FYS4150 - Computational Physics - Project 5

Eimund Smestad - Candidate number: 68
Email: eimundsm@fys.uio.no
8th December, 2014

Abstract

This project is a continuation of the study of diffusion of neurotransmitters across a synaptic cleft in project 4, but where I have extended to diffusion in 2 dimensions. An analytical solution are derived for the partial differential equation for 2D diffusion problem with wall boundaries. Different numerical methods for solving the 2D diffusion problems are studied to verify the methods and the analytical solution. These methods are explicit scheme, Jacobi iterative method, Monte Carlo and Metropolis algorithm. The different numerical methods are compared in computation time, stability and accuracy.

1 Diffusion of neurotransmitters

I will study diffusion as a transport process for neurotransmitters across synaptic cleft separating the cell membrane of two neurons, for more detail see [3]. The diffusion equation is the partial differential equation

$$\frac{\partial u(\mathbf{x},t)}{\partial t} = \nabla \cdot \left(D(\mathbf{x},t)\, \boldsymbol{\nabla} u(\mathbf{x},t) \right),$$

where u is the concentration of particular neurotransmitters at location $\mathbf{x}$ and time t with the diffusion coefficient D. In this study I consider the diffusion coefficient as constant, which simplify the diffusion equation to the heat equation

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

I will look at the concentration of neurotransmitter u in two dimensions with x_1 parallel with the direction between the presynaptic to the postsynaptic across the synaptic cleft, and x_2 is parallel with both presynaptic to the postsynaptic. Hence we have the differential equation

$$\frac{\partial u\left(\{x_i\}_{i=1}^2, t\right)}{\partial t} = D\sum_{j=1}^{2} \frac{\partial^2 u\left(\{x_i\}_{i=1}^2, t\right)}{\partial {x_j}^2}, \tag{1}$$

where $\{x_i\}_{i=1}^2 = (x_1, x_2) = \mathbf{x}$. The boundary and initial condition that I'm going to study is

$$\begin{aligned}
&\exists\{d,w\} \subseteq \mathbb{R}_{0+} \exists \{w_i\}_{i=1}^2 \subseteq \mathbb{R}_{0+}^{w-} \Big(\forall t \in \mathbb{R}_0 : \forall x_2 \in \mathbb{R}_{w_1}^{w_2} : u(0,x_2,t) = u_0 \\
&\quad \wedge \forall t \in \mathbb{R} \Big(\forall x_2 \in \mathbb{R}_{0+}^{w-} : u(d,x_2,t) = 0 \wedge \forall x_1 \in \mathbb{R}_0^d : \Big(u(x_1,0,t) = 0 \wedge u(x_1,w,t) = 0 \Big) \Big) \\
&\quad \wedge \forall x_1 \in \mathbb{R}_{0+}^{d-} \forall x_2 \in \mathbb{R}_{0+}^{w-} : u\left(\{x_i\}_{i=1}^2, 0\right) = 0 \wedge \forall x_2 \in \mathbb{R}_0^w \setminus \mathbb{R}_{w_1}^{w_2} : u(0,x_2,0) = 0 \Big)
\end{aligned} \tag{2}$$

where d is the distance between the presynaptic and the postsynaptic, and w is the width of the presynaptic and postsynaptic. Note that the notation $\forall x \in \mathbb{R}_{a+}^{b-} \Leftrightarrow a < x < b$, where as

$\forall x \in \mathbb{R}_a^b \Leftrightarrow a \leq x \leq b$. Note also that these boundary conditions implies that the neurotransmitters are transmitted from presynaptic at $x_1 = 0$ and $w_1 \leq x_2 \leq w_2$ with constant concentration u_0; the neurotransmitters are immediately absorbed at the postsynaptic $x_1 = d$; there are no neurotransmitters at boundary width $x_2 = 0$ and $x_2 = w$ of the synaptic cleft; and we have the initial condition at $t = 0$ where there are no neurotransmitters between the pre- and postsynaptic as well on the side of the synaptic vesicles $x_1 = 0, 0 \leq x_2 < w_1$ and $w_2 < x_2 \leq w$.

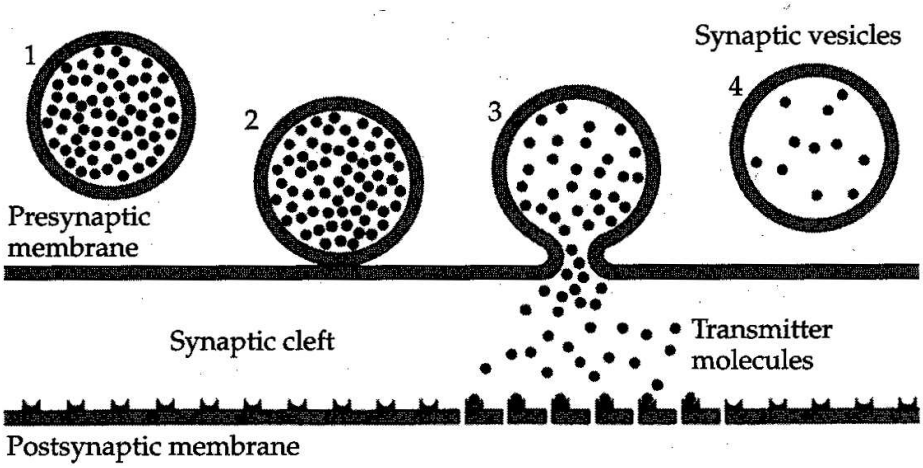

Figure 1.1: *Left: Schematic drawing of the process of vesicle release from the axon terminal and release of transmitter molecules into the synaptic cleft. (From Thompson: "The Brain", Worth Publ., 2000). Right: Molecular structure of the two important neurotransmitters glutamate and GABA.*

To solve the differential equation (1) with the boundary and initial condition (2) we make an ansatz that the solution is unique, which is the case for a deterministic system. We recognize the heat equation as part of the class of partial differential equations spanned by the Poisson's equation for each time instance. The Uniqueness theorem for the Poisson's equation $\nabla^2 u = f$ [9] says that the Poisson's equation has a unique solution with the Dirichlet boundary condition, where Dirichlet boundary condition is here defined as a boundary that specifies the values the solution must have at the boundary. Not to be confused with Dirichlet boundary condition with zero at the boundary, which is just a special case and I will refer to it as 0th Dirichlet boundary condition.

Unfortunately the boundary condition in (2) is not a Dirichlet boundary, since the boundary is not specified on the side of the synaptic vesicles $x_1 = 0, 0 \leq x_2 < w_1$ and $w_2 < x_2 \leq w$, and closer investigation will show that the boundary condition in (2) does not provide a unique solution. So we need to add further condition to make the solution unique, and I make an assumption that the total concentration u in an infinitesimal area has uniform concentration per length u_1 and u_2 in each direction x_1 and x_2 accordingly. Hence $u = u_1 u_2$ at every point and therefore we can write

$$\forall x_1 \in \mathbb{R}_0^d \forall x_2 \in \mathbb{R}_0^w \forall t \in \mathbb{R}_0 : u\left(\{x_i\}_{i=1}^2, t\right) = u_1(x_1, t)\, u_2(x_2, t) \ . \tag{3}$$

Putting this into the heat equation (1) we get

$$u_2(x_2, t)\frac{\partial u_1(x_1, t)}{\partial t} + u_1(x_1, t)\frac{\partial u_2(x_2, t)}{\partial t} = D\left(u_2(x_2, t)\frac{\partial^2 u_1(x_1, t)}{\partial {x_1}^2} + u_1(x_1, t)\frac{\partial^2 u_2(x_2, t)}{\partial {x_2}^2}\right),$$

which can be written as two heat equations

$$\forall i \in \mathbb{N}_1^2 : \frac{\partial u_i(x_i,t)}{\partial t} = D\frac{\partial^2 u_i(x_i,t)}{\partial {x_i}^2}\,. \tag{4}$$

I make another ansatz that the the boundary at $u(0,x_2,t)$ determined by $u_2(x_2,t)$ alone, and by satisfying initial condition $u(0,x_2,0)$ for u_2

$$\forall t \in \mathbb{R}_0 \left(: u_2(0,t) = u_2(w,t) = 0 \wedge \forall x_2 \in \mathbb{R}_{w_1}^{w_2} : u_2(x_2,t) = u_0\right) \wedge \forall x_2 \in \mathbb{R}_0^{w} \setminus \mathbb{R}_{w_1}^{w_2} : u_2(x_2,0) = 0 \tag{5}$$

we have now determined the values on all the boundaries have therefore Dirichlet boundary condition, and therefore a unique solution of u. We found the analytical solution to the heat equation in (4) for $i = 2$ with similar boundary and initial condition in project 4 [3], and I will therefore not show the derivation here, but just state the solutions

$$u_2(x_2,t) = u_0 \begin{cases} \frac{x_2}{w_1} - \sum_{n=1} \frac{2}{n\pi} \sin\left(n\pi\left(1-\frac{x_2}{w_1}\right)\right) \exp\left(-D\left(\frac{n\pi}{w_1}\right)^2 t\right) & : x_2 \in \mathbb{R}_0^{w_1-} \\ 1 & : x_2 \in \mathbb{R}_{w_1}^{w_2} \\ \frac{w-x_2}{w-w_2} - \sum_{n=1} \frac{2}{n\pi} \sin\left(n\pi\frac{x_2-w_2}{w-w_2}\right) \exp\left(-D\left(\frac{n\pi}{w-w_2}\right)^2 t\right) & : x_2 \in \mathbb{R}_{w_2+}^{w}\,. \end{cases} \tag{6}$$

Since I have established that the boundary at $u(0,x_2,t)$ determined by $u_2(x_2,t)$ alone, means that u_1 has the following boundary and initial condition

$$\forall t \in \mathbb{R}_0 \left(: u_1(0,t) = 1 \wedge u_1(d,t) = 0\right) \wedge \forall x_1 \in \mathbb{R}_{0+}^{d-} : u_1(x_1,0) = 0\,. \tag{7}$$

And using the analytical solution from project 4 to heat equation (4) for $i = 1$ with boundary and initial condition in (7), we have

$$u_1(x_1,t) = 1 - \frac{x_1}{d} - \sum_{n=1} \frac{2}{n\pi} \sin\left(n\pi\frac{x_1}{d}\right) \exp\left(-D\left(\frac{n\pi}{d}\right)^2 t\right). \tag{8}$$

To summarize the solution to the concentration u is given by (3) with (6) and (8).

2 Numerical methods

2.1 The θ-rule

The Taylor expansion is given by

$$u(x) = \sum_{n=0} \frac{\overset{(n)}{u}(x_0)}{n!}(x-x_0)^n \tag{9}$$

where $\overset{(n)}{u} = \frac{\mathrm{d}^n u}{\mathrm{d}t^n}$ and x_0 is a initial value where we step from to x . If we now use the first order approximation

$$u(x) \approx u(x_0) + \overset{(1)}{u}(x_0)(x - x_0) \ .$$

The first order differential equation $\overset{(1)}{u}(x) = f(x)$ is determined when we have the initial condition $u(x_0)$, however $\overset{(1)}{u}(x_0)$ is not an initial condition, and it depends on how we calculate it numerically from the initial condition. Now note that $\overset{(1)}{u}(x_0)$ is the same for different values of x in the approximation above and lets say that we calculate it as given from the approximation above;

$$\overset{(1)}{u}(x_0) \approx \frac{u(x) - u(x_0)}{x - x_0} \ . \tag{10}$$

So now use this in another point $x_\theta = \theta x + (1 - \theta) x_0$ which we also approximate to the first order, and if we use the expression above for $\overset{(1)}{u}(x_0)$ we get

$$\begin{aligned} u(x_\theta) &\approx u(x_0) + \overset{(1)}{u}(x_0)(x_\theta - x_0) = u(x_0) + \theta \overset{(1)}{u}(x_0)(x - x_0) \\ &\approx u(x_0) + \frac{u(x) - u(x_0)}{x - x_0}\theta(x - x_0) = \theta u(x) + (1 - \theta) u(x_0) \ , \end{aligned} \tag{11}$$

this is known as the θ-rule. The θ-rule can be used to approximate the solution of the following first order differential equation

$$\overset{(1)}{u}(x) = f(u(x)) \ , \tag{12}$$

where we use (10) to approximate the expression $\overset{(1)}{u}(x)$ and given an even better or worse approximation to the solution $u(x)$ by approximating $f(x) \approx f(x_\theta)$;

$$\frac{u(x) - u(x_0)}{x - x_0} \approx f\Big(u(x_\theta)\Big) = f\Big(\theta u(x) + (1 - \theta) u(x_0)\Big) \ ,$$

which discretize to

$$\frac{u_{i+1} - u_i}{x_{i+1} - x_i} = f\Big(\theta u_{i+1} + (1 - \theta) u_i\Big) \qquad \text{where } i \in \mathbb{N}_0 \text{ and } u_0 \text{ is an initial contidion.} \tag{13}$$

We can find the the next step in the numerical solution to (12) by solving this difference equation with regard to u_{i+1}. Note the above discretization is known as Forward Euler scheme (Explicit) when $\theta = 0$, Backward Euler scheme (Implicit) when $\theta = 1$ and Crank-Nicolson scheme when $\theta = \frac{1}{2}$.

2.2 Second order derivative

We approximated the first order derivative in (10), but we need to approximate the second order derivative to be able to solve the diffusion in (1). I start by expanding the Taylor series in (9) around the point $x_0 \pm \Delta x$ accordingly;

$$u(x_0 \pm \Delta x) = \sum_{n=0} \frac{\overset{(n)}{u}(x_0)}{n!} (\pm \Delta x)^n \ . \tag{14}$$

Now adding these two expansions

$$u(x_0+\Delta x)+u(x_0-\Delta x)=2\sum_{n=0}\frac{\overset{(2n)}{u}(x_0)}{(2!)}\Delta x^{2n}=2u(x_0)+\overset{(2)}{u}(x_0)\Delta x^2+2\sum_{n=2}\frac{\overset{(2n)}{u}(x_0)}{(2!)}\Delta x^{2n},$$

and solve it with

$$\begin{aligned}\overset{(2)}{u}(x_0)&=\frac{u(x_0+\Delta x)-2u(x_0)+u(x_0-\Delta x)}{\Delta x^2}-2\sum_{n=2}\frac{\overset{(2n)}{u}(x_0)}{(2!)}\Delta x^{2(n-1)}\\&=\frac{u(x_0+\Delta x)-2u(x_0)+u(x_0-\Delta x)}{\Delta x^2}+O\left(\Delta x^2\right),\end{aligned}$$

So the second order derivative can be approximated with

$$\overset{(2)}{u}(x_0)\approx\frac{u(x_0+\Delta x)-2u(x_0)+u(x_0-\Delta x)}{\Delta x^2}\tag{15}$$

with the local truncation error $O\left(\Delta x^2\right)$.

2.3 The heat equation

We want to discretize the dimensionless heat equation from (1), where we use $D=1$, $u_0=1$ and $d=1$,

$$\frac{\partial u\left(\{x_i\}_{i=1}^2,t\right)}{\partial t}=\sum_{\ell=1}^{2}\frac{\partial^2 u\left(\{x_i\}_{i=1}^2,t\right)}{\partial {x_\ell}^2},$$

to numerically solve diffusion of neurotransmitters. First we do the θ-rule discretization in (13)

$$\begin{aligned}\frac{u_{(i+1)\{j_k\}_{k=1}^2}-u_{i\{j_k\}_{k=1}^2}}{\Delta t}&=\sum_{\ell=1}^{2}\frac{\partial^2 u_{(i+\theta)\{j_k\}_{k=1}^2}}{\partial {x_\ell}^2}=\sum_{\ell=1}^{2}\frac{\partial^2\left(\theta u_{(i+1)\{j_k\}_{k=1}^2}+(1-\theta)u_{i\{j_k\}_{k=1}^2}\right)}{\partial {x_\ell}^2}\\&=\sum_{\ell=1}^{2}\left(\theta\frac{\partial u_{(i+1)\{j_k\}_{k=1}^2}}{\partial {x_\ell}^2}+(1-\theta)\frac{\partial u_{i\{j_k\}_{k=1}^2}}{\partial {x_\ell}^2}\right)\end{aligned}$$

where index i is stepping of t and j_k are stepping of x_k. Note also that the following notation expand accordingly $u_{i\{j_k\}_{k=1}^2}=u_{ij_1j_2}$, which becomes a more elegant notation for larger n in $u_{i\{j_k\}_{k=1}^n}=u_{ij_1j_2\ldots j_n}$. Now we implement the discretization of the second order in (15)

$$\begin{aligned}\frac{u_{(i+1)\{j_k\}_{k=1}^2}-u_{i\{j_k\}_{k=1}^2}}{\Delta t}=\sum_{\ell=1}^{2}\Bigg(&\frac{\theta}{\Delta {x_\ell}^2}\left(u_{(i+1)\{j_k+\delta_{k\ell}\}_{k=1}^2}-2u_{(i+1)\{j_k\}_{k=1}^2}+u_{(i+1)\{j_k-\delta_{k\ell}\}_{k=1}^2}\right)\\&+\frac{1-\theta}{\Delta {x_\ell}^2}\left(u_{i\{j_k+\delta_{k\ell}\}_{k=1}^2}-2u_{i\{j_k\}_{k=1}^2}+u_{i\{j_k-\delta_{k\ell}\}_{k=1}^2}\right)\Bigg),\end{aligned}\tag{16}$$

where $\delta_{k\ell}$ is the Kronecker delta, and we now clearly see the elegance of the notation $u_{i\{j_k\}_{k=1}^n}$.

The dimensionless initial condition from (4) gives us

$$u_{0\{j_k\}_{k=1}^2} = \begin{cases} 1 & : j_1 = 0 \text{ and } x_{2j_2} \in \mathbb{R}_{w_1}^{w_2} \\ 0 & : \text{elsewhere,} \end{cases}$$

where $x_{2j_2} = x_{20} + \frac{j_2}{\Delta x_2}$. For the explicit scheme $\theta = 0$ we get

$$u_{(i+1)\{j_k\}_{k=1}^2} = u_{i\{j_k\}_{k=1}^2} + \sum_{\ell=1}^{2} \alpha_\ell \left(u_{i\{j_k+\delta_{k\ell}\}_{k=1}^2} - 2u_{i\{j_k\}_{k=1}^2} + u_{i\{j_k-\delta_{k\ell}\}_{k=1}^2} \right) \tag{17}$$

where

$$\alpha_\ell = \frac{\Delta t}{\Delta {x_\ell}^2} \qquad \text{and} \qquad n_\ell = \frac{1}{\Delta x_\ell}\,,$$

where n_ℓ is number of grid points in ℓ direction.

2.3.1 Stability and convergence

According to Lax equivalence theorem a consistent finite difference method for well-posed (a solution exists, unique and continuous) linear initial value problem, the method is convergent if and only if stable. Which means that we only can show either stability and convergence to show both.

There is a theorem that states that a matrix $\mathbf{A}$ converges $\lim_{i\to\infty} \mathbf{A}^i = 0$ if and only if the spectral radius $\rho(\mathbf{A}) < 1$. The spectral radius is defined as

$$\rho(\mathbf{A}) = \max_m \left| \lambda_m \right|\,,$$

where λ_m are the eigenvalues of $\mathbf{A}$. The explicit scheme in (17) can be written as $\mathbf{u}_{i+1} = \mathbf{A}\mathbf{u}_i = \mathbf{A}^{i+1}\mathbf{u}_0$, and the this matrix can be written as

$$\mathbf{A} = \mathbf{I} - \mathbf{B}$$

where $\mathbf{I}$ is the identity matrix. This implies that the eigenvalues μ_m of $\mathbf{B}$ is related to the eigenvalues λ_m of $\mathbf{A}$ as follows

$$\lambda_m = 1 - \mu_m\,. \tag{18}$$

The eigenvalues μ_m can be found from the eigenequation (17)

$$\sum_{\ell=1}^{2} \alpha_\ell \left(-u_{i\{j_k+\delta_{k\ell}\}_{k=1}^2} + 2u_{i\{j_k\}_{k=1}^2} - u_{i\{j_k-\delta_{k\ell}\}_{k=1}^2} \right) = \mu_m u_{i\{j_k\}_{k=1}^2}\,. \tag{19}$$

Using the expression for$\alpha_\ell = \frac{\Delta t}{\Delta {x_\ell}^2}$ we have the discretization of the following equation

$$u\left(\{x_i\}_{i=1}^{2},t\right)=-{\omega_m}^2\sum_{\ell=1}^{2}\frac{\partial^2 u\left(\{x_i\}_{i=1}^{2},t\right)}{\partial {x_\ell}^2},$$

which has the solution

$$u\left(\{x_i\}_{i=1}^{2},t\right)=\sum_{\ell=1}^{2}A_\ell\left(t\right)\prod_{\ell=1}^{2}\sin\left(\omega_m x_\ell+\varphi_\ell\right)$$

where

$$\omega_m=\sqrt{\frac{\Delta t}{\mu_m}}.$$

Now using this solution into the eigenequation (19) we get

$$\begin{aligned}&\mu_m\sum_{\ell=1}^{2}\sin\left(\omega_m x_{\ell j_\ell}+\varphi_\ell\right)\\&=\sum_{\ell=1}^{2}\alpha_\ell\left(-\sin\left(\omega_m\left(x_{\ell j_\ell}+\Delta x_\ell\right)+\varphi_\ell\right)+2\sin\left(\omega_m x_{\ell j_\ell}+\varphi_\ell\right)-\sin\left(\omega_m\left(x_{\ell j_\ell}-\Delta x_\ell\right)+\varphi_\ell\right)\right)\end{aligned}$$

which can be simplified with the trigonometrical relation $\sin\left(u+v\right)=\sin u\cos v+\cos u\sin v$

$$\mu_m\sum_{\ell=1}^{2}\sin\left(\omega_m x_\ell+\varphi_\ell\right)=2\sum_{\ell=1}^{2}\alpha_\ell\left(1-\cos\left(\omega_m\Delta x_\ell\right)\right)\sin\left(\omega_m x_\ell+\varphi_\ell\right),\tag{20}$$

note that $\mu_m\neq 0$ because of $\omega_m\propto\frac{1}{\sqrt{\mu_m}}$. For this equation to be an eigenequtions we need to satisfy

$$\alpha_1\left(1-\cos\left(\omega_m\Delta x_1\right)\right)=\alpha_2\left(1-\cos\left(\omega_m\Delta x_2\right)\right).$$

If we have square lattice $\alpha_1=\alpha_2$ the above relation is automatically satisfied, if not we need to solve the equation with regard to ω_m, and from that find the eigenvalue μ_m from the relation $\omega_m=\sqrt{\frac{\Delta t}{\mu_m}}$. Since the approach with $\alpha_1\neq\alpha_2$ is difficult to solve analytically I will show the further steps with square lattice $\alpha=\alpha_1=\alpha_2$, and in the case of $\alpha_1\neq\alpha_2$ one can approximate to a square lattice problem with the smallest step size of Δx_ℓ. If you forfill the convergence criteria for the a square lattice with the smallest step size, then you are guaranteed to satisfy the convergence criteria in the non-square lattice case as well.

For the square lattice with α we see from (20) that the eigenvalue μ_m must be within

$$0\leq\mu_m\leq 2\alpha.$$

Inserting these eigenvalues into (18) and using the convergence condition $\rho\left(\mathbf{A}\right)<1$ yields the following inequality

$$-1 < 1 - 2\alpha < 1$$

which result in the constraint for convergences

$$\alpha < \frac{1}{2},$$

where the α is always positive. So when we want to solve the heat equation explicitly we satisfy the convergence criteria if we chose a time step accordingly

$$\Delta t < \min_{\ell} \frac{1}{{n_\ell}^2}. \tag{21}$$

2.4 Jacobi's iterative method

When we have $\theta \neq 0$ in (16) we have implicit schemes, and I rewrite (16) with unknowns on the left side and knowns on the right side;

$$\begin{aligned}\forall k \in \mathbb{N}_1^2 \forall j_k \in \mathbb{N}_1^{n_k-2} : &\left(1 + 2\theta \sum_{\ell=1}^{2} \alpha_\ell\right) u_{(i+1)\{j_k\}_{k=1}^2} - \theta \sum_{\ell=1}^{2} \alpha_\ell \left(u_{(i+1)\{j_k+\delta_{k\ell}\}_{k=1}^2} + u_{(i+1)\{j_k-\delta_{k\ell}\}_{k=1}^2}\right) \\ &= \left(1 - 2(1-\theta) \sum_{\ell=1}^{2} \alpha_\ell\right) u_{i\{j_k\}_{k=1}^2} + (1-\theta) \sum_{\ell=1}^{2} \alpha_\ell \left(u_{i\{j_k+\delta_{k\ell}\}_{k=1}^2} + u_{i\{j_k-\delta_{k\ell}\}_{k=1}^2}\right),\end{aligned}$$

where the boundary are given at the indices $j_k = 0$ and $j_k = n_k - 1$ where n_k are the number of points i k direction. Now I will transform this into a linear algebra problem $\mathbf{A}_{i+1}\mathbf{v}_{i+1} = \mathbf{b}_i$, where elements of $\mathbf{v}_{i+1}$ are

$$v_{(i+1)j} = v_{(i+1)(j_1 n_2 + j_2)} = u_{(i+1)\{j_k\}_{k=1}^2}. \tag{22}$$

The fixed boundary values with $u_0 = 1$ at the presynaptic $x_1 = 0$ and $w_1 \leq x_2 \leq w_2$ are given by the indices j in the set

$$\mathbb{B}_1 = \mathbb{N}_{m_1}^{m_2}.$$

We have non-fixed values at the boundary on the side of the synaptic vesicles at the presynaptic $x_1 = 0$, $0 \leq x_2 < w_1$ and $w_2 < x_2 < w$, and are given by the indices j in the set

$$\mathbb{B}_2 = j \in \mathbb{N}_0^{n_2-1} \setminus \mathbb{B}_1.$$

The boundary values at the postsynaptic $x_1 = d$ are given by the indices j in the set

$$\mathbb{B}_3 = \mathbb{N}_{n_1 n_2 - n_2}^{n_1 n_2 - 1}.$$

The left side boundary $x_2 = 0$ are given by the indices j in the set

$$\mathbb{B}_4 = \{\ell n_2\}_{\ell=1}^{n_1-2} .$$

The right side boundary $x_2 = w$ are given by the indices j in the set

$$\mathbb{B}_5 = \{\ell n_2 - 1\}_{\ell=2}^{n_1-1} .$$

And the boundary as a whole are given by the indices j in the set

$$\mathbb{B} = \bigcup_{i=1}^{5} \mathbb{B}_i .$$

The inner points are given by the indices j in the set

$$\mathbb{I} = \mathbb{N}_0^{n_1 n_2 - 1} \setminus \mathbb{B} .$$

We can now write the matrix elements of $\mathbf{A}_{(i+1)}$

$$a_{(i+1)jk} = a_{jk} = \begin{cases} 1 & : \forall j \in \mathbb{B} \setminus \mathbb{B}_2 : j = k & \text{(fixed boundary)} \\ 1 + 2\theta\alpha_2 & : \forall j \in \mathbb{B}_2 : j = k & \text{(side of vesicles)} \\ 1 + 2\theta \sum_{\ell=1}^{2} \alpha_\ell & : \forall j \in \mathbb{I} : j = k & \text{(the inner points } u_{(i+1)j_1 j_2}) \\ -\theta\alpha_2 & : \forall j \in \mathbb{I} \cup \mathbb{B}_2 : k \in \left\{j-1, j+1\right\} & \text{(diffusion in } x_2 \text{ direction)} \\ -\theta\alpha_1 & : \forall j \in \mathbb{I} : k \in \left\{j-n_2, j+n_2\right\} & \text{(diffusion in } x_1 \text{ direction)} \\ 0 & : \text{otherwise,} & \end{cases} \tag{23}$$

and the elements of the vector $\mathbf{b}_i$ are

$$b_{ij} = b_{i(j_1 n_2 + j_2)} = \begin{cases} 1 & : j \in \mathbb{B}_1 \\ \left(1 - 2\left(1-\theta\right)\alpha_2\right) u_{i0 j_2} + \left(1-\theta\right)\alpha_2 \left(u_{i0(j_2+1)} + u_{i0(j_2-1)}\right) & : j \in \mathbb{B}_2 \\ \left(1 - 2\left(1-\theta\right)\sum_{\ell=1}^{2}\alpha_\ell\right) u_{i\{j_k\}_{k=1}^{2}} + \left(1-\theta\right)\sum_{\ell=1}^{2}\alpha_\ell \left(u_{i\{j_k+\delta_{k\ell}\}_{k=1}^{2}} + u_{i\{j_k-\delta_{k\ell}\}_{k=1}^{2}}\right) & : j \in \mathbb{I} \\ 0 & : \text{otherwise.} \end{cases} \tag{24}$$

Unfortunately is the linear algebra problem $\mathbf{A}\mathbf{v}_{i+1} = \mathbf{b}_i$ is computational expensive to solve for the matrix in (23). To lower the computational time I therefore prepare the matrix $\mathbf{A}$ for Jacobi's iterative method by splitting it into a diagonal matrix $\mathbf{D}$ and a remainder matrix $\mathbf{R}$;

$$\mathbf{A} = \mathbf{D} + \mathbf{R} .$$

From (23) we see that the elements of the diagonal matrix are given by

$$d_{jk} = \begin{cases} 1 & : \forall j \in \mathbb{B} \setminus \mathbb{B}_2 : j = k \quad \text{(fixed boundary)} \\ 1 + 2\theta\alpha_2 & : \forall j \in \mathbb{B}_2 : j = k \quad \text{(side of vesicles)} \\ 1 + 2\theta \sum_{\ell=1}^{2} \alpha_\ell & : \forall j \in \mathbb{I} : j = k \quad \text{(the inner points } u_{(i+1)j_1 j_2}\text{)} \\ 0 & : \text{otherwise,} \end{cases} \tag{25}$$

and the elements of the remainder matrix are given by

$$r_{jk} = \begin{cases} -\theta\alpha_2 & : \forall j \in \mathbb{I} \cup \mathbb{B}_2 : k \in \left\{ j-1, j+1 \right\} \quad \text{(diffusion in } x_2 \text{ direction)} \\ -\theta\alpha_1 & : \forall j \in \mathbb{I} : k \in \left\{ j-n_2, j+n_2 \right\} \quad \text{(diffusion in } x_1 \text{ direction)} \\ 0 & : \text{otherwise.} \end{cases} \tag{26}$$

We can now introduce Jacobi's iterative method

$$\mathbf{v}_{i+1}^{(\ell)} = \begin{cases} \mathbf{v}_i & : \ell = 0 \\ \mathbf{D}^{-1} \left(\mathbf{b}_i - \mathbf{R}\mathbf{v}_{i+1}^{(\ell-1)} \right) & : \ell \in \mathbb{N}_1 \end{cases}$$

where ℓ is the number of iterations and $\ell = 0$ are the starting point. Rewriting it on element form yields

$$v_{(i+1)j}^{(\ell)} = \begin{cases} v_j & : \ell = 0 \\ \frac{1}{d_{jj}} \left(b_j - \sum_{k \neq j} r_{jk} v_{(i+1)k}^{(\ell-1)} \right) & : j \in \mathbb{N}_1 \,. \end{cases}$$

Using (22) and (26) we have the iterative solution to the heat equation in (6) with the boundaries in (2)

$$u_{(i+1)\{j_k\}_{k=1}^2}^{(\ell)} = \begin{cases} u_{i\{j_k\}_{k=1}^2} & : \ell = 0 \\ 1 & : j_1 = 0 \wedge j_2 \in \mathbb{N}_{m_1}^{m_2} \\ c_0 + c_1 \left(u_{(i+1)0(j_2+1)}^{(\ell-1)} + u_{(i+1)0(j_2-1)}^{(\ell-1)} \right) & : j_1 = 0 \wedge j_2 \in \mathbb{N}_0^{n_2-1} \setminus \mathbb{N}_{m_1}^{m_2} \\ c_2 + \sum_{o=1}^{2} c_{o+2} \left(u_{(i+1)\{j_k+\delta_{ko}\}_{k=1}^2}^{(\ell-1)} + u_{(i+1)\{j_k-\delta_{ko}\}_{k=1}^2}^{(\ell-1)} \right) & : j_1 \in \mathbb{N}_1^{n_1-2} \wedge j_2 \in \mathbb{N}_1^{n_2-2} \end{cases} \tag{27}$$

$$c_0 = c_5 u_{i0j_2} + c_6 \left(u_{i0(j_2+1)} + u_{i0(j_2-1)} \right) \tag{28}$$

$$c_1 = \frac{\theta\alpha_2}{1 + 2\theta\alpha_2} \tag{29}$$

$$c_2 = c_7 u_{i\{j_k\}_{k=1}^2} + \sum_{\ell=1}^{2} c_{\ell+7} \left(u_{i\{j_k+\delta_{k\ell}\}_{k=1}^2} + u_{i\{j_k-\delta_{k\ell}\}_{k=1}^2} \right) \tag{30}$$

$$c_3 = \frac{\theta\alpha_1}{1 + 2\theta \sum_{\ell=1}^{2} \alpha_\ell} \tag{31}$$

$$c_4 = \frac{\theta\alpha_2}{1 + 2\theta \sum_{\ell=1}^{2} \alpha_\ell} \tag{32}$$

$$c_5 = \frac{1 - 2(1-\theta)\alpha_2}{1 + 2\theta\alpha_2} \tag{33}$$

$$c_6 = \frac{(1-\theta)\alpha_2}{1 + 2\theta\alpha_2} \tag{34}$$

$$c_7 = \frac{1 - 2(1-\theta)\sum_{\ell=1}^{2} \alpha_\ell}{1 + 2\theta \sum_{\ell=1}^{2} \alpha_\ell} \tag{35}$$

$$c_8 = \frac{(1-\theta)\alpha_1}{1 + 2\theta \sum_{\ell=1}^{2} \alpha_\ell} \tag{36}$$

$$c_9 = \frac{(1-\theta)\alpha_2}{1 + 2\theta \sum_{\ell=1}^{2} \alpha_\ell} \tag{37}$$

2.4.1 Stability and convergence

The same procedure as discussed for convergence and stability for the explicit scheme applies for the Jacobi method as well, but now with the spectral radius $\rho\left(\mathbf{D}^{-1}\mathbf{R}\right) < 1$, because the Jacobi method are given by

$$\mathbf{v}_{i+1}^{(\ell)} = \mathbf{D}^{-1}\left(\mathbf{b}_i - \mathbf{R}\mathbf{v}_{i+1}^{(\ell-1)}\right) = \mathbf{D}^{-1}\sum_{j=0}^{\ell-1}\left(-\mathbf{D}^{-1}\mathbf{R}\right)^j \mathbf{b}_i + \left(-\mathbf{D}^{-1}\mathbf{R}\right)^\ell \mathbf{v}_{i+1}^{(0)}\,.$$

Similar steps as shown in section 2.3.1 it can be shown that the always converges for our diffusion problem. More generally it can be shown that the Jacobi iterative method converges when the matrix $\mathbf{A} = \mathbf{D} + \mathbf{R}$ is diagonally dominant

$$\left|a_{ii}\right| > \sum_{j\neq i} \left|a_{ij}\right|\,,$$

which is the case for the matrix spanned out by (23) because $1 + 2\theta\alpha_2 > 2\theta\alpha_2$ and $1 + 2\theta\sum_{\ell=1}^{2}\alpha_\ell > 2\theta\sum_{\ell=1}^{2}\alpha_\ell$.

2.5 Markov chains

I assume that diffusion is a memoryless physical process that approximates to a stochastic process, hence diffusion is assumed to satisfy the Markov property and therefore is a Markov process where the transition between states are described by Markov chains

$$u^{+}_{(i+1)\{j_k\}_{k=1}^{n}} = \sum_{\{\ell_k\}_{k=1}^{n}} W_{\{j_k\}_{k=1}^{n}\{\ell_k\}_{k=1}^{n}} u_{i\{\ell_k\}_{k=1}^{n}}\,, \tag{38}$$

where $u_{i\{j_k\}_{k=1}^{n}}$ is the probability distribution function "PDF" at time step i, and $W_{\{j_k\}_{k=1}^{n}\{\ell_k\}_{k=1}^{n}}$ is the transition probability from state $\{\ell_k\}_{k=1}^{n}$ to $\left\{j_k\right\}_{k=1}^{n}$. And the reverse Markov chain is given by

$$u^{-}_{(i+1)\{j_k\}_{k=1}^{n}} = \sum_{\{\ell_k\}_{k=1}^{n}} W_{\{\ell_k\}_{k=1}^{n}\{j_k\}_{k=1}^{n}} u_{i\{j_k\}_{k=1}^{n}}\,, \tag{39}$$

Using the assumption that diffusion can be modelled as Markov process then we can use a Master equation to describe the normalized concentration at time step $i+1$ by the difference of transition to and from the state $\left\{j_k\right\}_{k=1}^{n}$ plus the amount that was in the state;

$$u_{(i+1)\{j_k\}_{k=1}^{n}} = u_{i\{j_k\}_{k=1}^{n}} + u^{+}_{(i+1)\{j_k\}_{k=1}^{n}} - u^{-}_{(i+1)\{j_k\}_{k=1}^{n}} \tag{40}$$

Note that the reverse Markov chain is all the transitions that a state $\left\{j_k\right\}_{k=1}$ takes, including transition to itself. Therefore the reverse Markov chain in time step $i+1$ describes all the transitions that the state does at time step i;

$$u_{i\{j_k\}_{k=1}^{n}} = u^{-}_{(i+1)\{j_k\}_{k=1}^{n}}\,,$$

which yields

$$u_{(i+1)\{j_k\}_{k=1}^{n}} = u^{+}_{(i+1)\{j_k\}_{k=1}^{n}}\,. \tag{41}$$

Further, diffusion is assumed to be approximated as stochastic move in distance and direction, where distance and direction is stochastically independent. Which means that transition probability can be written as

$$W_{\{j_k\}_{k=1}^{n}\{\ell_k\}_{k=1}^{2}} = T_{\{j_k\}_{k=1}^{n}\{\ell_k\}_{k=1}^{2}} A_{\{j_k\}_{k=1}^{n}\{\ell_k\}_{k=1}^{2}}\,, \tag{42}$$

where $T_{\{j_k\}_{k=1}^{n}\{\ell_k\}_{k=1}^{2}}$ is the probability of moving the distance between the states $\left\{j_k\right\}_{k=1}^{n}$ and $\{\ell_k\}_{k=1}^{2}$, and $A_{\{j_k\}_{k=1}^{n}\{\ell_k\}_{k=1}^{2}}$ is the probability of moving in the direction between the states $\left\{j_k\right\}_{k=1}^{n}$ and $\{\ell_k\}_{k=1}^{2}$.

Assume that we move a distance ℓ from state $\left\{j_k\right\}_{k=1}^{n}$, which means that the states $\left\{j_k + \delta_{km}\ell\right\}$ are the possible destinations, which puts the following constraint on the direction probability A;

$$\sum_{m=1}^{n}\left(A_{\{j_k+\delta_{km}\ell\}_{k=1}^{n}\{j_k\}_{k=1}^{n}} + A_{\{j_k-\delta_{km}\ell\}_{k=1}^{n}\{j_k\}_{k=1}^{n}}\right) = 1\,. \tag{43}$$

Now assuming isotropic diffusion, which means that

$$\forall m, o \in \mathbb{N}_1^n : A_{\{j_k \pm \delta_{km}\ell\}_{k=1}^n \{j_k\}_{k=1}^n} = A_{\{j_k \pm \delta_{ko}\ell\}_{k=1}^n \{j_k\}_{k=1}^n}$$
$$\forall m, o \in \mathbb{N}_1^n : T_{\{j_k \pm \delta_{km}\ell\}_{k=1}^n \{j_k\}_{k=1}^n} = T_{\{j_k \pm \delta_{ko}\ell\}_{k=1}^n \{j_k\}_{k=1}^n}$$

and the constraint on the direction probability then yields

$$A = A_{\{j_k\}_{k=1}^n \{\ell_k\}_{k=1}^n} = \frac{1}{2n}\,, \tag{44}$$

and the distance probability T is only dependent on the distance ℓ

$$p_\ell = T_{\{j_k \pm \delta_{km}\ell\}_{k=1}^n \{j_k\}_{k=1}^n}\,,$$

which has the constraint

$$\sum_\ell p_\ell = 1\,.$$

Then the normalized concentration (41) is now given by

$$u_{(i+1)\{j_k\}_{k=1}^n} = \frac{1}{2n} \sum_\ell p_\ell \sum_{m=1}^n \left(u_{i\{j_k + \delta_{km}\ell\}_{k=1}^n} + u_{i\{j_k - \delta_{km}\ell\}_{k=1}^n} \right),$$

which can be rewritten to

$$\frac{u_{(i+1)\{j_k\}_{k=1}^n} - u_{i\{j_k\}_{k=1}^n}}{\Delta t} = \sum_\ell \frac{p_\ell\,(\ell\Delta x)^2}{2n\Delta t} \sum_{m=1}^n \frac{u_{i\{j_k+\delta_{km}\ell\}_{k=1}^n} - 2u_{i\{j_k\}_{k=1}^n} + u_{i\{j_k-\delta_{km}\ell\}_{k=1}^n}}{(\ell\Delta x)^2}\,. \tag{45}$$

When we have the Kronecker delta distribution $p_\ell = \delta_{1\ell}$ this becomes the discretization of the heat equation in (1)

$$\frac{u_{(i+1)\{j_k\}_{k=1}^n} - u_{i\{j_k\}_{k=1}^n}}{\Delta t} = \frac{\Delta x^2}{2n\Delta t} \sum_{m=1}^n \frac{u_{i\{j_k+\delta_{km}\}_{k=1}^n} - 2u_{i\{j_k\}_{k=1}^n} + u_{i\{j_k-\delta_{km}\}_{k=1}^n}}{\Delta x^2}\,,$$

with the relation to the diffusion constant D as follows

$$\Delta x = \sqrt{2nD\Delta t}\,. \tag{46}$$

For a general distribution p_ℓ we see from (45) that the step length now is given by

$$\Delta x_\ell = \ell\Delta x = \ell\sqrt{2nD\Delta t}\,. \tag{47}$$

If we use a random generator to make a proposition to move at each time step, we get a deterministic running time with approximation in the result, which is known as Monte Carlo algorithm.

2.5.1 Metropolis-Hastings algorithm

The steps above with Markov chains are designed to give a approximated time evolution. However if we are only interested in the steady state solution, we can derive a faster algorithm to reach it, which is the Metropolis-Hastings algorithm. To derive the Metropolis-Hastings algorithm we start by looking back at the Master equation in (40) for equilibrium, where we realize that transition to and from the state $\left\{j_k\right\}_{k=1}^n$ must be equal

$$\lim_{i\to\infty}\left(u^+_{(i+1)\{j_k\}_{k=1}^n} - u^-_{(i+1)\{j_k\}_{k=1}^n}\right) = 0\,.$$

Using the equality of transition to and from the state $\left\{j_k\right\}_{k=1}^n$ at equilibrium, the Markov chains in (38) and (39), with proposed transition distribution T which is independent of the accepted A, must satisfy the following equation

$$\sum_{\{\ell_k\}_{k=1}^n}\left(T_{\{j_k\}_{k=1}^n\{\ell_k\}_{k=1}^n}A_{\{j_k\}_{k=1}^n\{\ell_k\}_{k=1}^n}u_{i\{\ell_k\}_{k=1}^n} - T_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}A_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}u_{i\{j_k\}_{k=1}^n}\right) = 0\,,$$

which is satisfied by the relation

$$\frac{A_{\{j_k\}_{k=1}^n\{\ell_k\}_{k=1}^n}}{A_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}} = \frac{T_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}u_{i\{j_k\}_{k=1}^n}}{T_{\{j_k\}_{k=1}^n\{\ell_k\}_{k=1}^n}u_{i\{\ell_k\}_{k=1}^n}}\,,$$

where T is a distribution that we propose and A is a probability to accept the proposition. We reach equilibrium faster when the transition from states are as large as possible, which means $A_{\{j_k\}_{k=1}^n\{\ell_k\}_{k=1}^n}$ should be as close to 1 as possible and still satisfy that $A_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n} \leq 1$, which is the Metropolis-Hastings algorithm;

$$A_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n} = \min\left(1, \frac{T_{\{j_k\}_{k=1}^n\{\ell_k\}_{k=1}^n}u_{i\{\ell_k\}_{k=1}^n}}{T_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}u_{i\{j_k\}_{k=1}^n}}\right), \tag{48}$$

where the two states change accordingly to a Master equation when we transition from $\left\{j_k\right\}_{k=1}^2$ to $\{\ell_k\}_{k=1}^2$

$$u_{(i+1)\{j_k\}_{k=1}^n} = T_{\{j_k\}_{k=1}^n\{\ell_k\}_{k=1}^n}A_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}u_{i\{\ell_k\}_{k=1}^n} + \left(1 - T_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}A_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}\right)u_{i\{j_k\}_{k=1}^n} \tag{49}$$

$$u_{(i+1)\{\ell_k\}_{k=1}^n} = T_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}A_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}u_{i\{j_k\}_{k=1}^n} + \left(1 - T_{\{j_k\}_{k=1}^n\{\ell_k\}_{k=1}^n}A_{\{\ell_k\}_{k=1}^n\{j_k\}_{k=1}^n}\right)u_{i\{\ell_k\}_{k=1}^n}\,. \tag{50}$$

2.5.2 Wall boundary

The Markov chain give insight into how to treat a boundary that is a wall. A wall is a boundary that does not allow transitions beyond it. If we look at direction probability A in (43) and assume that our wall sets $A_{\left\{j_k-\delta_{kp}\right\}_{k=1}^n\{j_k\}_{k=1}^n} = 0$ we get

$$A_{\left\{j_k+\delta_{kp}\ell\right\}_{k=1}^n\{j_k\}_{k=1}^n} + \sum_{\substack{m=1\\m\neq p}}^{n}\left(A_{\{j_k+\delta_{km}\ell\}_{k=1}^n\{j_k\}_{k=1}^n} + A_{\{j_k-\delta_{km}\ell\}_{k=1}^n\{j_k\}_{k=1}^n}\right) = 1\,,$$

and imposing isotropy the direction probability at the wall becomes

$$A = A_{\{j_k\}_{k=1}^n \{\ell_k\}_{k=1}^n} = \frac{1}{2n-1}\,.$$

Now our Master equation becomes

$$u_{(i+1)\{j_k\}_{k=1}^n} = \frac{1}{2n-1}\sum_\ell p_\ell \left(u_{i\left\{j_k+\delta_{kp}\ell\right\}_{k=1}^n} + \sum_{\substack{m=1\\ m\neq p}}^{n} \left(u_{i\{j_k+\delta_{km}\ell\}_{k=1}^n} + u_{i\{j_k-\delta_{km}\ell\}_{k=1}^n} \right) \right)$$

which can be rewritten to

$$\begin{aligned}\frac{u_{(i+1)\{j_k\}_{k=1}^n} - u_{i\{j_k\}_{k=1}^n}}{\Delta t} =& \sum_\ell \frac{p_\ell\,(\ell\Delta x)^2}{(2n-1)\,\Delta t} \sum_{\substack{m=1\\ m\neq p}}^{n} \frac{u_{i\{j_k+\delta_{km}\ell\}_{k=1}^n} - 2u_{i\{j_k\}_{k=1}^n} + u_{i\{j_k-\delta_{km}\ell\}_{k=1}^n}}{(\ell\Delta x)^2} \\ &+ \sum_\ell \frac{p_\ell \ell \Delta x}{(2n-1)\,\Delta t} \frac{u_{i\left\{j_k+\delta_{kp}\ell\right\}_{k=1}^n} - u_{i\{j_k\}_{k=1}^n}}{\ell\Delta x}\,.\end{aligned}$$

When we have the Kronecker delta distribution $p_\ell = \delta_{1\ell}$ this becomes the following discretization

$$\begin{aligned}\frac{u_{(i+1)\{j_k\}_{k=1}^n} - u_{i\{j_k\}_{k=1}^n}}{\Delta t} =& \frac{\Delta x^2}{(2n-1)\,\Delta t} \sum_{\substack{m=1\\ m\neq p}}^{n} \frac{u_{i\{j_k+\delta_{km}\}_{k=1}^n} - 2u_{i\{j_k\}_{k=1}^n} + u_{i\{j_k-\delta_{km}\}_{k=1}^n}}{\Delta x^2} \\ &+ \frac{\Delta x}{(2n-1)\,\Delta t} \frac{u_{i\left\{j_k+\delta_{kp}\ell\right\}_{k=1}^n} - u_{i\{j_k\}_{k=1}^n}}{\Delta x}\,.\end{aligned}$$

Relating to the diffusion constant D gives

$$\Delta x = \sqrt{(2n-1)\,D\Delta t}\,, \tag{51}$$

and in general

$$\Delta x_\ell = \ell\Delta x = \ell\sqrt{(2n-1)\,D\Delta t} \tag{52}$$

which results in the discretization

$$\begin{aligned}\frac{u_{(i+1)\{j_k\}_{k=1}^n} - u_{i\{j_k\}_{k=1}^n}}{\Delta t} =& D \sum_{\substack{m=1\\ m\neq p}}^{n} \frac{u_{i\{j_k+\delta_{km}\}_{k=1}^n} - 2u_{i\{j_k\}_{k=1}^n} + u_{i\{j_k-\delta_{km}\}_{k=1}^n}}{\Delta x^2} \\ &+ \frac{D}{\Delta x} \frac{u_{i\left\{j_k+\delta_{kp}\right\}_{k=1}^n} - u_{i\{j_k\}_{k=1}^n}}{\Delta x}\,.\end{aligned}$$

To avoid that the above equation diverges for smaller an smaller step sizes, we realize that the we must satisfy the following the partial differential equation at the wall boundary

$$\frac{\partial u\left(\{x_i\}_{i=1}^n, t\right)}{\partial t} = D \sum_{\substack{m=1 \\ m \neq p}}^{n} \frac{\partial^2 u\left(\{x_i\}_{i=1}^n, t\right)}{\partial {x_m}^2} \qquad \text{and} \qquad \frac{\partial u\left(\{x_i\}_{i=1}^n, t\right)}{\partial x_p} = 0\,, \tag{53}$$

and we have proven the ansatz that I made in the analytical derivation where the boundary $u\left(0, x_2, t\right)$ is determined by $u_2\left(x_2, t\right)$ which yielded the solution on the boundary as in (6).

3 Implementation

3.1 Chains

An important part of the Monte Carlo simulation of diffusion is memory management of the particles that are added and removed continuously in the simulation. Regular arrays are inefficient to use as storage in this case, which would require continuously allocation and deallocation of the complete array of particles for every time we add or remove a particle. Therefore I made a `struct Chain` where it's objects points to two neighbour objects in such a way that it form a chain that is similar to an array. Each `Chain` object contain an template object of data, so it's functionality is like an array. However the advantage is that you can break the chain at any point you want and remove or add elements and recombine the chain again without having to reallocate the memory of the other elements in the chain. This is achieved by the functions `Chain::Add` and `Chain::Remove`.

`template<class T> struct Chain`

```
template<class T> struct Chain {
  unsigned int N;
  T e;
  Chain<T>* owner;
  Chain<T>* prev;
  Chain<T>* next;

  Chain() {
    N = 0;
    owner = this;
    prev = this;
    next = this;
  }

  ~Chain() {
    if(N) {
      Chain<T>* p = this;
      for(int i = 1; i < N; i++)
        p = p->next;
      for(int i = 0; i < N; i++) {
        delete p->next;
        p = p->prev;
      }
    }
  }
```

```
  void Add(T element) {        // Add next
    if(next == this) {
      next = new Chain<T>;
      next->next = next;
    }
    else {
      next->prev = new Chain<T>;
      next->prev->next = next;
      next = next->prev;
    }
    owner->N++;
    next->owner = owner;
    next->prev = this;
    next->e = element;
  }

  void Remove() {              // Remove next
    if(next->next == next) {
      delete next;
      next = this;
    } else {
      next = next->next;
      delete next->prev;
      next->prev = this;
    }
    owner->N--;
  }
};
```

3.2 Space

Since this project makes extensive use of 2D arrays, it was important to make simple use of multidimensional arrays, since there is a lot of hazel to manage multidimensional arrays. For generality I made `struct Space` to make it easy to allocate and deallocate memory of an arbitrary dimensional array, where the dimension is set by the template argument `D`. To enable this feature I needed a generic way to represent an arbitrary dimensional pointer, which I did with a very simple `struct Pointer` that inherit itself till the correct dimensionality of the pointer is reached. This is done by adding an astric for each parent, and the dimensionality of the pointer is given by a template argument `D`.

`struct Pointer`

```
template<typename T, unsigned int D> struct Pointer : Pointer<T*,D-1> {
};

template<typename T> struct Pointer<T,0> {
  typedef T Type;
};
```

It's not only memory management that is useful operation to do on a multidimensional array. Because the `struct Space` enables a easy way to traverse all the elements in a multidimensional array by recursively call same member function, it was natural to equip `Space` with different functionality to operate on an multidimensional array, for instance mathematical operation as adding two multidimensional arrays through `Space<T,D>::Add`, or normalize all the elements through

`Space<T,D>::Normalize`. I also made a easy to use function for writing an arbitrary dimensional array to file `Space<T,D>::ToFile`, and I also made functions in Python in plot.py to read this files and rebuild it to a multidimensional array ready to plot.

A very important functionality in the Monte Carlo simulations in this project was to map the position of a particle into a multidimensional array, therefore I made a function `Space<T,D>::Map` that takes a chain of particles maps into a space that is spanned out by a multidimensional array.

Because of the extensive and important use of `Space` in my work in this project, I will display it's code here:

struct Space

```
template<typename T, unsigned int D> struct Space {

  static void Add(typename Pointer<T,D>::Type s1, typename Pointer<T,D>::
      Type s2, unsigned int n[D]) {
    for(int i = 0; i < n[0]; i++)
      Space<T,D-1>::Add(s1[i], s2[i], &n[1]);
  }

  static typename Pointer<T,D>::Type Allocate(unsigned int n[D]) {
    typename Pointer<T,D>::Type array = new typename Pointer<T,D-1>::Type[n
        [0]];
    for(int i = 0; i < n[0]; i++)
      array[i] = Space<T,D-1>::Allocate(&n[1]);
    return array;
  }

  static void ArrayFile(ofstream& file, typename Pointer<T,D>::Type array,
      unsigned int n[D]) {
    for(int i = 0; i < n[0]; i++) {
      if(D == 1 && i)
        file << '\t';
      else if(D > 1 && i)
        file << endl;
      Space<T,D-1>::ArrayFile(file, array[i], &n[1]);
    }
  }

  template<typename V> static typename Pointer<T,D>::Type Cast(typename
      Pointer<V,D>::Type space, unsigned int n[D]) {
    typename Pointer<T,D>::Type array = new typename Pointer<T,D-1>::Type[n
        [0]];
    for(int i = 0; i < n[0]; i++)
      array[i] = Space<T,D-1>::Cast(space[i], &n[1]);
    return array;
  }

  static void Deallocate(typename Pointer<T,D>::Type array, unsigned int n[
      D]) {
    for(int i = 0; i < n[0]; i++)
      Space<T,D-1>::Deallocate(array[i], &n[1]);
    delete [] array;
  }
```

```
static void DeRange(T** range) {
  for(int i = 0; i < D; i++)
    delete [] range[i];
  delete [] range;
}

template<typename V> static typename Pointer<V,D>::Type Map(Chain<Vector<
    T,D>>& chain, T* range[D], unsigned int n[D]) {
  auto space = Space<V,D>::Allocate(n);
  auto p = chain.owner;
  for(unsigned int i = 0; i < chain.N; i++) {
    p = p->next;
    Space<T,D>::Mapping<V>(space, p->e.e, range, n);
  }
  return space;
}

template<typename V> static void Map(typename Pointer<V,D>::Type& space,
    Chain<Vector<T,D>>& chain, T* range[D], unsigned int n[D]) {
  auto p = chain.owner;
  for(unsigned int i = 0; i < chain.N; i++) {
    p = p->next;
    Space<T,D>::Mapping<V>(space, p->e.e, range, n);
  }
}

template<typename V> static void Mapping(typename Pointer<V,D>::Type
    space, T element[D], T* range[D], unsigned int n[D]) {
  unsigned int m = n[0]-1;
  if(element[0] >= range[0][0] || element[0] <= range[0][m]) {
    for(unsigned int i = 0; i < m; i++) {
      if(element[0] < (range[0][i]+range[0][i+1])/2) {
        Space<T,D-1>::template Mapping<V>(space[i], &element[1], &range
            [1], &n[1]);
        return;
      }
    }
    Space<T,D-1>::template Mapping<V>(space[m], &element[1], &range[1], &
        n[1]);
  }
}

static T Max(typename Pointer<T,D>::Type space, unsigned int n[D]) {
  T val, max = 0;
  for(unsigned int i = 0; i < n[0]; i++) {
    val = Space<T,D-1>::Max(space[i], &n[1]);
    if(val > max)
      max = val;
    }
  return max;
}

static void Normalize(typename Pointer<T,D>::Type space, unsigned int n[D
    ], T val) {
```

```
    for(int i = 0; i < n[0]; i++)
      Space<T,D-1>::Normalize(space[i], &n[1], val);
  }

  template<typename V> static typename Pointer<T,D>::Type Normalize(
      typename Pointer<V,D>::Type space, unsigned int n[D], V val) {
    typename Pointer<T,D>::Type array = new typename Pointer<T,D-1>::Type[n
        [0]];
    for(int i = 0; i < n[0]; i++)
      array[i] = Space<T,D-1>::template Normalize<V>(space[i], &n[1], val);
    return array;
  }

  static T** Range(T lower[D], T upper[D], unsigned int n[D]) {
    T** array = new T*[D];
    for(int i = 0; i < D; i++) {
      T step = (upper[i]-lower[i])/(n[i]-1);
      array[i] = new T[n[i]];
      array[i][0] = lower[i];
      for(int j = 1; j < n[i]; j++)
        array[i][j] = array[i][j-1] + step;
      }
    return array;
  }

  static void RangeFile(ofstream& file, T* range[D], unsigned int n[D]) {
    for(int i = 0; i < D; i++) {
      if(i)
        file << endl;
      Space<T,1>::ArrayFile(file, range[i], &n[i]);
    }
  }

  static void ToFile(ofstream& file, T* range[D], typename Pointer<T,D>::
      Type array, unsigned int n[D]) {
    file << D << endl;
    RangeFile(file, range, n);
    file << endl;
    ArrayFile(file, array, n);
  }
};

template<typename T> struct Space<T,0> {

  static void Add(T& s1, T& s2, unsigned int[0]) {
    s1 += s2;
  }

  static T Allocate(unsigned int[0]) {
    return T(0);
  }

  static void ArrayFile(ofstream& file, T array, unsigned int[0]) {
    file << array;
  }
```

```
  template<typename V> static T Cast(V& space, unsigned int[0]) {
    return static_cast<T>(space);
  }

  static void Deallocate(T, unsigned int[0]) {
  }

  template<typename V> static void Mapping(typename Pointer<V,0>::Type&
      space, T[0], T*[0], unsigned int[0]) {
    space++;
  }

  static T Max(T space, unsigned int[0]) {
    return space;
  }

  static T Normalize(T& space, unsigned int[0], T val) {
    space /= val;
  }

  template<typename V> static T Normalize(V space, unsigned int[0], V val)
      {
    return static_cast<T>(space) / static_cast<T>(val);
  }
};
```

3.3 Experiment

The Monte Carlo simulation is a stochastic method and is therefore of interest to run them in many experiments to get a better result. In fact the Monte Carlo simulation I did in this project I used only 1 seeding particle for each experiment, and the result from one 1 experiment is unusable, but with many experiments one gets good result.

A experiment in general can have arbitrary number of input parameters, therefore I made a variadic template function that takes a arbitrary number of input parameters that are sent to a function pointer for the experiment that we want to run many times. Each experiment should return a chain that is mapped and accumulated in a multidimensional array. And when the experiments are done the multidimensional array are normalized with the largest value in the array.

Experiment

```
template<typename V, typename T, unsigned int D, typename C, typename... P>
    typename Pointer<T,D>::Type Experiment(V N, T* x[D], unsigned int n[D],
    Delegate<C,Chain<Vector<T,D>>,P...> experiment, P... arg) {

  auto space = Space<V,D>::Allocate(n);

  for(int i = 0; i < N; i++) {
    Chain<Vector<T,D>> states = experiment(arg...);
    Space<T,D>::template Map<V>(space, states, x, n);
  }

  auto space2 = Space<T,D>::template Normalize<V>(space, n, Space<V,D>::Max
```

```
    (space, n));
  Space<V,D>::Deallocate(space, n);
  return space2;
}
```

3.4 Monte Carlo

I made a Monte Carlo method that solves 1D and 2D diffusion problem which has only 1 seeding particle at the source. Which means that when this seeding particle moves, a new particle seeding particle is added at the source to take the place of the old seeding particle.

Diffusion1D_MonteCarlo

```
template<typename C, typename T> Chain<Vector<T,1>> Diffusion1D_MonteCarlo(
    T t, T dt, T d, Delegate<C,T> step) {

  T val;
  unsigned int n = t / dt;
  T dx = sqrt(2*dt);
  uniform_real_distribution<T> pdf(0.0,1.0);  // Distribution for accepting
      a move
  default_random_engine rng;                  // Set RNG seeding value
  rng.seed(chrono::high_resolution_clock::now().time_since_epoch().count())
      ;

  Chain<Vector<T,1>> particles, *p;
  Vector<T,1> vec = {0};
  particles.Add(vec);                         // Seeding particle

  for(int i = 0; i < n; i++) {                // Loop of timesteps
    p = &particles;
    for(int j = 0; j < particles.N; j++) {    // Loop of particles
      val = pdf(rng);                         // Random number
      p = p->next;                            // Next particle

      if(val <= 0.5) {                        // Sampling rule

        val = p->e.e[0] - dx * step();        // Calculate backward move
        if(val > 0)                           // Valid move
          p->e.e[0] = val;

      } else {

        if(p->e.e[0] == 0) {                  // Particle at the source
          p->prev->Add(p->e);                 // Add new particle at prev
          j++;
        }
        p->e.e[0] += dx * step();             // Forward move particle
        if(p->e.e[0] >= d) {                  // Remove particle
          p = p->prev;                        // into the postsynaptic
          p->Remove();
          j--;
        }
      }
    }
```

```
  }

  return particles;
}
```

Diffusion2D_MonteCarlo

```
template<typename C, typename T> Chain<Vector<T,2>> Diffusion2D_MonteCarlo(
    T t, T dt, T d[2], T w[2], Delegate<C,T> step) {

  T val;
  unsigned int n = t / dt;
  T dx = sqrt(4*dt);
  T dx0 = sqrt(3*dt);
  uniform_real_distribution<T> pdf(0.0,1.0);  // Distribution for accepting
      a move
  default_random_engine rng;                  // Set RNG seeding value
  rng.seed(chrono::high_resolution_clock::now().time_since_epoch().count())
      ;

  Chain<Vector<T,2>> particles, *p;
  T dw = w[1]-w[0];
  Vector<T,2> vec = {0,0};
  vec.e[1] = w[0] + pdf(rng)*dw;              // Initial position
  particles.Add(vec);                         // of seeding particles

  for(int i = 0; i < n; i++) {                // Loop of timesteps
    p = &particles;
    for(int j = 0; j < particles.N; j++) {    // Loop of particles
      p = p->next;                            // Next particle
      val = pdf(rng);                         // Random number

      if(p->e.e[0] == 0) {                    // Wall boundary

        if(val <= (T)1/3) {                   // x direction

          if(p->e.e[1] >= w[0] && p->e.e[1] <= w[1]) {
            vec.e[1] = w[0] + pdf(rng)*dw;
            p->prev->Add(vec);                // Add new particle
            j++;
          }
          p->e.e[0] += dx0 * step();          // Forward move particle
          if(p->e.e[0] >= d[0]) {             // Remove particle
            p = p->prev;                      // into the postsynaptic
            p->Remove();
            j--;
          }

        } else if(val <= (T)2/3) {            // y direction

          val = p->e.e[1] - dx0 * step();
          if(val < w[0] || val > w[1]) {
            p->e.e[1] = val;                  // Backward move
            if(p->e.e[1] <= 0) {              // Remove particle
              p = p->prev;                    // outside cleft
              p->Remove();
```

```
            j--;
          }
        }

      } else {

        val = p->e.e[1] + dx0 * step();
        if(val < w[0] || val > w[1]) {
          p->e.e[1] = val;                    // Backward move
          if(p->e.e[1] >= d[1]) {             // Remove particle
            p = p->prev;                      // outside cleft
            p->Remove();
            j--;
          }
        }
      }

    } else {                                  // Free particle

      if(val <= 0.25) {                       // x direction

        val = p->e.e[0] - dx * step();        // Backward move
        if(val > 0)                           // Valid move
          p->e.e[0] = val;

      } else if(val <= 0.5) {

        p->e.e[0] += dx * step();             // Forward move
        if(p->e.e[0] >= d[0]) {               // Remove particle
          p = p->prev;                        // into the postsynaptic
          p->Remove();
          j--;
        }

      } else if(val <= 0.75) {                // y direction

        p->e.e[1] -= dx * step();             // Backward move
        if(p->e.e[1] <= 0) {                  // Remove particle
          p = p->prev;                        // outside cleft
          p->Remove();
          j--;
        }

      } else {

        p->e.e[1] += dx * step();             // Backward move
        if(p->e.e[1] >= d[1]) {               // Remove particle
          p = p->prev;                        // outside cleft
          p->Remove();
          j--;
        }
      }
    }
  }
}
```

```
  return particles;
}
```

3.5 Metropolis

I made a Metropolis algorithm that finds the equilibrium state of 1D diffusion problem.

Metropolis

```
template<typename T> void Metropolis(T T1, T& w1, T T2, T& w2) {
  T a1 = T1 * w1;
  T a2 = T2 * w2;
  T A;

  if(a1 <= a2)          // Metropolis algorithm
    A = 1;
  else
  A = a2/a1;

  w1 += A*a2 - A*a1;   // Master equation
  w2 += A*a1 - A*a2;
}
```

The implementation of the Metropolis algorithm that I used was a forward and backward loop over the lattice elements which is done N times.

Diffusion1D_Metropolis

```
template<typename T> T* Diffusion1D_Metropolis(unsigned long N, unsigned
    int n) {

  T val;
  auto s = Space<T,1>::Allocate(&n);
  s[0] = 1;

  unsigned int j = 0;
  for(unsigned long i = 0; i < N; i++) {

    for(j = 1; j < n; j++)
      Metropolis(0.5, s[j-1], 0.5, s[j]);    // Forward move
    s[0] = 1;
    s[n-1] = 0;

    for(j = n-1; j; j--)
      Metropolis(0.5, s[j], 0.5, s[j-1]);    // Backward move
    s[0] = 1;
    s[n-1] = 0;
  }
  return s;
}
```

3.6 Explicit 2D solver

I made a explicit solver of the 2D diffusion as described in (17).

Diffusion2D_Explicit

```
template<typename T> T** Diffusion2D_Explicit(T alpha, T t, T d[2], T w[2],
    unsigned int n[2]) {

  Vector<unsigned int,2> m = Diffusion2D_Source(d[1], w, n[1]);
  T** u = Diffusion2D_Initialize<T>(m, n);
  T** v = Diffusion2D_Initialize<T>(m, n);
  T** z;
  Vector<T,2> dx2 = Diffusion2D_deltaX2(d, n);
  T dt = Diffusion2D_deltaT<T>(alpha, dx2);        // Calculate time step
  Vector<T,2> a = Diffusion2D_alpha<T>(dt, dx2);   // Alpha values
  unsigned int nt = t/dt;

  n[0]--;
  n[1]--;
  for(int i = 0, j, k; i < nt; i++) {              // Number of time steps

    for(k = 1; k < m.e[0]; k++) {                  // Left wall boundary
      v[0][k] = u[0][k];
      v[0][k] += a.e[1]*(u[0][k+1] - 2.0*u[0][k] + u[0][k-1]);
    }
    for(k = m.e[1]+1; k < n[1]; k++) {             // Right wall boundary
      v[0][k] = u[0][k];
      v[0][k] += a.e[1]*(u[0][k+1] - 2.0*u[0][k] + u[0][k-1]);
    }

    for(j = 1; j < n[0]; j++) {                    // Inner points
      for(k = 1; k < n[1]; k++) {
        v[j][k] = u[j][k];
        v[j][k] += a.e[0]*(u[j+1][k] - 2.0*u[j][k] + u[j-1][k]);
        v[j][k] += a.e[1]*(u[j][k+1] - 2.0*u[j][k] + u[j][k-1]);
      }
    }
    z = u;                              // Prepare for next time step
    u = v;
    v = z;
  }
  n[0]++;
  n[1]++;
  Space<T,2>::Deallocate(v, n);

  return u;
}
```

3.7 Jacobi 2D solver

This is the theta implicit scheme for solving a 2D diffusion problem.

Diffusion2D_Jacobi

```
template<typename T> T** Diffusion2D_Jacobi(T theta, T alpha, T t,T d[2], T
    w[2], unsigned int n[2]) {

  Vector<unsigned int,2> m = Diffusion2D_Source(d[1], w, n[1]);
  T** u = Diffusion2D_Initialize<T>(m, n);
```

```
T** v = Diffusion2D_Initialize<T>(m, n);
auto c = Space<T,2>::Allocate(n);
T** z;
Vector<T,2> dx2 = Diffusion2D_deltaX2(d, n);
T dt = Diffusion2D_deltaT<T>(alpha, dx2);        // Calculate time step
Vector<T,2> a = Diffusion2D_alpha<T>(dt, dx2);   // Alpha values
unsigned int nt = t/dt;

n[0]--;
n[1]--;

T diff;
T c0 = theta * a.e[1];                           // Precalculate constants
T c1 = 1 + 2 * c0;
T c2 = theta * a.e[0];
T c3 = 1 + 2 * (c0 + c2);
T c4 = a.e[1] - c0;
T c5 = 1 - 2 * c4;
T c6 = a.e[0] - c2;
T c7 = (c5 - 2 * c6) / c3;
T c8 = c6 / c3;
T c9 = c4 / c3;
c5 = c5 / c1;
c6 = c4 / c1;
c1 = c0 / c1;
c4 = c0 / c3;
c3 = c2 / c3;

for(int i = 0, j, k; i < nt; i++) {      // Number of time steps

  for(k = 1; k < m.e[0]; k++)            // Calculate coefficients
    c[0][k] = c5 * u[0][k] + c6 * (u[0][k+1] + u[0][k-1]);
  for(k = m.e[1]+1; k < n[1]; k++)
    c[0][k] = c5 * u[0][k] + c6 * (u[0][k+1] + u[0][k-1]);
  for(j = 1; j < n[0]; j++) {
    for(k = 1; k < n[1]; k++)
      c[j][k] = c7 * u[j][k] + c8 * (u[j+1][k] + u[j-1][k]) + c9 * (u[j][
          k+1] + u[j][k-1]);
  }

  do {                                   // Iterate for each time step
    diff = 0;

    for(k = 1; k < m.e[0]; k++) {        // Left wall boundary
      v[0][k] = c[0][k] + c1 * (u[0][k+1] + u[0][k-1]);
      diff += abs(v[0][k] - u[0][k]);
    }
    for(k = m.e[1]+1; k < n[1]; k++) {   // Right wall boundary
      v[0][k] = c[0][k] + c1 * (u[0][k+1] + u[0][k-1]);
      diff += abs(v[0][k] - u[0][k]);
    }

    for(j = 1; j < n[0]; j++) {          // Inner points
      for(k = 1; k < n[1]; k++) {
        v[j][k] = c[j][k] + c3 * (u[j+1][k] + u[j-1][k]) + c4 * (u[j][k
```

```
                +1] + u[j][k-1]);
            diff += abs(v[j][k] - u[j][k]);
          }
        }

        diff /= n[0] * n[1];

        z = u;                                  // Prepare for next iteration
        u = v;
        v = z;

      } while(diff > 0.00001);                  // Converged?
    }

    n[0]++;
    n[1]++;
    Space<T,2>::Deallocate(v, n);
    Space<T,2>::Deallocate(c, n);

    return u;
}
```

4 Results

Order of relative error of v to u is calculated by

$$\epsilon = \log_{10} \left| \frac{v-u}{u} \right| .$$

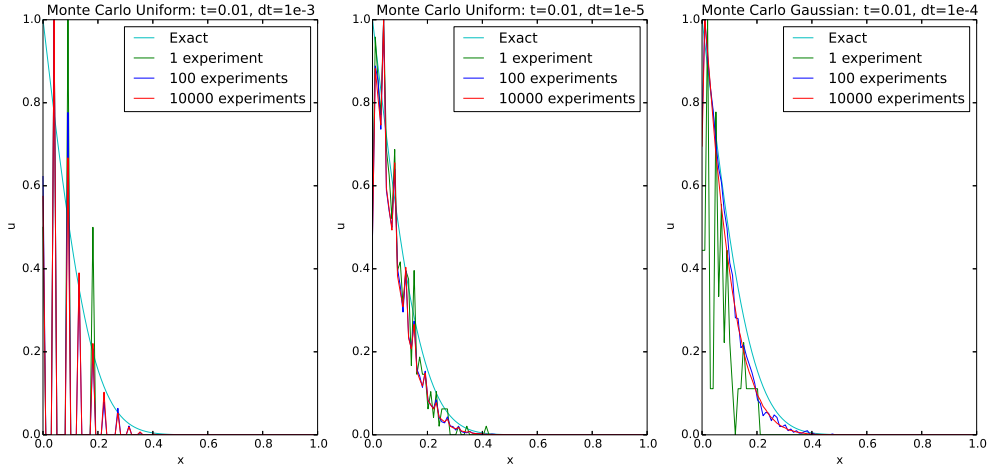

Figure 4.1: *Monte Carlo simulations for 1D diffusion problem, plotted in 100 lattice. Uniform step is of size* Δx (46)*, and Gaussian step is of size* Δx_ℓ (47) *where* $\ell = -\log x$ *and* x *is a uniform random number between 0 and 1, this correspond to mean value 0 and* $1/\sqrt{2}$ *standard divination in the Gaussian distribution. 1 Monte Carlo experiment has only 1 seeding particle. Uniform distribution requires smaller time step than the Gaussian distribution to give a good approximation. The problem with uniform distribution is that it leaves grid points empty which should not be empty when the plot grid is smaller than the step size of the simulation. This is not a problem for Gaussian distribution.*

N	MCU 1e-3	MCU 1e-5	MCG 1e-4
1	2.1e-05	0.040336	0.000868
100	0.000533	3.59375	0.08213
10000	0.052626	352.045	8.20021

Table 4.1: *Calculation time for the Monte Carlo 1D in seconds for* $t = 0.01$. *MCU = Monte Carlo Uniform, MCG = Monte Carlo Gaussian.*

N	MCU 1e-3	MCU 1e-5	MCG 1e-4
1	1.59379	0.590181	0.957192
100	0	-0.203279	-0.20342
10000	0	-0.223313	-0.222473

Table 4.2: *Order of relative error for the Monte Carlo 1D at* $t = 0.01$. *MCU = Monte Carlo Uniform, MCG = Monte Carlo Gaussian. Relative error of calculated and exact value less than* 10^{-2} *are excluded.*

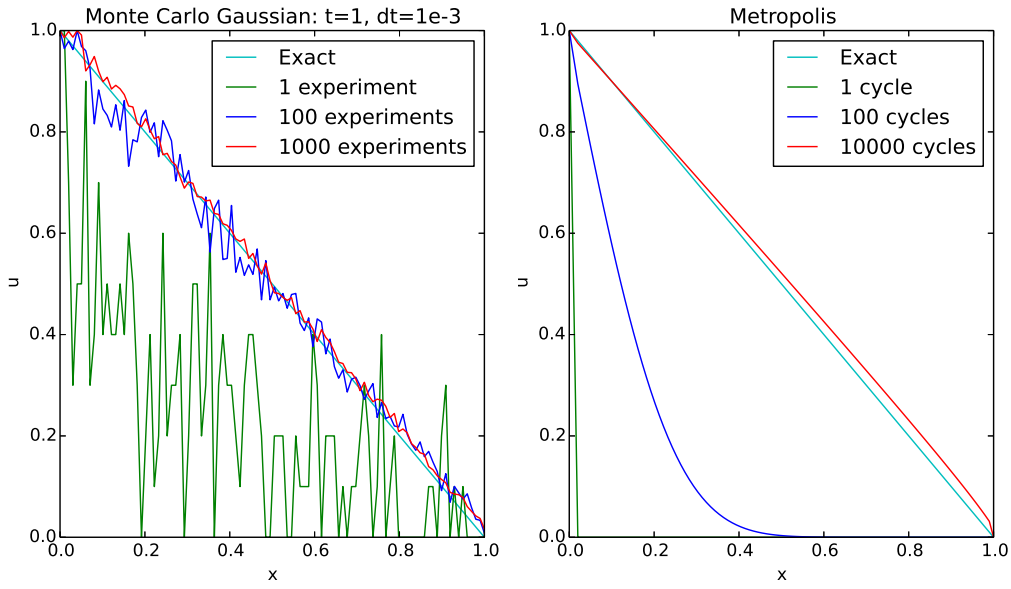

Figure 4.2: *Monte Carlo simulations and Metropolis simulation for 1D diffusion problem at equilibrium, plotted in 100 lattice. Gaussian step is of size* Δx_ℓ (47) *where* $\ell = -\log x$ *and* x *is a uniform random number between 0 and 1, this correspond to mean value 0 and* $1/\sqrt{2}$ *standard divination in the Gaussian distribution. 1 Monte Carlo experiment has only 1 seeding particle. 1 Metropolis cycle is a loop forward and backward on all the grid points.*

N	MCG 1e-3	N	Metropolis
1	0.058192	1	3e-06
100	5.06813	100	0.0002
1000	50.3166	10000	0.022167

Table 4.3: *Calculation time for 1D diffusion in seconds for* $t = 1$. *MCG = Monte Carlo Gaussian.*

N	MCG 1e-3	N	Metropolis
1	0.361791	1	0
100	-0.37639	100	-1.62978e-09
1000	-0.740775	10000	-0.440437

Table 4.4: *Order of relative error for 1D diffusion at* $t = 1$. *MCG = Monte Carlo Gaussian. Relative error of calculated and exact value less than* 10^{-1} *are excluded.*

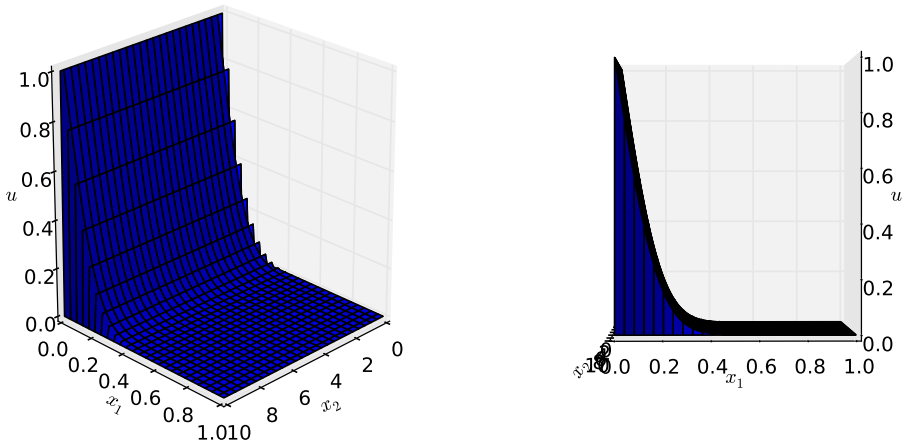

Figure 4.3: *Exact solution at* $t = 0.01$ *of 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$. *Which corresponds well with 1D dimensional case in Figure 4.1.*

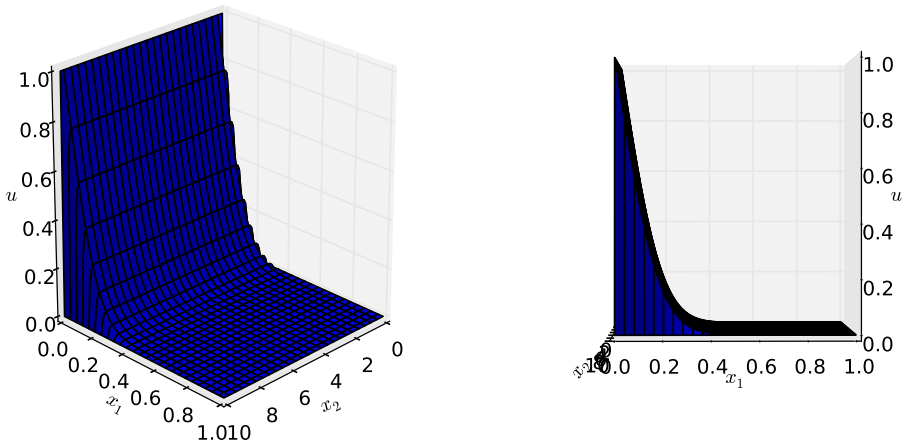

Figure 4.4: *Explicit solution in 100x100 lattice at* $t = 0.01$ *with* $\alpha = 0.49$ *of 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$.

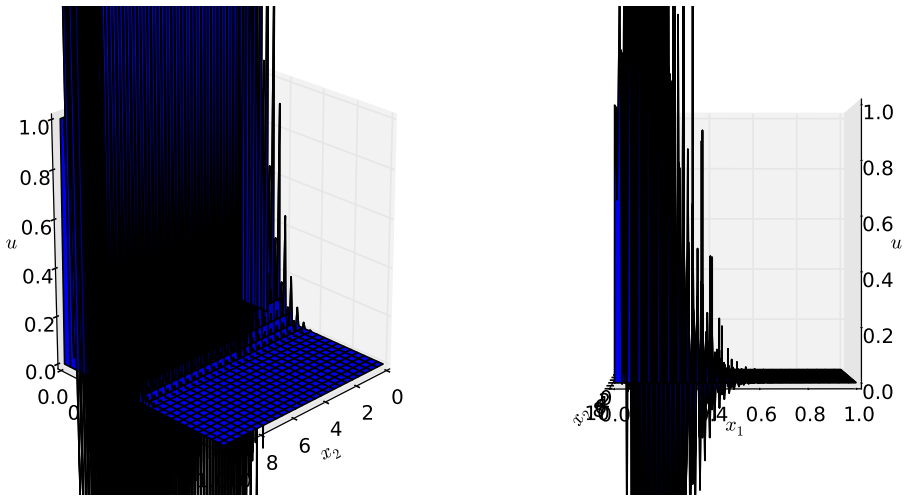

Figure 4.5: *Explicit solution in 100x100 lattice at* $t = 0.01$ *with* $\alpha = 0.51$ *of 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$*. Instability in the solution because of the convergence criteria is not meet.*

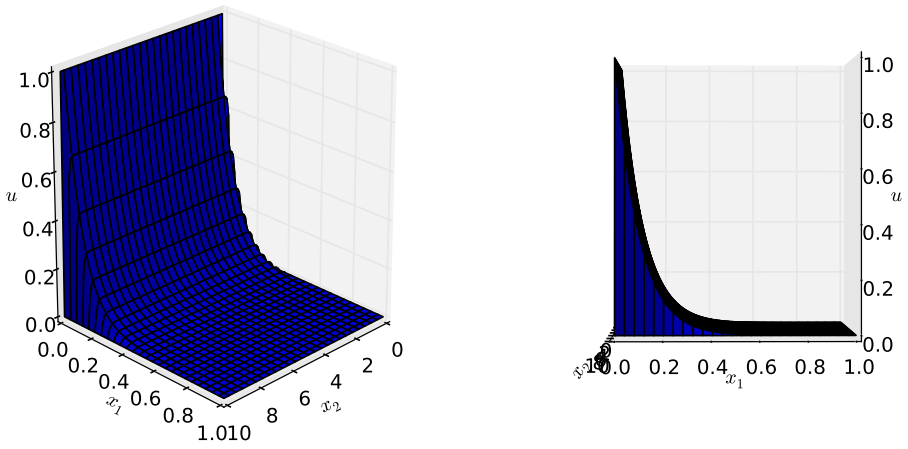

Figure 4.6: *Jacobi solution in 100x100 lattice at* $t = 0.01$ *with* $\theta = 1$ *and* $\alpha = 100$ *of 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$*.*

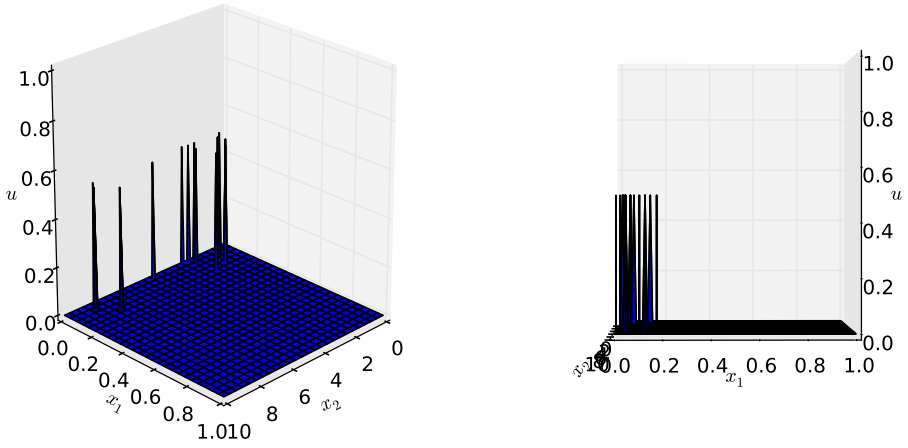

Figure 4.7: *1 Monte Carlo Gaussian experiment in 100x100 lattice at* $t = 0.01$ *with time step* 10^{-4} *for 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$. *Gaussian step is of size* Δx_ℓ (47) *where* $\ell = -\log x$ *and* x *is a uniform random number between 0 and 1, this correspond to mean value 0 and* $1/\sqrt{2}$ *standard divination in the Gaussian distribution. 1 Monte Carlo experiment has only 1 seeding particle.*

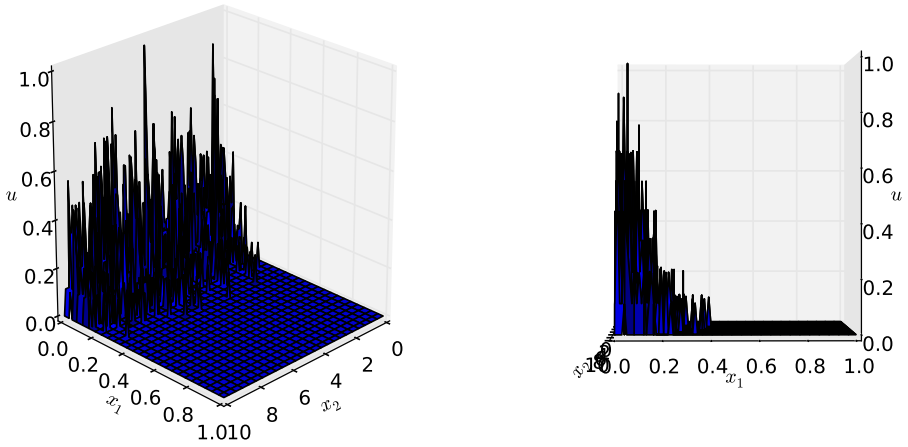

Figure 4.8: *100 Monte Carlo Gaussian experiments in 100x100 lattice at* $t = 0.01$ *with time step* 10^{-4} *for 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$. *Gaussian step is of size* Δx_ℓ (47) *where* $\ell = -\log x$ *and* x *is a uniform random number between 0 and 1, this correspond to mean value 0 and* $1/\sqrt{2}$ *standard divination in the Gaussian distribution. 1 Monte Carlo experiment has only 1 seeding particle.*

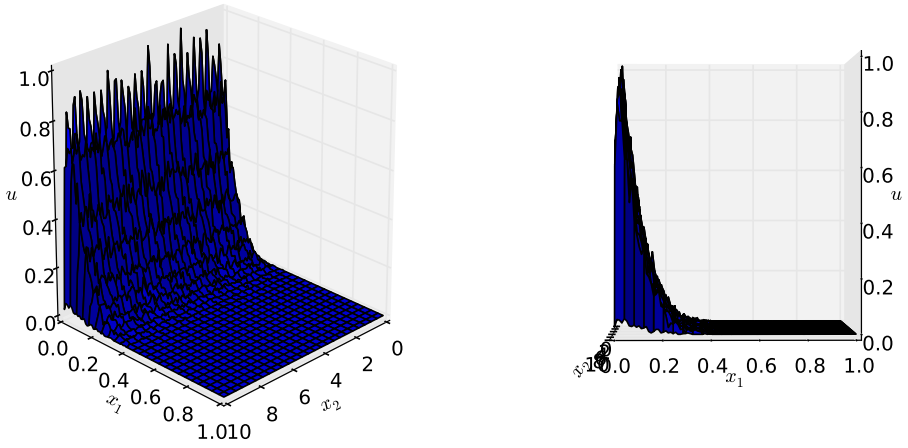

Figure 4.9: *10000 Monte Carlo Gaussian experiments in 100x100 lattice at* $t = 0.01$ *with time step* 10^{-4} *for 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$. *Gaussian step is of size* Δx_ℓ (47) *where* $\ell = -\log x$ *and* x *is a uniform random number between 0 and 1, this correspond to mean value 0 and* $1/\sqrt{2}$ *standard divination in the Gaussian distribution. 1 Monte Carlo experiment has only 1 seeding particle.*

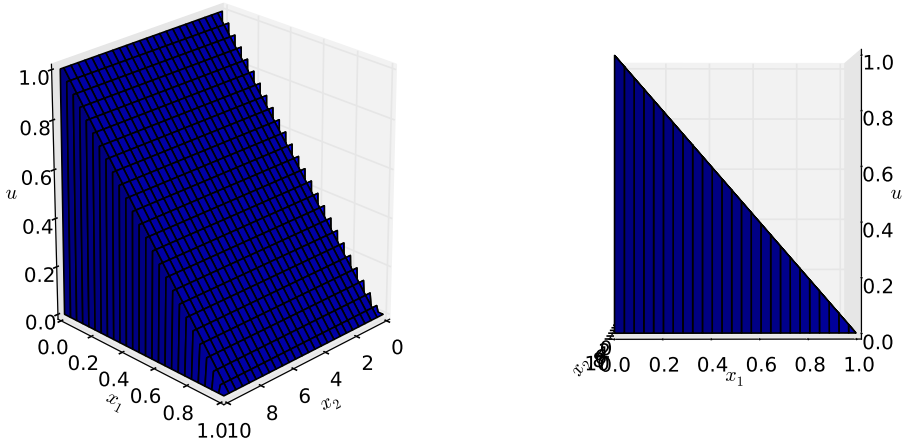

Figure 4.10: *Exact solution at* $t = 1$ *of 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$. *Which corresponds well with 1D dimensional case in Figure 4.2.*

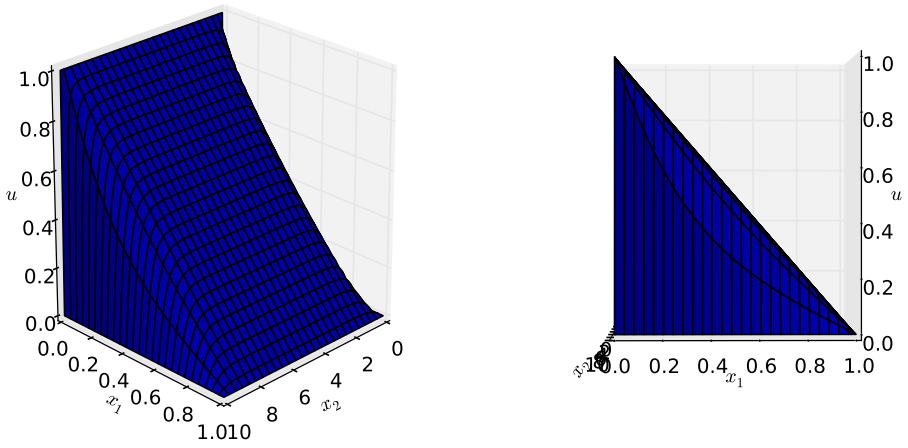

Figure 4.11: *Explicit solution in 100x100 lattice at* $t = 1$ *with* $\alpha = 0.49$ *of 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$.

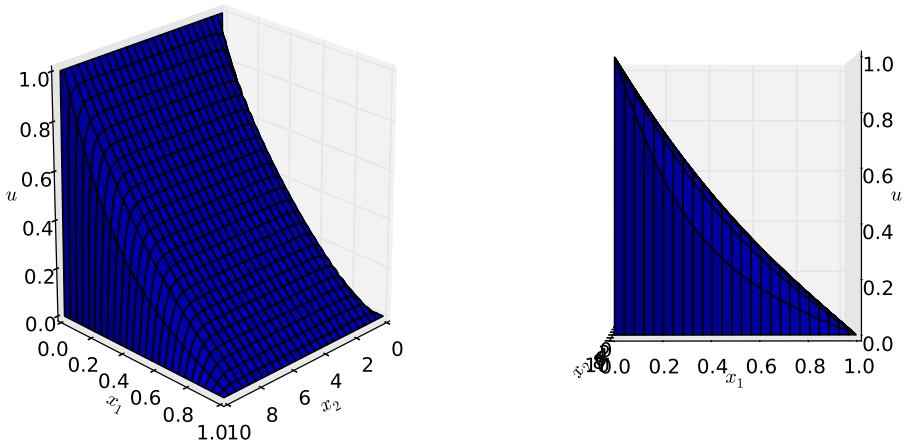

Figure 4.12: *Jacobi solution in 100x100 lattice at* $t = 1$ *with* $\theta = 1$ *and* $\alpha = 10000$ *of 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$.

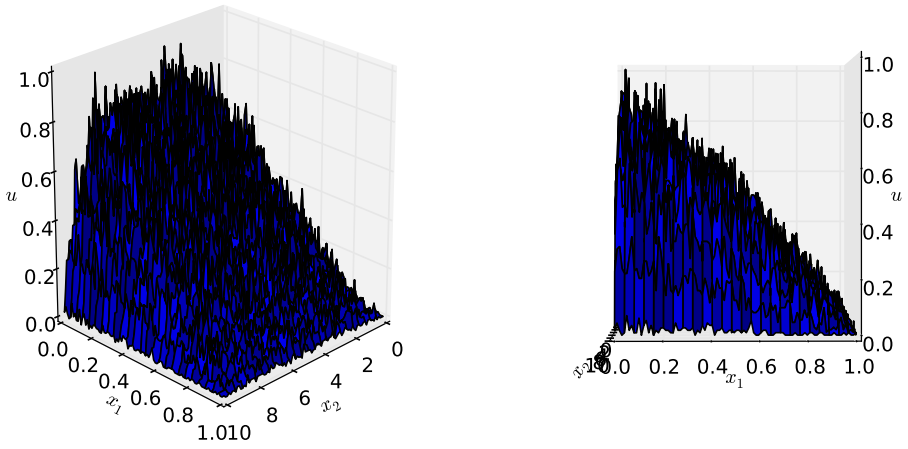

Figure 4.13: *10000 Monte Carlo Gaussian experiments in 100x100 lattice at* $t = 1$ *with time step* 10^{-3} *for 2D diffusion with source boundary of neurotransmitters* $0.1 \leq x_2 \leq 9.9$. *Gaussian step is of size* Δx_ℓ (47) *where* $\ell = -\log x$ *and* x *is a uniform random number between 0 and 1, this correspond to mean value 0 and* $1/\sqrt{2}$ *standard divination in the Gaussian distribution. 1 Monte Carlo experiment has only 1 seeding particle.*

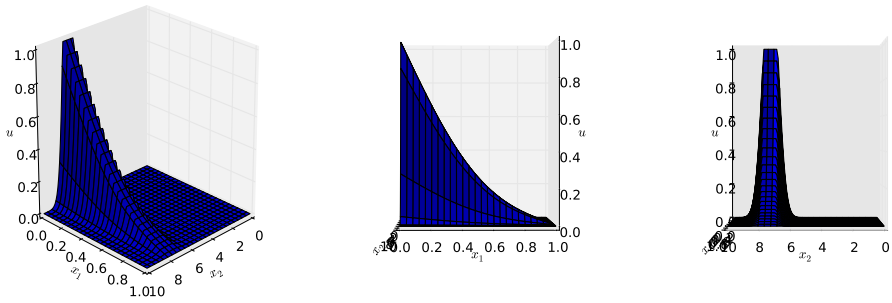

Figure 4.14: *Exact solution at* $t = 0.1$ *of 2D diffusion with source boundary of neurotransmitters* $7 \leq x_2 \leq 8$.

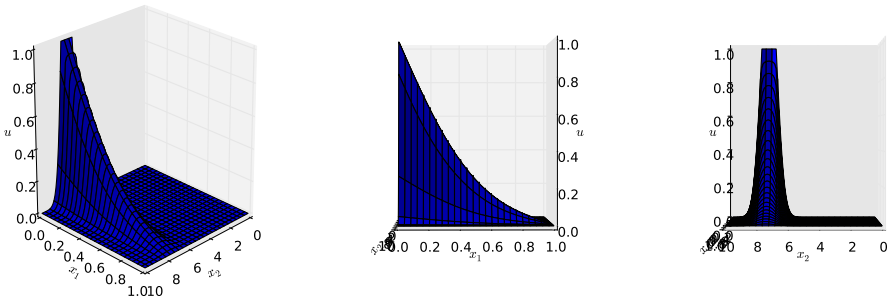

Figure 4.15: *Explicit solution in 100x100 lattice at* $t = 0.1$ *with* $\alpha = 0.49$ *of 2D diffusion with source boundary of neurotransmitters* $7 \leq x_2 \leq 8$.

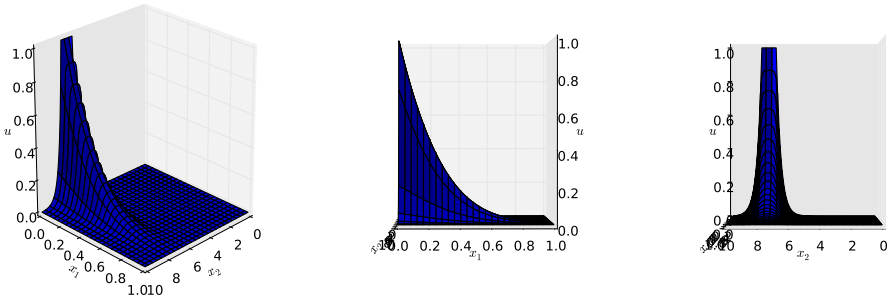

Figure 4.16: *Jacobi solution in 100x100 lattice at* $t = 0.1$ *with* $\theta = 1$ *and* $\alpha = 1000$ *of 2D diffusion with source boundary of neurotransmitters* $7 \leq x_2 \leq 8$.

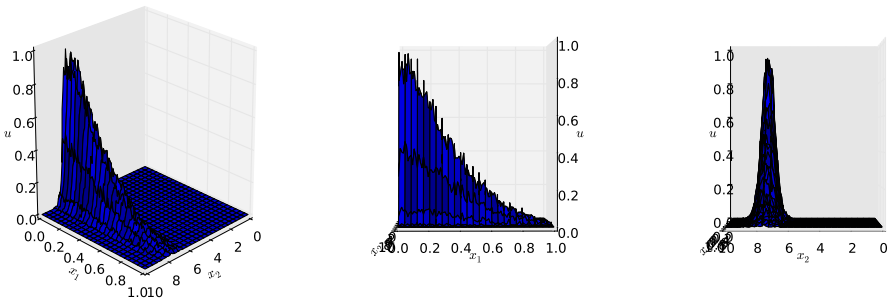

Figure 4.17: *10000 Monte Carlo Gaussian experiments in 100x100 lattice at* $t = 0.1$ *with time step* 10^{-3} *for 2D diffusion with source boundary of neurotransmitters* $7 \leq x_2 \leq 8$. *Gaussian step is of size* Δx_ℓ (47) *where* $\ell = -\log x$ *and* x *is a uniform random number between 0 and 1, this correspond to mean value 0 and* $1/\sqrt{2}$ *standard divination in the Gaussian distribution. 1 Monte Carlo experiment has only 1 seeding particle.*

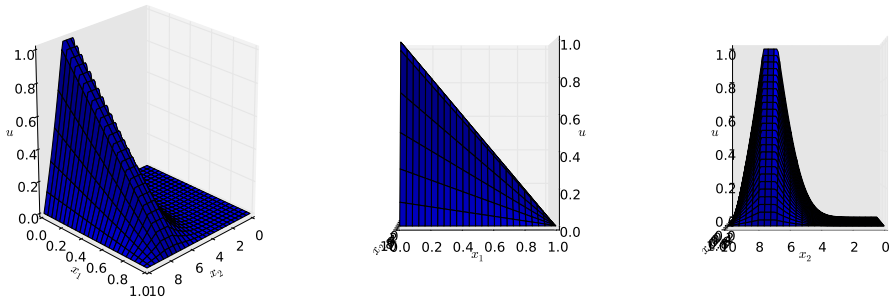

Figure 4.18: *Exact solution at* $t = 1$ *of 2D diffusion with source boundary of neurotransmitters* $7 \leq x_2 \leq 8$.

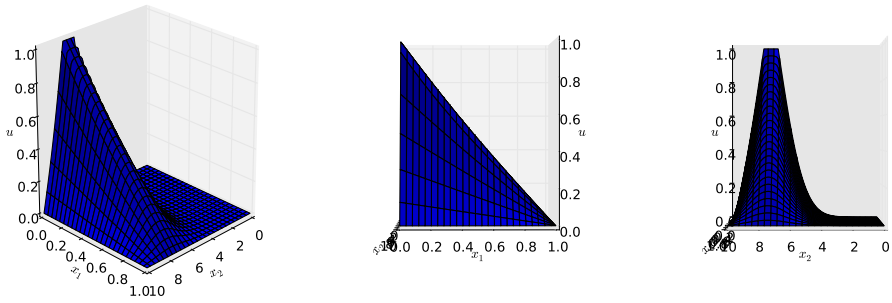

Figure 4.19: *Explicit solution in 100x100 lattice at* $t = 1$ *with* $\alpha = 0.49$ *of 2D diffusion with source boundary of neurotransmitters* $7 \leq x_2 \leq 8$.

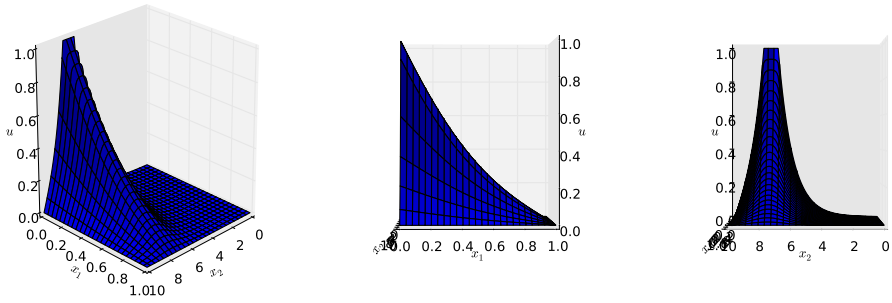

Figure 4.20: *Jacobi solution in 100x100 lattice at* $t = 1$ *with* $\theta = 1$ *and* $\alpha = 10000$ *of 2D diffusion with source boundary of neurotransmitters* $7 \leq x_2 \leq 8$.

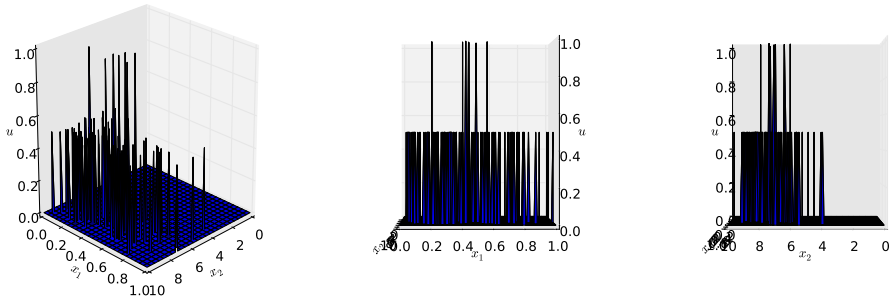

Figure 4.21: *100 Monte Carlo Gaussian experiments in 100x100 lattice at* $t = 1$ *with time step* 10^{-3} *for 2D diffusion with source boundary of neurotransmitters* $7 \leq x_2 \leq 8$*. Gaussian step is of size* Δx_ℓ (47) *where* $\ell = -\log x$ *and* x *is a uniform random number between 0 and 1, this correspond to mean value 0 and* $1/\sqrt{2}$ *standard divination in the Gaussian distribution. 1 Monte Carlo experiment has only 1 seeding particle.*

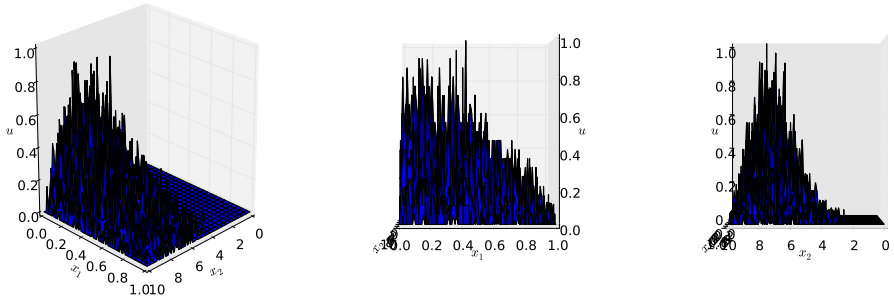

Figure 4.22: *10000 Monte Carlo Gaussian experiments in 100x100 lattice at* $t = 1$ *with time step* 10^{-3} *for 2D diffusion with source boundary of neurotransmitters* $7 \leq x_2 \leq 8$*. Gaussian step is of size* Δx_ℓ (47) *where* $\ell = -\log x$ *and* x *is a uniform random number between 0 and 1, this correspond to mean value 0 and* $1/\sqrt{2}$ *standard divination in the Gaussian distribution. 1 Monte Carlo experiment has only 1 seeding particle.*

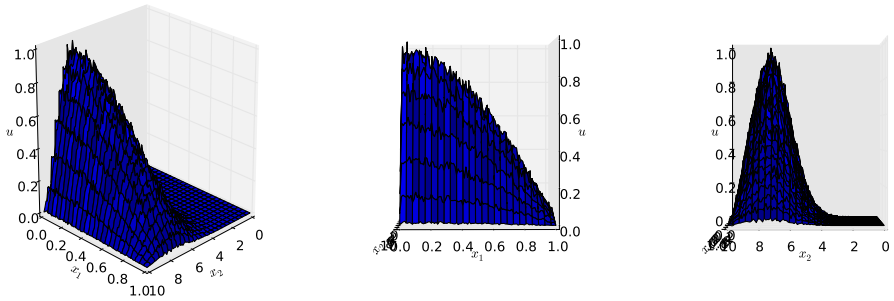

Figure 4.23: *1000000 Monte Carlo Gaussian experiments in 100x100 lattice at* $t = 1$ *with time step* 10^{-3} *for 2D diffusion with source boundary of neurotransmitters* $7 \leq x_2 \leq 8$*. Gaussian step is of size* Δx_ℓ (47) *where* $\ell = -\log x$ *and* x *is a uniform random number between 0 and 1, this correspond to mean value 0 and* $1/\sqrt{2}$ *standard divination in the Gaussian distribution. 1 Monte Carlo experiment has only 1 seeding particle.*

Case	Explicit	Jacobi	Monte Carlo 10000
$t = 0.01, 0.1 \leq x_2 \leq 9.9$	0.023558	0.063603	5.70168
$t = 1, 0.1 \leq x_2 \leq 9.9$	2.26775	0.555716	166.821
$t = 0.1, 7 \leq x_2 \leq 8$	0.226618	0.132611	3.36406
$t = 1, 7 \leq x_2 \leq 8$	2.26925	0.337927	26.0108

Table 4.5: *Calculation time for 2D diffusion in seconds.*

Case	Explicit	Jacobi	Monte Carlo 10000
$t = 0.01, 0.1 \leq x_2 \leq 9.9$	-0.442654	-0.292427	-0.071021
$t = 1, 0.1 \leq x_2 \leq 9.9$	-0.0751966	-0.0602731	0.982644
$t = 0.1, 7 \leq x_2 \leq 8$	-0.540811	-0.0942849	-0.0344229
$t = 1, 7 \leq x_2 \leq 8$	-0.859688	-0.250161	1.02781

Table 4.6: *Order of relative error for 2D diffusion Relative error of calculated and exact value less than* 10^{-1} *are excluded.*

5 Conclusion

A 2D diffusion has more complexity to consider when solving it than a 1D, especially when it comes to the boundary conditions. But by studying the problem with different methods builds confidence that the considerations made are reasonable. I solved the 2 dimensional heat equation with wall boundary analytically and I verified the assumptions that I made was reasonable with alternative derivation with Markov chains. The advantage of Markov chains is that it's easier to set up the physical model without setting up a mathematical model to solve it.

The results show good agreement with each other, however there are difference in when the different methods are most applicable. For the 1D Monte Carlo simulation we clearly see that it's advantages to have a Gaussian step length rather than uniform. This is because the uniform step length have the problem with gaps in plots due to the lattice resolution, which is amended by the Gaussian distribution on the step length which enables the particles to more smoothly distribute out in space and fill the gaps in the plot. Because of the stochastic nature of the Monte Carlo methods results in more noise in the results than other schemes for solving partial differential equation. And we see from project 4 that the Monte Carlo simulations is slower and less accurate for solving a 1D case. The advantage of the Monte Carlo method is that you can do few experiments and still get an good idea of the shape of the result, even though the result has a lot of noise. This means that one can get an good idea of the solution with less computation time. Another advantage is that Monte Carlo methods may be easier to model a physical process, and can help in finding a analytical solution to the problem. And also seems that Monte Carlo methods requires less resources to increase dimensionality of the problem than lattice solvers. This is probably due to that particles does not have to align up in space like a lattice, and can therefore use fewer extra computation points to get a good enough resolution.

The best numerical method to solve the 2D diffusion problem is by far the Jacobi iterative method. This is because Jacobi method is numerical stable for all time step sizes, which makes it's much faster than the explicit scheme for longer end times of the results, since the explicit scheme is dependent of small enough time step to converge.

6 Attachments

The files produced in working with this project can be found at
https://github.com/Eimund/UiO/tree/master/FYS4150/Project%205

The source files developed are

1. Array.h
2. Delegate.h
3. Experiment.h
4. Type.h
5. project5.cpp
6. plot.py

7 Resources

1. QT Creator 5.3.1 with C11
2. Eclipse Standard/SDK - Version: Luna Release (4.4.0) with PyDev for Python
3. Ubuntu 14.04.1 LTS
4. ThinkPad W540 P/N: 20BG0042MN with 32 GB RAM

References

[1] Morten Hjorth-Jensen, *FYS4150 - Project 5* - *Diffusion in two dimensions*, University of Oslo, 2014

[2] Morten Hjorth-Jensen, *Computational Physics - Lecture Notes Fall 2014*, University of Oslo, 2014

[3] Eimund Smestad, *FYS4150 - Computational Physics - Project 4*, University of Oslo, 2014

[4] Farnell and Gibson, *Monte Carlo simulation of diffusion in a spatially nonhomogeneous medium: A biased random walk on an asymmetrical lattice*, Journal of Computational Physics Vol. 208 p. 253-265, 2005

[5] http://en.wikipedia.org/wiki/Diffusion_equation

[6] http://en.wikipedia.org/wiki/Heat_equation

[7] http://en.wikipedia.org/wiki/Dirichlet_boundary_condition

[8] http://en.wikipedia.org/wiki/Poisson%27s_equation

[9] http://en.wikipedia.org/wiki/Uniqueness_theorem_for_Poisson%27s_equation

[10] http://en.wikipedia.org/wiki/Ansatz

[11] http://en.wikipedia.org/wiki/Kronecker_delta

[12] http://en.wikipedia.org/wiki/Jacobi_method

[13] http://en.wikipedia.org/wiki/Stochastic_process

[14] http://en.wikipedia.org/wiki/Independence_%28probability_theory%29

[15] http://en.wikipedia.org/wiki/Markov_property

[16] http://en.wikipedia.org/wiki/Markov_process

[17] http://en.wikipedia.org/wiki/Markov_chain

[18] http://en.wikipedia.org/wiki/Master_equation

[19] http://en.wikipedia.org/wiki/Metropolis%E2%80%93Hastings_algorithm

[20] http://en.wikipedia.org/wiki/Monte_Carlo_algorithm

[21] http://en.wikipedia.org/wiki/Isotropy

[22] http://en.wikipedia.org/wiki/Lax_equivalence_theorem

[23] http://en.wikipedia.org/wiki/Well-posed_problem

[24] http://en.wikipedia.org/wiki/Spectral_radius