha & 1-4\alpha & \alpha & ... & 0 \\ 0 & \alpha & 1-4\alpha & ... & 0 \\ 0 & ... & ... & ... & \alpha \\ 0 & ... & ... & \alpha & 1-4\alpha \end{bmatrix}$$

and the two vectors $\vec{U}_{j,k}$ and $\vec{U}_{i,k}$ as

$$\vec{U}_{j,k} = \begin{bmatrix} u_{1,j,k} \\ u_{2,j,k} \\ u_{3,j,k} \\ \cdots \\ u_{n,n,k} \end{bmatrix} \vec{U}_{i,k} = \begin{bmatrix} u_{i,1,k} \\ u_{i,2,k} \\ u_{i,3,k} \\ \cdots \\ u_{i,n,k} \end{bmatrix}$$

We can then write the problem as the following matrix-vector multiplication

$$U_{i,j,k+1} = \hat{A}\left(\vec{U}_{j,k} + \vec{U}_{i,k}\right)$$

The stability of this scheme is guaranteed by the spectral radius theorem discussed in section II-III as long as

$$\alpha < \frac{1}{4}$$

This stability condition is even more restrictive than in the corresponding 1D case, which we found in project 4 to be $\alpha < 1/2$. Thus we need a even lower dt for a fixed dx. Depending on the time-scale of the system we are investigating the explicit 2D algorithm may be too data consuming for practical use. The code we used for implementing this algorithm is found in section VI-III.

III. Implicit scheme in 2D

We can also write an implicit algorithm for solving the 2D diffusion equation. The procedure is similar to the explicit case, and we define again u_{xx} and u_{xy} as

$$u_{xx} = \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{dx^2}$$

and

$$u_{yy} = \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j,k-1}}{dy^2}$$

We now use the backward going Euler formula for the first derivative in time, in its discretized form we have

$$u_t = \frac{u_{i,j,k} - u_{i,j,k-1}}{dt}$$

Adding these terms together we obtain the final result

$$u_{i,j,k} = \frac{1}{1 + 4 * \alpha}\left[\alpha \Delta_{i,j,k} + u_{i,j,k-1}\right] \qquad (5)$$

Here $\Delta_{i,j,k}$ is defined the same way as before. The matrix for this equation is

$$\hat{A} = \begin{bmatrix} 1+4\alpha & -\alpha & ... & -\alpha & 0 \\ -\alpha & 1+4\alpha & 0 & ... & -\alpha \\ ... & 0 & 1+4\alpha & 0 & ... \\ -\alpha & ... & 0 & ... & -\alpha \\ 0 & -\alpha & ... & -\alpha & 1+4\alpha \end{bmatrix}$$

Which is positive definite, thus the spectral theorem guarantees the convergence and stability of the algorithm for all values of α. Now $u_{i,j,k-1}$ is entirely determined by the initial and boundary conditions, and it is our only know term. For the following time steps only the boundary values are determined and we need to solve the equations for the interior by an iterative method. We will use Jacobi's method. Summarized this algorithm reads

1. Make an initial guess for $u_{i,j}$ for all interior points.

2. Use Eq. 6 to compute $u_{i,j}^m$ at all interior points. Here m stands for the iteration number.
3. Stop if the convergence threshold is reached, otherwise continue to the next step.
4. Update the new value of $u_{i,j}$ for the given iteration.
5. Repeat from step 2.

The convergence threshold in our case is the difference between the new and old $u_{i,j}$. While very simple and cheap per iteration, the Jacobi iteration is slow to converge, especially for larger grids. The code written based on this algorithm can be found in section VI-IV

IV. Results

In this section we present our results for the different schemes employed. We will compare them to each other and discuss the effect of varying critical parameters.

I. Diffusion in 1D

We wrote a code based on the Monte Carlo random walk method which is included in section VI. In Fig. 3-5 we display our data obtain while tweaking certain parameters. In Figure 3 we see the effect of varying the number of walkers used, when the number of Monte Carlo cycles remains constant at 100 cycles. In Fig. 4 we keep the number of walkers constant but vary the number of Monte Carlo cycles. In both cases we see that increasing the relevant parameter smooths the function and reduce local density fluctuations. This is because we increase the number of data points to put in our histogram.

However, these density fluctuations mimics the true diffusion process in the synaptic cleft, or any other particle diffusion process for that matter. In any system of discrete particles the density will not follow the theoretical well defined curve, but the randomness of Brownian motion will give local fluctuations. Thus we can claim that the random walk method is closer to the true process than any explicit solution to the diffusion equation. After all, the way the random walkers move around is exactly the same as particles undergoing Brownian motion. In Fig. 5 we plot the solution at four different points in time. As the theoretical result predicts the concentration follows an rapidly decreasing exponential function at low t, and approaches the equilibrium solution of a straight line as t increases.

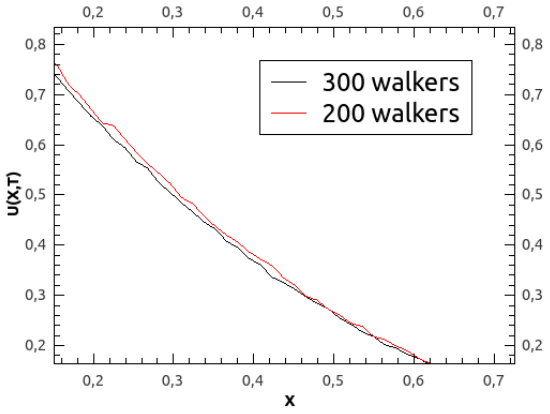

Figure 3: *A plot of the concentration as a function of x, zoomed in for detailed view. Here we compare the solutions when using 200 and 300 walkers. We observe that increasing the number of walkers smooths out the result as we expected in the discussion in section III-I.*

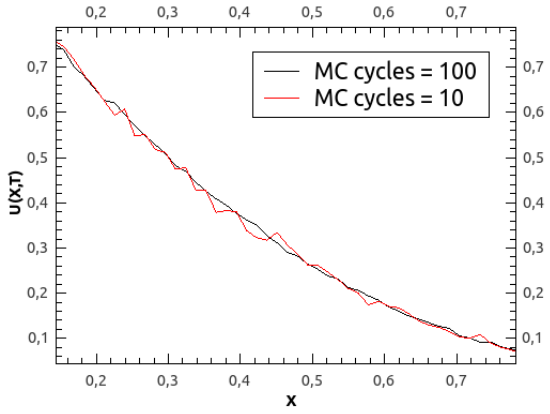

Figure 4: *Here we show again a zoomed in version of our results. This time we investigate the effect of varying the number of Monte Carlo cycles. Again we observe that by increasing the number of MC cycles, the concentration becomes a smoother function.*

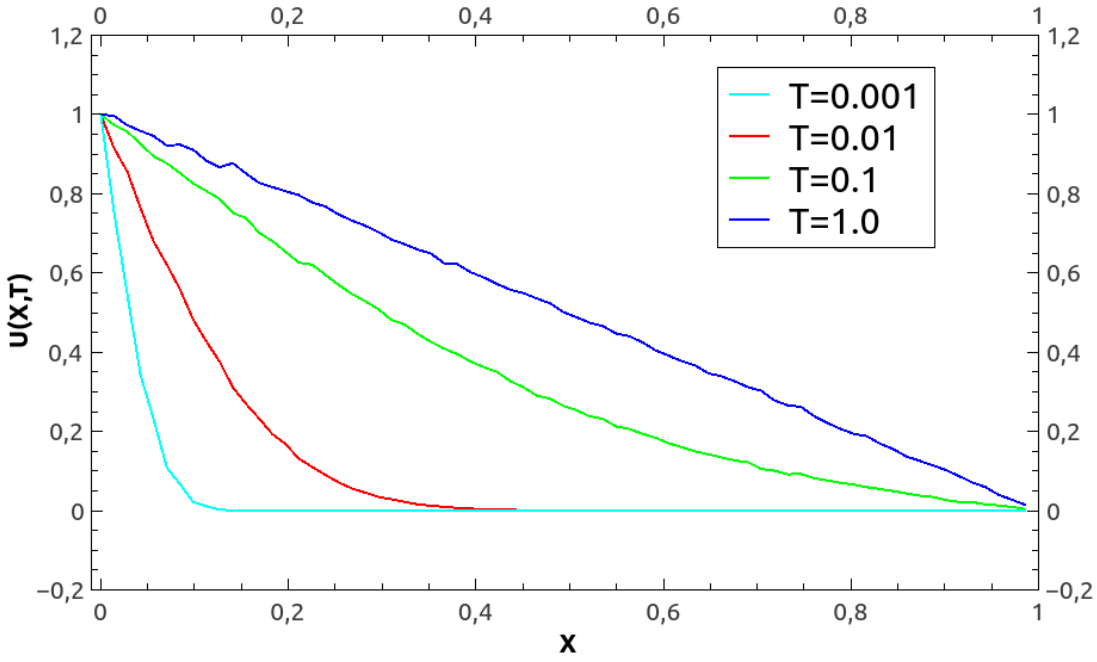

Figure 5: *Here we plot the solutions at 4 different points in time. As we expected from theory the function is strongly exponential at low t, and approaches the equilibrium solution of a straight line as t increases. The time displayed in the legend is given in units of seconds.*

Now we want to see the effect of letting the step length vary with the Gaussian distribution as the weight function. We use the mean value $\mu = 0$ and the standard deviation $\sigma = 1/\sqrt{2}$. In Figure 6 we compare the solutions with Gaussian and constant step length. This means that the step length is now given by $l = \sqrt{2Dd}t\epsilon$ where ϵ is the random number generated from the Gaussian distribution. All other parameters stay the same. We observe that the concentration decreases faster for the Gaussian step length as a function of x. The total concentration is also lower in this case. This result becomes obvious when you take into consideration the distribution of the Gaussian function which is displayed in Fig. 7. The effective step length is decreased when multiplied with ϵ, thus the particles use longer time to move across the cleft.

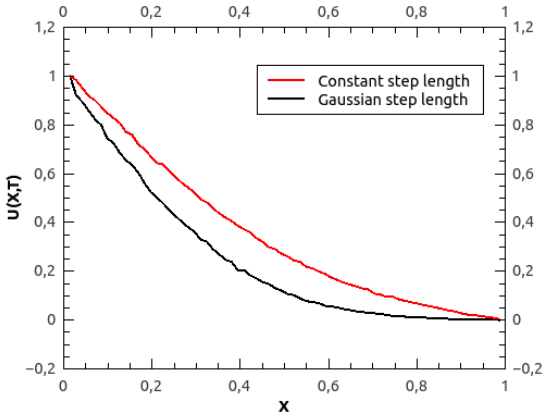

Figure 6: *Here we plot u(x,t) as a function of x for the constant step length, and the Gaussian distributed step length. We observe that the exponential function has a higher time-constant in the case of the Gaussian distribution.*

In Fig.8 we compare the solutions for the Gaussian and constant step length with the explicit solution found in project 4. We see that the plot of the Gaussian step length fits very well with the explicit solution, and the constant step length solution deviates considerably from the other two. This is again a result of mimicking real processes in nature. The diffusing particles does not move a constant length in any direction. Rather the length of the steps is distributed around a mean value, which is exactly the kind of behaviour the Gaussian distribution describe. Thus by introducing the Gaussian distributed step length, our random walk model comes even closer to the real natural phenomena of diffusion.

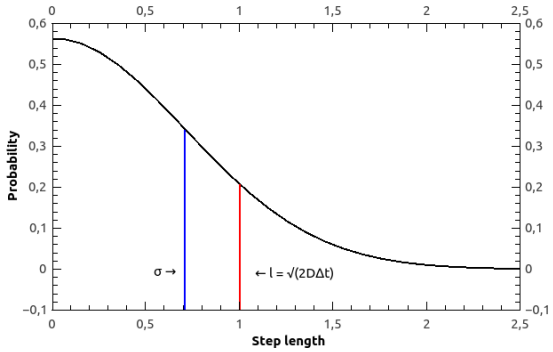

Figure 7: *The Gaussian distribution for $\mu = 0$ and $\sigma = 1/\sqrt{2}$. The constant step length is marked by the red line and σ is marked by the blue line.*

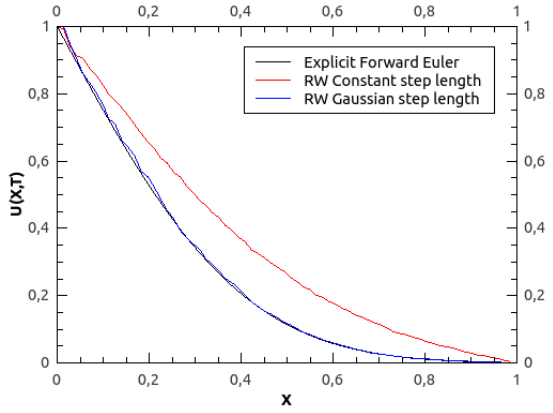

Figure 8: *A comparison between the solutions from the Monte Carlo random walk with constant and Gaussian distributed step length, and the explicit solution from project 4. We see that the curve with Gaussian step length fits better with the explicit solution than the curve with constant step length.*

II. Diffusion in 2D

Now we look at diffusion in two dimensions. As previously mention we introduced two new boundary conditions, $u(x,0,t) = u(x,1,t) = 0$. That is, we assume the particles exits the synapse at all sides of the box, except at the pre synaptic side. The source code for the programs is found in section VI. In Fig. 9 and 10 we plot our result from the explicit 2D scheme discussed in section III-II. Fig. 9 is the result for t=0.5 sec and Fig. 10 is the result for t=0.005 sec.

As in the 2D case we observe an rapid exponential decay for low t, and as t increases the rate of the exponential decay decreases. In one dimension the equilibrium solution was a straight line, however this is no longer the case for two dimensions. By introducing the two new boundary conditions at $y = 0$ and $y = 1$ walkers can exit the system at any boundary except at $x = 0$, which changes the equilibrium solution. A contour map of the explicit solution for high t is shown in Fig. 13. Here we used $t = 1$ sec, and when comparing this the solution from $t = 0.1$ sec there is was apparent change, thus we can assume this is the equilibrium solution.

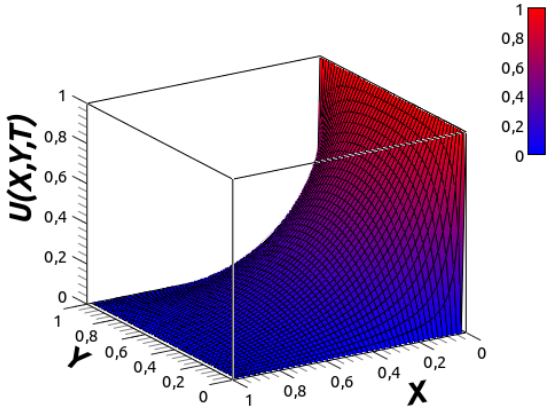

Figure 9: *Wire surface plot of the solution obtained from the explicit scheme plotted at t=0.5 sec.*

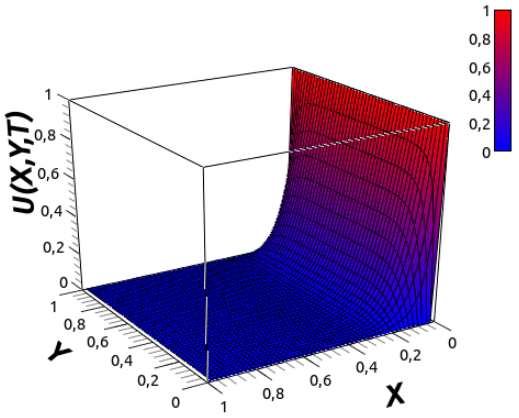

Figure 10: *Wire surface plot of the solution obtained from the explicit scheme plotted at t=0.005 sec.*

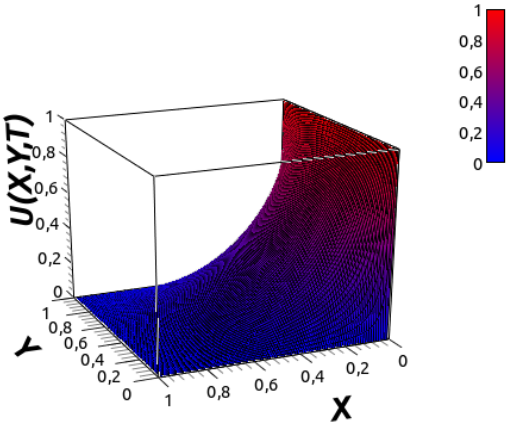

Figure 11: *Wire surface plot of the solution obtained from the implicit scheme plotted at t=0.1 sec.*

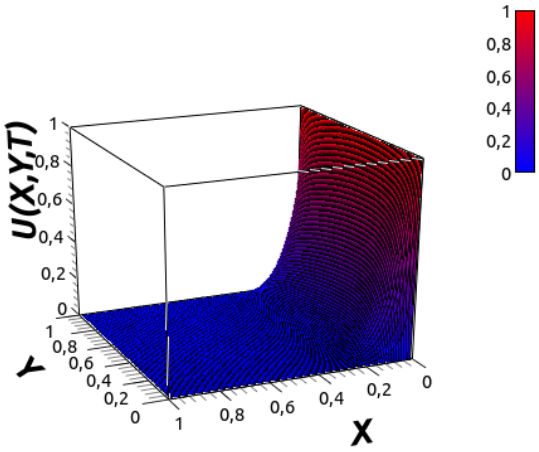

Figure 12: *Wire surface plot of the solution obtained from the implicit scheme plotted at t=0.025 sec.*

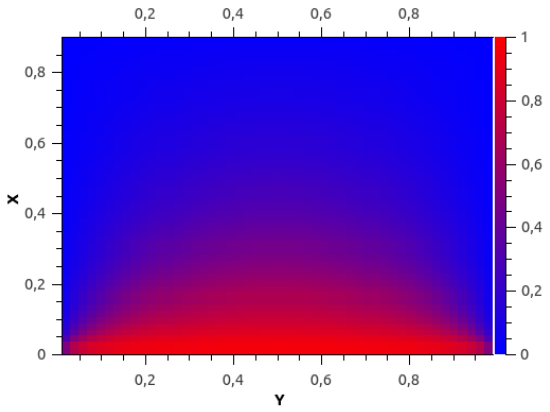

Figure 13: *Contour map of the equilibrium solution for large t. We see that even in equilibrium u(x,y,t) decreases exponentially as we move away from* $u(0, y, t)$.

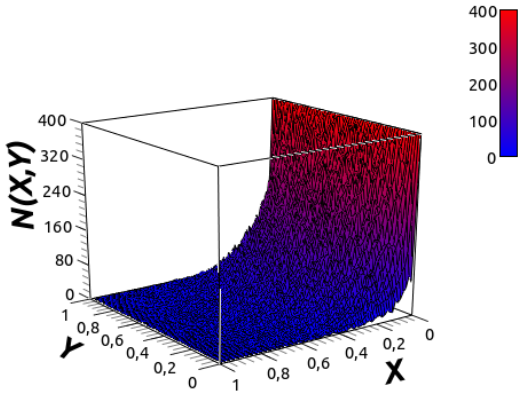

Figure 14: *Here we show N(x,y,t) where N is the total number of walkers. This plot is generated by using the 2D Monte Carlo random walk.*

When comparing the solutions from the explicit scheme (Fig. 9 and 10) to the ones from the implicit scheme (Fig. 11 and 12), the first thing that apparent is that the number of grid points are higher for the implicit scheme. This is because, as previously discussed, the explicit scheme is limited $\alpha < 1/4$ but the implicit scheme is stable and converging for all α. Thus we could use a higher number of grid points when using the implicit scheme. However both solves the problem very well. While the explicit scheme is limited in number of grid points, the implicit scheme is slow to converge. When choosing which algorithm one wants to use, these are both important points to consider. Finally in Fig. 14 we show our solution from the 2D random walk. Notice that here the z-axis shows total number of walkers, and not the concentration as previously. The shape and form of this surface fits very well with our implicit and explicit results. As in the 1D case, the local distribution of particles fluctuate so the surface is not smooth. And as previously this fluctuation is closer to the true nature of particle diffusion processes than the explicit and implicit results.

V. Conclusion

In this final project we have modelled the diffusion of neurotransmitters across the synaptic cleft. We have done this in one and two dimensions. In project 4 we used an implicit and explicit scheme to do this, but here we used the Monte Carlo random walk method. Each neurotransmitter is a "walker" which has an equal probability to move in any direction (back or forth). If we put a number of these walkers at the presynaptic side and track their movement the boundary conditions we set up will give a net movement from $x = 0$ to $x = 1$. We store their final position over a given time and reset the system. The process is repeated, and each cycle is a Monte Carlo cycle. By letting the step length vary according to the Gaussian distribution the models imitation of true diffusion increases, and the results fit well with the explicit solution.

We extended to model to two dimensions by letting the walkers move in both the x and y plane. Following this we also used an implicit and an explicit scheme to solve the diffusion equation. All of these schemes worked well, but there are pros and cons attached to each of them. The 2D random walk is closest to the true nature of the system, but is the most computationally demanding. The explicit scheme

is limited in its stability and convergence, and the implicit scheme has a slow convergence for a high number of grid points.

A final comment about our 2D solution is that the net diffusion from $x = 0$ to $x = 1$ is severely reduced by the additional boundary conditions. Most particles exits through the sides, and never reach across. This is because we modelled the system as a square grid, but in reality $L_x << L_y$ so the particles would only escape the system at the far edges and most would diffuse across the synapse. Although the 2D solution, with the boundary conditions we chose, is not very interesting in from biological point of view, the numerical recipes and algorithms we used in this project is very valuable to know.

References

[1] Morten Hjorth-Jensen, "Computational Physics Lecture Notes Fall 2012", October 12, 2012.

[2] Farnell and Gibson, Journal of Computational Physics, volume 208, pages 253-265 (2005)

[3] R. H. Thompson, "The Brain: A Neuroscience Primer", Worth Publ., (2000)

VI. Source Code

I. Random Walk in 1D

```
#include <iostream>
#include <armadillo>
#include <math.h>
#include <time.h>
#include <fstream>

using namespace std;
using namespace arma;

// Function to calculate the Gaussian distribution with mean m and stddev s.
float box_muller(float m, float s)
{
    float x1, x2, w, y1;
    static float y2;
    static int use_last = 0;

    if (use_last)
    {
        y1 = y2;
        use_last = 0;
    }
    else
    {
        do {
            x1 = 2.0 * ((double) rand() / (RAND_MAX)) - 1.0;
            x2 = 2.0 * ((double) rand() / (RAND_MAX)) - 1.0;
            w = x1 * x1 + x2 * x2;
        } while ( w >= 1.0 );

        w = sqrt( (-2.0 * log( w ) ) / w );
        y1 = x1 * w;
        y2 = x2 * w;
        use_last = 1;
    }

    return( m + y1 * s );
}

int main()
{
    int MC_cycles, N_walkers;
    double D,dt,l,pi;
    pi = 3.14159265359;
    D = 1;
    dt= 0.0001;
    l = sqrt(2*D*dt);
    MC_cycles = 500;
    N_walkers = 100;

    int NumberBins = 1/l+1;
    double Delta_x = 1./((double) NumberBins);
    const double FinalTime = 0.1;
    const int ntimesteps  = FinalTime/dt;

    vector<double> position;
```

```
    // histogram to hold the solution
    double *hist = new double[NumberBins];

    // initialize arrays
    for (int bin =0; bin < NumberBins; bin++){
      hist[bin] = histsquared[bin] = histtemp[bin] = stddev[bin] = 0;}


for (int cycle =0; cycle < MC_cycles; cycle++){


        // In the beginning of every cycle, reset to initial population

        position.resize(N_walkers);

        // Initialize walker array
        for (int walker =0; walker < N_walkers; walker++) position[walker] = 0.0;

        // loop over time
        for (int step = 0; step < ntimesteps; step++){
        double prev;
        int addwalkers = 0;

        // Loop over walkers, they will eventually produce the final histogram
        for (int walker = 0; walker < position.size(); walker++){

        prev = position[walker];  // Store previous walker position

      // Steplength determined from the Gaussian distribution

        double r = box_muller(0.0,(1/sqrt(2.0)));
        position[walker] += l*r;


     // Constant steplength movement. Comment out if using Gaussian distribution above.

       if (((double) rand() / (RAND_MAX)) > 0.5){

          position[walker] += l;
        }
        else {
          position[walker] -= l;
        }

        // Remove walkers at x=1
        if ( ( (position[walker] < Delta_x) && (prev >= Delta_x) ) || position[walker] >=1){
          position.erase(position.begin()+walker);
          walker--;
          continue;
        }

        if (position[walker] >= Delta_x && prev < Delta_x)
            addwalkers++;  // Add walkers as they move away from x=0

        if (position[walker] < 0)
              position[walker] = 0; // Walker moves into wall at x < 0

        } // end loop over walkers

        // Add 1 walker at x=0 for every walker that moved away from zero
```

```
        for (int i = 0; i < addwalkers; i++) position.push_back(0.0);

        } //end time loop


        // Set up temporary histogram
          for (int walker = 0; walker < position.size(); walker++)
          histtemp[(int) floor(position[walker]/Delta_x)]++;

        // Save to histogram
          for (int bin =0; bin < NumberBins; bin++) {
          hist[bin] += histtemp[bin];
          histtemp[bin] = 0;
            }


        } // end loop over Monte Carlo cycles

        // Normalize with number of Monte Carlo cycles
        double NormFactor = 1.0/(MC_cycles);
        for (int bin =0; bin < NumberBins; bin++) {
          hist[bin] *=   NormFactor;
        }

// Write out data
ofstream hist_out;
hist_out.open("histogram");
for(int bin=0; bin < NumberBins;bin++)
{
    hist_out        << bin*Delta_x <<"  "<< hist[bin] << endl;
}
hist_out.close();


}
```

II. Random Walk in 2D

```
#include <iostream>
#include <armadillo>
#include <math.h>
#include <time.h>
#include <fstream>

using namespace std;
using namespace arma;

// Function to calculate the Gaussian distribution with mean m and stddev s.
float box_muller(float m, float s)
{
    float x1, x2, w, y1;
    static float y2;
    static int use_last = 0;

    if (use_last)
    {
        y1 = y2;
        use_last = 0;
    }
    else
```

```
    {
        do {
            x1 = 2.0 * ((double) rand() / (RAND_MAX)) - 1.0;
            x2 = 2.0 * ((double) rand() / (RAND_MAX)) - 1.0;
            w = x1 * x1 + x2 * x2;
        } while ( w >= 1.0 );

        w = sqrt( (-2.0 * log( w ) ) / w );
        y1 = x1 * w;
        y2 = x2 * w;
        use_last = 1;
    }

    return( m + y1 * s );
}

int main()
{
    int MC_cycles, N_walkers;
    double D,dt,l,pi;
    pi = 3.14159265359;
    D = 1;
    dt= 0.0001;
    l = sqrt(2*D*dt);
    MC_cycles = 100;
    N_walkers = 100;

    int NumberBins = 1/l+1;
    double Delta_x = 1./((double) NumberBins);
    double Delta_y = 1./((double) NumberBins);
    const double FinalTime = 0.1;
    const int ntimesteps  = FinalTime/dt;

    vector<double> position_x;
    vector<double> position_y;
    vector<double> y; // to keep track of the y coordinate of walkers moving away from x=0

    // histogram to store data
    mat hist(NumberBins,NumberBins);
    mat hist_temp(NumberBins,NumberBins);



for (int cycle =0; cycle < MC_cycles; cycle++){


        // In the beginning of every cycle, reset to initial population

        position_x.resize(N_walkers);
        position_y.resize(N_walkers);
        y.resize(N_walkers);

        // initialize walker array
        for (int walker =0; walker < N_walkers; walker++)
           {
                position_y[walker] = walker*Delta_y;
                position_x[walker] = 0.0;
                y[walker] = 0.0;
           }

        // loop over time
```

```
      for (int step = 0; step < ntimesteps; step++){
      double prev_x,prev_y;
      int addwalkers = 0;

      // Loop over walkers, they will eventually produce the final histogram
      for (int walker = 0; walker < position_x.size(); walker++){

      prev_x = position_x[walker];  // previous walker position_x
      prev_y = position_y[walker];  // previous walker position_y

      // Gaussian distributed step length, or constant if gauss = 1.
       double gauss = box_muller(0.0,(1/sqrt(2.0)));
      // double gauss = 1

  // Equal probability of moving in either direction.
      if (position_x[walker] < Delta_x)

      {
          position_x[walker] +=1*gauss;
      }

      else
      {
     double r = ((double) rand() / (RAND_MAX));

     if ( r > 0.75)       { position_x[walker] += 1*gauss;}

     else if ( r > 0.50) { position_x[walker] -= 1*gauss;}

     else if ( r > 0.25) { position_y[walker] += 1*gauss;}

      else {position_y[walker] -= 1*gauss;}
      }

         // if walker goes outside x=1 or move back into x=0 after having moved away it is removed
       if ( ( (position_x[walker] < Delta_x) && (prev_x >= Delta_x) ) || position_x[walker] >=1){
        position_x.erase(position_x.begin()+walker);
        position_y.erase(position_y.begin()+walker);
        walker--;
        continue;
      }

      if ( (position_y[walker] >=1) ){        // if walker goes outside y=1 it is removed
        position_x.erase(position_x.begin()+walker);
        position_y.erase(position_y.begin()+walker);
        walker--;
        continue;
      }

      if ( (position_y[walker] <=0) ){        // if walker goes outside y = 0 it is removed
        position_x.erase(position_x.begin()+walker);
        position_y.erase(position_y.begin()+walker);
        walker--;
        continue;
      }

      if (position_x[walker] >= Delta_x && prev_x < Delta_x)
      {
          y[addwalkers] = position_y[walker]; // y coordinate of added walker
          addwalkers++;                       // add walkers when one move away from x=0
      }
```

```
        if (position_x[walker] < 0)
              position_x[walker] = 0; //  moves into wall at x < 0



        } // end loop over walkers

        // add 1 walker at x=0 for every walker that moved away from zero
        // y coordinate stored from before
        for (int i = 0; i < addwalkers; i++){
            position_x.push_back(0.0);
            position_y.push_back( y[i] );
        }


        } //end time loop


        // set up temporary histogram
          for (int walker = 0; walker < position_x.size(); walker++)
          {
              hist_temp((int)floor(position_x[walker]/Delta_x),(int)floor(position_y[walker]/(Delta_y)))++;
          }

          for (int bin =0; bin < NumberBins; bin++) {
              for (int bin2=0;bin2< NumberBins;bin2++)
              {
                hist(bin,bin2) += hist_temp(bin,bin2);
                hist_temp(bin,bin2) = 0;
            }}


        } // end loop over Monte Carlo cycles

       // Normalize with number of Monte Carlo cycles
        double NormFactor = 1.0/(MC_cycles);
        for (int bin =0; bin < NumberBins; bin++) {
            for (int bin2=0; bin2 < NumberBins; bin2++)
            {
              hist(bin,bin2) *=   NormFactor;
            }

        }

// write out data in matrix
ofstream hist_out;
hist_out.open("histogram");
hist_out << hist<< endl;
hist_out.close();

}
```

III. Explicit scheme in 2D

```
#include <iostream>
#include <armadillo>
#include <math.h>
#include <time.h>
#include <fstream>
```

```
using namespace arma;
using namespace std;

int main()
{
    double D = 1;
    int n = 50;
    ofstream data1,data2,data3;
    double dt = 5e-5;
    int tsteps = 100000;
    double xsteps = 1/double(n);
    double a = dt/(D*xsteps*xsteps);

     mat U = zeros<mat>(n,n);

     // Initial conditions
     for (int j=0 ; j < n ; j++){
         U(0,j)=1;
     }

     for (int t=1 ; t < tsteps ; t++){

         for (int j=0 ; j < n ; j++){
         for (int i=0 ; i < n ; i++){
             if(i==0)          { U(i,j) = 1; } // Boundary conditions
             else if(j==0)     { U(i,j) = 0; }
             else if(i==(n-1)) { U(i,j) = 0; }
             else if(j==(n-1)) { U(i,j) = 0; }
             else              { U(i,j) = U(i,j)+a*( U(i+1,j) + U(i-1,j)
                                       + U(i,j+1) + U(i,j-1) - 4*U(i,j) ); }
         }}

     }

        data1.open("data1");
        data1 << U << endl;
        data1.close();

}
```

IV. Implicit scheme in 2D

```
#include <iostream>
#include <armadillo>
#include <math.h>
#include <time.h>
#include <fstream>

using namespace std;
using namespace arma;



int main()
{
    double D = 1;
    int n = 100;
    ofstream data1;
    double dt = 5e-5;
```

```
    int tsteps = 500;
    double xsteps = 1/double(n);
    double a = dt/(D*xsteps*xsteps);

    // Initialize temporary and solution matrix
     mat U = zeros<mat>(n,n);
     mat U_temp = zeros<mat>(n,n);

     // Initial conditions guess and inital value.
     for (int i=0 ; i < n ; i++){
         for(int j=0 ; j < n ; j++){
         U_temp(i,j)=0.5;
         U(0,j)=1;}
     }


 // Start time iteration
     for (int t=0; t < tsteps ; t++){

         int max_iter = 1000;
         int iterations = 0;
         double diff=1;

// Compute inner coordinates by Jacobis algorithm
     while((iterations<=max_iter) && (diff>0.1)){
         diff = 0.0;
         U_temp = U;

         for (int j=1 ; j < n ; j++){
         for (int i=1 ; i < n ; i++){
             if(i==0)          { U(i,j) = 1; } // Boundary conditions
             else if(j==0)     { U(i,j) = 0; }
             else if(i==(n-1)) { U(i,j) = 0; }
             else if(j==(n-1)) { U(i,j) = 0; }
             else              { U(i,j) = (1/(1+4*a))*(a*( U_temp(i+1,j) + U_temp(i-1,j)
                                    + U_temp(i,j+1) + U_temp(i,j-1))+U(i,j)); }

             diff +=fabs(U_temp(i,j)-U(i,j)); // Calculate difference

         }}
        iterations++;
        diff/=pow((double)n,2.0);
     }

}
        data1.open("data1");
        data1 << U << endl;
        data1.close();

}
```