MULTISCALE MODELING OF DIFFUSION PROCESSES IN THE BRAIN

by

Fredrik E Pettersen
f.e.pettersen@fys.uio.no

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

June 2014

Contents

List of Figures

Chapter 1

Introduction

This thesis is an attempt at modeling diffusion processes in which part of the process takes place on a length scale so small that the continuum approximation becomes invalid. In this part we will therefore try and introduce another model of the diffusion process in the hope that this will give us something extra. It is the hope of Hans Petter that this thesis will be an introduction to a new research project to understand mesoscale physics in collaboration with Gaute Einevoll at UMB.

The very first approach was to simply try the problem on a bit. That is to try and substitute some small part of the mesh in a Finite Difference Diffusion solver (Froward Euler scheme) with a stochastic diffusion solver. A random walk method was implemented on part of the mesh to take over the equation-solving. This was done in 1 and 2 spatial dimensions with the aim of finding potential difficulties so that we can further investigate them.
Upon switching length-scales a fundamental question arises almost immediately; what is the continuum limit? In our case this question takes a slightly different, and possibly more answerable form; what is the conversion rate between the continuum model and the microscopic model, and by extension, what does a walker correspond to? The first instinct of this candidate was to just try some conversion rate (say some value corresponds to some number of walkers), and this was implemented in both 1 and 2 dimensions.

1.1 Background

This is very much a first draft, and only my thoughts around what I understand as the basis of the thesis.
Though the scope of this thesis might seem a bit *sought*, it is in fact a real world concern from the computational neuroscience group at UMB. Their work contains network simulations of neurons, which they are trying to tweak with experimental data. The experimental data come from measurements of voltage levels in the Extra Cellular Space (ECS).

The ECS is, essentially, the space which separates neurons and has an immensely complicated geometry as we can see from figure **??**. The width of the ECS varies, but is in general $\sim 1\mu$m. Parts of the ECS is, however, much narrower. In the ECS there are a number of diffusion governed transport mechanisms which are vital for the function of the neurons. In fact, there are examples of snake venom which have a shrinking effect on the ECS, and it is thought that this causes large scale neuronal death very quickly. For the researchers at UMB, however, the ECS is of importance because the diffusion tensor in the ECS is related to the conductivity tensor through equation 1.4, which in turn lets them teak parameters in their network models.

$$D = \frac{k_B T}{6\pi\eta r}, \tag{1.1}$$

Though there are several reasons to study diffusion processes in the ECS, this project has a specific goal in mind. The Einstein relation, eq. (1.1), relates the diffusion constant to the viscosity of the medium in which the diffusion is taking place. From the definition of viscosity, μ, we have

$$v_d = \mu F \tag{1.2}$$

where $F = qE$ is the standard electrical force acting on a charged particle. We can also define the current from the drift velocity of the particles as

$$J = cqv_d = \sigma E \tag{1.3}$$

where $\sigma = c\mu q^2$ is the electrical conductance, in this case, of the ECS. Inserting this in the Einstein relation, eq. (1.1), lets us express the conductivity in terms of the diffusion constant as

$$\sigma = \frac{cq}{k_B T} D \tag{1.4}$$

Equation (1.4) is trivially generalizable to tensor notation, where the conductance and diffusion "constant" are both 2. order tensors.

$$\langle r^2 \rangle = 2dDt \tag{1.5}$$

The fundamental problem here is that, while diffusion in its own is a truly multi-scale process, we have no way of knowing for sure that the continuum models, and all they bring with them, are correct for this type of geometry. In particular, the Einstein relation 1.1 has, to my knowledge, only been derived for diffusion in a homogeneous media. The definition of a homogeneous media includes a mean free path close to infinity, which we will have trouble arguing for the existence of in the ECS. On the other hand, I know that another Einstein relation 1.5 is widely used in other fields of physics where the media in question is hardly homogeneous. In molecular dynamics simulations, the two mentioned Einstein relations are used to measure the viscosity of fluids in nano-porous materials, and with success as far as I know.

1.2 A short introduction to mathematical neuroscience

Neuroscience is the scientific study of the nervous system, but in the traditional sense it focuses very much on what parts of the brain are responsible

for what. Mathematical neruroscience is more focused on the physics and chemistry involved in the different parts of the brain and nervous system. An example of the power of this approach is the classical work done by Hodgkin and Huxley in 1952, earning them the Nobel price in Physiology or medicine in 1963. Through four non-linear coupled differential equations they were able to predict the propagation speed of signals along the squid giant axon to a quite high precision.

The human brain consists of two types of cells; the neurons and neuroglia. Neurons are tasked with signal processing and transport, while the glia are thought to have more janitorial tasks. The neurons are bathed in a salt solution that is mainly Na^+ and Cl^-. Inside the neurons, a highly regulated salt solution of mainly K^+ sets up a potential difference relative to the outside of the cell of approximately -65mV. The neurons are in constant communication with each other through action potentials, which are disturbances in the membrane potentials of neurons. These action potentials are generated in the body of the cell, called the soma, from where they propagate down the axon without loss of amplitude. This is achieved by constantly amplifying the signal using ion pumps (see the Hodgin-Huxley model of the action potential [**graham2011principles**]). After propagating down the axon, the action potential reaches a synapse which is a gate to another neuron. If the action potential is of significant strength, vesicles carrying neurotransmitters merge with the synapse membrane, letting the neurotransmitters diffuse to the dendrite of the other neuron. If enough neurotransmitters reach the post-synaptic side, the signal continues propagating to the soma of this neuron, and the entire process starts over again.
The interest of this project lies, mainly, in the diffusion processes that take place in the space between these types of cells, the so-called extracellular space (ECS). This is a narrow space ($\sim 10 - 100$ nm [2]) with a highly complicated geometry (Figure **??**). Surprisingly, the ECS adds up to 20% of the total brain volume. We can understand this by realizing that every part of a cell must be separated from another cell by the ECS. Since the cells consists of axons and dendrites which can be viewed as (somewhat) fractal, we see that this means separating a vast amount of surface area from other surface areas.

The ECS is thought to support the diffusion of oxygen and nutrients to the neurons and glia, and diffusion of carbon dioxide and other waste from these cells through the blood - brain barrier and into the bloodflow.

Chapter 2

Basic Theory

In this chapter we will take a closer look at random walks, both in general and the transition from the statistical view to partial differential equations. We will take a look at different algorithms to produce random walks, and discuss their pros and cons in light of this project. Then we will take a quick look at partial differential equations and numerical solution of them.

2.1 Introduction to random walks

The most basic random walk is a walker on the x-axis which will take a step of a fixed length to the right with a probability p, or to the left with a probability $q = 1 - p$. Using (pseudo-) random numbers on a computer we can simulate the outcomes of a random walk. For each step (of which there are N) we draw a random number, r, between 0 and 1 from some distribution (say a uniform one) which will be the probability. If $r \leq p$ the walker will take a step to the left, otherwise it will take a step to the right. After the N steps the walker will have taken R steps to the right, and $L = N - R$ steps to the left. The net displacement from the origin will be $S = R - L$.
This simple approach is easily generalizable to two and three dimensions by having $2d$ possible outcomes from the random number, where d is the dimensionality. In two dimensions the walker will step up if $r \in (0.75, 1]$ and left if $r \in [0, 0.25)$, for example.

2.1.1 Further discussion and analysis of the introduction

The following derivation is borrowed from a compendium in statistical mechanics by Finn Ravndal.
If we do sufficiently many walks, the net displacement will vary from $S = +N$ to $S = -N$ representing all steps to the right and all steps to the left respectively. The probability of all steps being to the right is $P_N(N) = p^N$. Should one of the steps be to the left, and the rest to the right we will get a net displacement of $S = N - 2$ with the probability $P_N(R = N - 1) = Np^{N-1}q$. We can generalize this to finding the probability of a walk with a R steps to the right as

$$P_N(R) = \binom{N}{R} p^R q^{N-R} \tag{2.1}$$

where $\binom{N}{R} = \frac{N!}{R!(N-R)!}$ is the number of walks which satisfy the net displacement in question, or the multiplicity of this walk in statistical mechanics

terms. Equation 2.1 is the Bernoulli probability distribution, which is normalized.

$$\sum_{R=0}^{N} P_N(R) = (p+q)^N = 1^N = 1$$

We can use this distribution to calculate various average properties of a walk consisting of N steps. For example, the average number of steps to the right is

$$\langle R \rangle = \sum_{R=0}^{N} R P_N(R) = \sum_{R=0}^{N} \binom{N}{R} R p^R q^{N-R} =$$

$$p\frac{d}{dp}\sum_{R=0}^{N}\binom{N}{R}p^R q^{N-R} = p\frac{d}{dp}(p+q)^N = Np(p+q)^{N-1} = Np$$

From this we can also find the average value of the net displacement using $S = R - L = R - (N - R) = 2R - N$.

$$\langle S \rangle = \langle 2R \rangle - N = 2Np - N(p+q) = N(2p - p - q) = N(p-q)$$

We notice that the average net displacement is greatly dependent on the relationship between p and q, and that any symmetric walk will have an expected net displacement of zero. In many cases we will be more interested in the mean square displacement than the displacement itself, because many important large scale parameters can be related to the root-mean-square displacement. This can also be calculated rather straightforwardly.

$$\langle R^2 \rangle = \sum_{R=0}^{N} R^2 P_N(R) = \sum_{R=0}^{N} \binom{N}{R} R^2 p^R q^{N-R} =$$

$$\left(p\frac{d}{dp}\right)^2 \sum_{R=0}^{N}\binom{N}{R}p^R q^{N-R} = \left(p\frac{d}{dp}\right)^2 (p+q)^N$$

$$= Np(p+q)^{N-1} + p^2 N(N-1)(p+q)^{N-2} = (Np)^2 + Np(1-p) = (Np)^2 + Npq$$

Like before, the average net displacement is given as $S^2 = (2R - N)^2$ and we obtain

$$\langle S^2 \rangle = 4\langle R^2 \rangle - 4N\langle R \rangle + N^2 = 4((Np)^2 + Npq) - 4N^2 p + N^2$$
$$= N^2(4p^2 - 4p + 1) + 4Npq = N^2(2p-1)^2 + 4Npq = N^2(p-q)^2 + 4Npq.$$

which for the 1D symmetric walk gives $\langle S^2\rangle = N$ and the variance, denoted $\langle \Delta S^2\rangle = \langle \langle S^2\rangle - \langle S\rangle^2\rangle$, is found by insertion as

$$\langle \Delta S^2\rangle = \langle N^2(p-q)^2 + 4Npq - (N(p-q))^2\rangle = 4Npq \tag{2.2}$$

When the number of steps gets very large we can approximate the Bernoulli distribution (eq. 2.1) by the Gaussian distribution. This is most easily done in the symmetric case where $p = q = \frac{1}{2}$, but it is sufficient for the steplengths to have a finite variance (*find something to refer to*). The Bernoulli distribution then simplifies to

$$P(S,N) = \left(\frac{1}{2}\right)^N \frac{N!}{R!L!} \tag{2.3}$$

on which we apply Stirling's famous formula for large factorials $n! \simeq \sqrt{2\pi n}\cdot n^n e^{-n}$.

$$\begin{aligned} P(S,N) &= \left(\frac{1}{2}\right)^N \frac{N!}{R!L!} \\ &= \exp\left(-N\ln 2 + \ln\sqrt{2\pi N} + N\ln N - \ln\sqrt{2\pi R} - R\ln R - \ln\sqrt{2\pi L} - L\ln L\right) \\ &= \sqrt{\frac{N}{2\pi RL}}\exp\left(-R\ln\frac{2R}{N} - L\ln\frac{2L}{N}\right) \end{aligned}$$

Where we have used $R+L = N$. We now insert for $\frac{2R}{N} = 1+\frac{S}{N}$ and $\frac{2L}{N} = 1-\frac{S}{N}$ and expand the logarithms to first order, $RL = \frac{N^2-S^2}{4}$ in the prefactor, and approximate $1 - \frac{S^2}{N^2} \simeq 1$. This gives

$$P(S,N) = \sqrt{\frac{2}{\pi N}}\exp\left(\frac{-S^2}{2N}\right) \tag{2.4}$$

which is an ordinary, discrete Gaussian distribution with $\langle S\rangle = 0$ and $\langle S^2\rangle = N$. If we keep assuming that the walker is on the x-axis, and let the step length, a, get small the final position will be $x = Sa$ which we can assume is a continuous variable. Similarly, we let the time interval between each step, τ, be small and let the walk run for a continuous time $t = N\tau$. This changes the distribution 2.4 to

$$P(x,t) = \frac{1}{2a}\sqrt{\frac{2\tau}{\pi t}}\exp\left(-\frac{x^2\tau}{2a^2 t}\right). \tag{2.5}$$

The prefactor $\frac{1}{2a}$ is needed to normalize the continuous probability distribution since the separation between each possible final position in walks with

the same number of steps is $\Delta x = 2a$. We also introduce the diffusion constant

$$D = \frac{a^2}{2\tau} \tag{2.6}$$

making the distribution

$$P(x,t) = \sqrt{\frac{1}{4\pi Dt}} \exp\left(-\frac{x^2}{4Dt}\right) \tag{2.7}$$

Introducing x also gives us the expectation value and variance of x on a form which will be useful later.

We have $x = Sa$ which means

$$\langle x \rangle = a\langle S \rangle$$

and

$$\langle x^2 \rangle = a^2\langle S^2 \rangle$$

Finally by insertion we find the variance $\langle \Delta x^2 \rangle$

$$\langle \Delta x^2 \rangle = \langle \langle x^2 \rangle - \langle x \rangle^2 \rangle = \langle a^2\langle S^2 \rangle - a^2\langle S \rangle^2 \rangle = 4Npqa^2 \tag{2.8}$$

2.1.2 More general Random Walks

In the more general case, the position of a random walker, $\mathbf{r}$ at a time t_i is given by the sum

$$\mathbf{r}(t_i) = \sum_{j=0}^{i} \Delta\mathbf{x}(t_j) \tag{2.9}$$

where $\Delta\mathbf{x}(t_j) = (\Delta x(t_j), \Delta y(t_j), \Delta z(t_j))$ in 3D. Each $\Delta x, \Delta y, \Delta z$ is a random number drawn from a distribution with a finite variance $\sigma^2 = \langle \Delta x^2 \rangle$. By the central limit theorem, any stochastic process with a well defined mean and variance can, given enough samples, be approximated by a Gaussian distribution. This means that the probability of finding the walker at some position x after M steps is

$$P(x,M) \propto e^{-\frac{x^2}{2M\sigma^2}} \tag{2.10}$$

Remember that the actual Gaussian distribution is

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{(n-\mu)^2}{2\sigma^2}\right)$$

We can introduce an Einstein relation $\sigma^2 = 2dD\Delta t$ and the obvious relation $t = M\Delta t$ to get a more desirable exponent. We see that $\langle \Delta x^2 \rangle = 2Dt$ in the one dimensional case where $d = 1$.

The introduction of the Einstein relation might put some restrictions on our model. Normalizing the expression gives us

$$P(x,t) = \sqrt{\frac{1}{4Dt}} \exp\left(-\frac{x^2}{4Dt}\right) \tag{2.11}$$

If we have a large number, N, of walkers, their concentration will be $C(x,t) = NP(x,t)$. The concentration is conserved, so any amount that flows out of an area must reflect as a decrease in concentration. We can express this by the flow of concentration

$$\frac{\partial C}{\partial t} - \nabla \cdot \mathbf{J} = S \tag{2.12}$$

where $\mathbf{J}$ is the flow vector and S is a source term which in our case will be zero. From Fick's first law we know that $\mathbf{J} = -D\nabla C$. Inserting this gives us

$$\frac{\partial C}{\partial t} = \nabla \cdot (D \cdot \nabla C) \tag{2.13}$$

which is the diffusion equation. By insertion we can check that this version (2.11) of the Gaussian distribution fulfills the diffusion equation. Starting with only the time derivative gives us

$$\frac{\partial P}{\partial t} = -\frac{4\pi D \exp\left(-\frac{x^2}{4Dt}\right)}{2\sqrt{(4\pi Dt)^3}} + \frac{x^2 \exp\left(-\frac{x^2}{4Dt}\right)}{4Dt^2\sqrt{4\pi Dt}}$$

$$= \exp\left(-\frac{x^2}{4Dt}\right)\left(\frac{8Dx^2}{2\sqrt{\pi}(4Dt)^{5/2}} - \frac{(4D)^2 t}{2\sqrt{\pi}(4Dt)^{5/2}}\right) = \frac{4D\exp\left(-\frac{x^2}{4Dt}\right)(x^2 - 2Dt)}{\sqrt{\pi}(4Dt)^{5/2}}$$

We then finish by doing the spatial derivative

$$D\frac{\partial^2 P}{\partial x^2} = \frac{D}{\sqrt{4\pi Dt}}\frac{\partial}{\partial x}\left[-\exp\left(\frac{-x^2}{4Dt}\right)\left(\frac{-2x}{4Dt}\right)\right]$$

$$= \frac{2D}{4Dt\sqrt{4\pi Dt}}\exp\left(\frac{-x^2}{4Dt}\right)\left[1 - x\left(\frac{2x}{4Dt}\right)\right] = \frac{4D\exp\left(-\frac{x^2}{4Dt}\right)(x^2 - 2Dt)}{\sqrt{\pi}(4Dt)^{5/2}}$$

and see that they are equal, meaning that the diffusion equation is satisfied.

2.1.3 Choosing random walk algorithm

The simplest random walk model, which places walkers on discrete mesh points and uses a fixed step length, has been used with great success to model diffusion processes. Farnell and Gibson discuss this in their article [1]. In this project we will be torn between chosing a realistic algorithm to advance the random walkers, like brownian motion or to go for simplicity. That being said, by the central limit theorem both models will after some timesteps be described by a Gaussian distribution meaning that on the PDE scale we will not know the difference. Hence it will make no sense to not use the simplest random walk model. We should note that it will be quite easy to change the algorithm used for random walks, and so we have not locked ourselves to anything yet.

2.1.4 Random walks and anisotropy

Any real problem where parts of the diffusion process cannot be modelled by the continuum approximation is bound to be anisotropic. There is reason to believe that an anisotropic diffusion process on the PDE level will lead to an anisotropic random walk model as well, but how do we model this. If we simply replace the diffusion constant by a function $D = D(\mathbf{x})$ (see eq 2.45) we are at least started, but this will not quite be sufficient as Farnell and Gibson point out [1]. Through their experiments they found that only adjusting the steplength will not improve the error noticeably and reasoned that this is because the walkers are still as likely to jump in both directions (right or left in 1d), and that the stepsize is the same in both cases, hence the model does not resemble anisotropy. They went on to introduce an adjusted steplength and an adjusted step probability, a solution they landed on after trial and error. The expressions they proposed are listed in equations 2.14 to 2.18.

$$\Delta_p(x) = \frac{1}{2}\left(L(x) + L(x + \Delta_p(x))\right) \to L(x) + \frac{1}{2}L(x)L'(x) \tag{2.14}$$

$$\Delta_m(x) = \frac{1}{2}\left(L(x) + L(x - \Delta_m(x))\right) \to L(x) + \frac{1}{2}L(x)L'(x) \tag{2.15}$$

where $L(x)$ is defined in equation 2.16 and $\Delta_p(x)$ and $\Delta_m(x)$ are the adjusted steplengths to the right and left, respectively.

$$L(x) = \sqrt{2D(x)\Delta t} \tag{2.16}$$

We also have the adjusted jump probabilities $T_r(x)$ and $T_l(x)$ which are the probabilities for a walker at position x to jump right or left, respectively.

These are defined in equations 2.17 and 2.18

$$T_r(x) = \frac{1}{2} + \frac{1}{4}L'(x) \tag{2.17}$$

$$T_l(x) = \frac{1}{2} - \frac{1}{4}L'(x) \tag{2.18}$$

We notice that the adjusted steplength we proposed to start with is still a part of the final expressions.

2.1.5 Random walks and drift

Another point we have yet to say something about is diffusion that has a drift term, $\frac{\partial u}{\partial x}$.

Initially one thought that diffusion in the Extra Cellular Space of the brain was governed by a drift term, but the modern perception is that this drift term is in the very least negligible [2]. Though it is unlikely that we will include a drift term in our model, we will say a few words about it here since it is of importance in other applications and might be a natural extension at some point, should someone else use this work.

We model random walkers with drift by simply adding some vector to the Brownian motion model, thus forcing all walkers to have a tendency to walk a certain direction. This approach can also be used in the fixed steplength (or variable steplength in the anisotropic case) if we express the new step, $\mathbf{s}$, as

$$\mathbf{s} = (\pm l \text{ or } 0, \pm l \text{ or } 0) + \mathbf{d}$$

where $\mathbf{d}$ denotes the drift of the walker.

We can set up the continuity equation for a concentration, $C(x,t) = NP(x,t)$ of random walkers which are affected by a drift.

$$\frac{\partial C}{\partial t} + \nabla \cdot \mathbf{j} = S \tag{2.19}$$

Where $\mathbf{j}$ denotes the total flux of walkers through some enclosed volume and S is a source/sink term. Since the walkers are affected by drift the flux will consist of two terms; $\mathbf{j} = \mathbf{j}_{diff} + \mathbf{j}_{drift}$. From Fick's first law we know that $\mathbf{j}_{diff} = -D\nabla C$. The second flux term is the advective flux which will be equal to the average velocity of the system; $\mathbf{j}_{drift} = \mathbf{v}C$. Inserting this in the continuity equation gives us the well known convection diffusion equation (2.20).

$$\frac{\partial C}{\partial t} = \nabla \cdot (D\nabla C) - \nabla \cdot (\mathbf{v}C) + S \tag{2.20}$$

Which in many cases will simplify to

$$\frac{\partial C}{\partial t} = D\nabla^2 C - \mathbf{v} \cdot \nabla C \tag{2.21}$$

In order to determine all the parameters of the convection diffusion equation 2.20 we will need to go through some of the calculations from chapter 2.1. The situation is the same, a walker in one dimension which can jump left or right, but this time will also move a finite distance d each timestep. This will make the expected net displacement

$$\langle S \rangle = R - L + Nd = N(p - q) + Nd$$

and the expected mean square displacement

$$\langle S^2 \rangle = (2\langle R \rangle - N)^2 + (Nd)^2 = N^2(p - q)^2 + 4Npq + (Nd)^2$$

which in turn gives us the variance

$$\begin{aligned} \langle \Delta S^2 \rangle &= \langle \langle S^2 \rangle - \langle S \rangle^2 \rangle \\ &= N^2(p - q)^2 + 4Npq + (Nd)^2 - N^2(p - q)^2 - (Nd)^2 \\ \langle \Delta S^2 \rangle &= 4Npq \end{aligned}$$

This shows us that the variance is untouched by the drift term, but not the mean which for the symmetric case is $\langle S \rangle = Nd$. When we convert this to the continuous variables x and t we get the solution shown in equation 2.22.

$$C(x, t) = \frac{N}{\sqrt{4\pi Dt}} \exp\left(-\frac{(x - vt)^2}{4Dt}\right) \tag{2.22}$$

Where $v = \frac{d}{\Delta t}$ is the velocity of the concentration and D is the well known diffusion constant, inserted from the Einstein relation $\sigma^2 = 2D\Delta t$.

2.2 Some words about partial differential equations

2.2.1 Discretizing

To maintain a bit of generality we will look at the (potentially) anisotropic diffusion equation in 2d. The extension to 3d is trivial, as is the 1d version.

$$\frac{\partial u}{\partial t} = \nabla D \nabla u + f \tag{2.23}$$

where f is some source term. The final expression and scheme will depend on how we chose to approximate the time derivative, but the spatial derivative will have the same approximation.
We start off by doing the innermost derivative in one dimension. The generalization to more dimensions is trivial, and will consist of adding the same terms for the y and z derivatives.

$$\left[\frac{d}{dx}u\right]^n \approx \frac{u^n_{i+1/2} - u^n_{i-1/2}}{\Delta x}$$

Where we have made the approximate derivative around the point x_i. We then set $\phi(x) = D\frac{du}{dx}$ and do the second derivative

$$\left[\frac{d}{dx}\phi\right]^n \approx \frac{\phi^n_{i+1/2} - \phi^n_{i-1/2}}{\Delta x}$$

and insert for ϕ

$$\frac{\phi^n_{i+1/2} - \phi^n_{i-1/2}}{\Delta x} = \frac{1}{\Delta x^2}\left(D_{i+1/2}(u^n_{i+1} - u^n_{i+1}) - D_{i-1/2}(u^n_i - u^n_{i-1})\right)$$

Since we can only evaluate the diffusion constant at the mesh points (or strictly speaking since it is a lot simpler to do so) we must approximate $D_{i\pm1/2} \approx 0.5(D_{i\pm1} + D_i)$. Inserting this gives us

$$\begin{aligned}\nabla D\nabla u \approx &\frac{1}{2\Delta x^2}\left((D_{i+1,j} + D_{i,j})(u_{i+1,j} - u_{i,j}) - (D_{i,j} + D_{i-1,j})(u_{i,j} - u_{i-1,j})\right)\\ &+\frac{1}{2\Delta y^2}\left((D_{i,j+1} + D_{i,j})(u_{i,j+1} - u_{i,j}) - (D_{i,j} + D_{i,j-1})(u_{i,j} - u_{i,j-1})\right)\end{aligned}$$

The discretization of the time-derivative is where we can see a difference between the two schemes we will use in this project. When ordinary differential equations are discretized one can clearly see how this difference arises, and so we will write our PDE using a new notation

$$D_t u = Ou \tag{2.24}$$

where the right-hand-side operator O indicates some operation like the double spatial derivative. Introducing the general discretization of the time derivative gives us equation 2.25 known as theta-rule. Setting $\theta = 0$ yields the FE discretization, and $\theta = 1$ the BE discretization.

$$\frac{u^{n+1} - u^n}{\Delta t} = O\left(\theta u^{n+1} + (1-\theta)u^n\right) \tag{2.25}$$

From the theta rule we can see that the only difference between the FE and BE scheme is at what time-step we evaluate the right-hand-side of the equation. The theta-rule can give other schemes as well, using some weighted average of the right-hand-side at t^n and t^{n+1} but we will not look into these in this project. We can now summarize by writing out the FE discretization as it will be implemented in 1d in equation 2.26.

$$u_i^{n+1} = \frac{\Delta t}{2\Delta x^2}\left((D_{i+1}+D_i)(u_{i+1}^n - u_i^n) - (D_i + D_{i-1})(u_i^n - u_{i-1}^n)\right) + u_i^n \tag{2.26}$$

We will come back to the FE discretization when we discuss stability later.
Looking at the BE discretization which is written out in 1d in equation 2.27 we notice that there are quite a few more unknowns per mesh-point.

$$u_i^{n+1} = \frac{\Delta t}{2\Delta x^2}\left((D_{i+1}+D_i)(u_{i+1}^{n+1} - u_i^{n+1}) - (D_i + D_{i-1})(u_i^{n+1} - u_{i-1}^{n+1})\right) + u_i^n \tag{2.27}$$

Writing out the calculations for a small mesh we recognize a pattern which we can exploit.

$$\begin{aligned}
u_0^{n+1} &= \frac{\Delta t}{2\Delta x^2}\left(2(D_0+D_1)(u_1^{n+1} - u_0^{n+1})\right) + u_0^n \\
u_1^{n+1} &= \frac{\Delta t}{2\Delta x^2}\left((D_2+D_1)(u_2^{n+1} - u_1^{n+1}) - (D_1 + D_0)(u_1^{n+1} - u_0^{n+1})\right) + u_1^n \\
u_2^{n+1} &= \frac{\Delta t}{2\Delta x^2}\left((D_3+D_2)(u_3^{n+1} - u_2^{n+1}) - (D_2 + D_1)(u_2^{n+1} - u_1^{n+1})\right) + u_2^n \\
u_3^{n+1} &= \frac{\Delta t}{2\Delta x^2}\left(2(D_2+D_3)(u_3^{n+1} - u_2^{n+1})\right) + u_3^n
\end{aligned}$$

Rearranging this and setting $a = \frac{\Delta t}{2\Delta x^2}$ gives us a normal system of linear equations

$$\begin{aligned}
u_0^{n+1}\left(1 + 2a(D_0+D_1)\right) - 2au_1^{n+1}(D_1+D_0) &= u_0^n \\
u_1^{n+1}\left(1 + a(D_2+2D_1+D_0)\right) - au_2^{n+1}(D_2+D_1) - au_0^{n+1}(D_1+D_0) &= u_1^n \\
u_2^{n+1}\left(1 + a(D_3+2D_2+D_1)\right) - au_3^{n+1}(D_3+D_2) - au_1^{n+1}(D_2+D_1) &= u_2^n \\
u_3^{n+1}\left(1 + 2a(D_3+D_2)\right) - 2au_2^{n+1}(D_3+D_2) &= u_3^n
\end{aligned}$$

which we arrange as

$$\begin{pmatrix}
1+2a(D_0+D_1) & -2a(D_1+D_0) & 0 & 0 \\
-a(D_1+D_0) & 1+a(D_2+2D_1+D_0) & -a(D_2+D_1) & 0 \\
0 & -a(D_2+D_1) & 1+a(D_3+2D_2+D_1) & -a(D_3+D_2) \\
0 & 0 & 1+2a(D_3+D_2) & -2a(D_3+D_2)
\end{pmatrix} \mathbf{u}^{n+1} = \mathbf{u}^n \tag{2.28}$$

$$\mathbf{A}\mathbf{u}^n = \mathbf{u}^{n-1} \tag{2.29}$$

Immediately we notice a problem with the implicit scheme. If we solve the system of equations by the fool-proof Gaussian elimination we will use some $\mathcal{O}(n^3)$ FLOPs per time-step. This will get even worse in more spatial dimensions; $\mathcal{O}(n^6)$ in 2d and $\mathcal{O}(n^9)$ in 3d. As a comparison the explicit scheme will make due with $\mathcal{O}(n^d)$ FLOPs. There are, however ways to improve this. Seeing as the matrix $\mathbf{A}$ does not change as long as none of the parameters change we can use a LU-decomposition. This will demand a decomposition of $\mathcal{O}(n^3)$ FLOPs, but all the subsequent steps will be $\mathcal{O}(n^2)$ FLOPs ($\mathcal{O}(n^{2d})$ for higher dimensions). We are still not quite at the level of the explicit scheme, but it is a clear improvement.
Looking closer at A we notice that it is not only sparse, but tridiagonal. This calls for further optimization which brings the required number of FLOPs down to $\mathcal{O}(n)$ making it equally efficient to the explicit scheme. More on tridiagonal Gaussian elimination later.

2.2.2 Stability

In section 2.2.1 we used the Forward Euler approximation to the time derivative. Unfortunately the resulting scheme is potentially unstable, as we shall now see. We start out by assuming that the solution $u(x,t)$ is on the form

$$u(x,t) = A^n \exp(ikp\Delta x) \tag{2.30}$$

where $i^2 = -1$ is the imaginary unit and A^n is an amplification factor which, for the solution 2.30 ideally should be $\exp(-\pi^2 t)$, but will be something else in the numerical case. We notice that we must have $|A| \leq 1$ if u is to not blow up. Inserting 2.30 in the simplified version of the variable coefficient scheme (where the coefficient is constant) gives us the following

$$\begin{aligned}
&\exp(ikp\Delta x)\left(A^{n+1} - A^n\right) = \\
&\qquad A^n \frac{D\Delta t}{\Delta x^2}\left(\exp(ik(p+1)\Delta x) - 2\exp(ikp\Delta x) + \exp(ik(p-1)\Delta x)\right) \\
&A^n \exp(ikp\Delta x)\left(A - 1\right) = \\
&\quad A^n \exp(ikp\Delta x)\frac{D\Delta t}{\Delta x^2}\left(\exp(ik\Delta x) - 2 + \exp(-ik\Delta x)\right)
\end{aligned}$$

Using the well known identities $\exp(iax) + \exp(-iax) = \frac{1}{2}\cos^2\left(\frac{ax}{2}\right)$ and $\cos^2(ax) - 1 = \sin^2(ax)$ gives us

$$A - 1 = \frac{D\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right) \tag{2.31}$$

We now insert for the "worst case scenario" $max(\sin^2\left(\frac{k\Delta x}{2}\right)) = 1$

$$A = \frac{D\Delta t}{2\Delta x^2} + 1 \implies \Delta t \leq \frac{\Delta x^2}{2D} \tag{2.32}$$

In 2d this criterion is halved, and for the anisotropic case we must insert for the maximum value of D which, again, will be the "worst case scenario".
If we insert the same solution (eq. 2.30) in the BE scheme we get

$$\exp(ikp\Delta x)\left(A^n - A^{n-1}\right) = A^n \frac{D\Delta t}{\Delta x^2}\left(\exp(ik(p+1)\Delta x) - 2\exp(ikp\Delta x) + \exp(ik(p-1)\Delta x)\right)$$

$$A^n \exp(ikp\Delta x)\left(1 - A^{-1}\right) = A^n \exp(ikp\Delta x)\frac{D\Delta t}{\Delta x^2}\left(\exp(ik\Delta x) - 2 + \exp(-ik\Delta x)\right)$$

which leads to

$$A = \frac{1}{1 + \frac{D\Delta t}{\Delta x^2}} \tag{2.33}$$

Equation 2.33 is smaller than 1 for all $\Delta t > 0$ which means that the scheme is unconditionally stable.

2.2.3 Truncation error

As we know the numerical derivative is not the analytical derivative, but an approximation. This approximation has a well defined residual, or truncation error which we can find by Taylor expansion.

$$R = \frac{u(t_{n+1}) - u(t_n)}{\Delta t} - u'(t_n)$$

Remember Taylor expansion of $u(t+h) = \sum_{i=0}^{\infty} \frac{1}{i!}\frac{d^i}{dt^i}u(t)h^i$

$$\begin{aligned} R &= \frac{u(t_n) + u'(t_n)\Delta t + 0.5u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3) - u(t_n)}{\Delta t} - u'(t_n) \\ &= u''(t_n)\Delta t + \mathcal{O}(\Delta t^2) = \mathcal{O}(\Delta t) \end{aligned}$$

We can do better than this by using another discretization scheme for the PDE, but in our case the PDE is not the only error source seeing as we will combine it with a random walk solver. Quantifying an error term The block-tridiagonal solver is taken from [], and is listed below.
for the random walk solver is not straightforward, but naturally it will be

closely coupled to the number of walkers used. So far the error seems to behave as expected, meaning that introducing very many walkers might reduce the error to $\mathcal{O}(\Delta t^2)$ if the number of walkers, N is proportionate to $N \propto \frac{1}{\Delta t^2}$. Since $\Delta t \leq \frac{D\Delta x^2}{2}$ by the stability constraint (in 1D), we will already for small meshes of some 20 points need to introduce ~ 600000 walkers per unit "concentration" per mesh-point in the walk-area. This will be such a costly operation that it will not necessarily be worth it.
The spatial derivative also has a well defined residual which is found by Taylor expansion.

$$R = \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{\Delta x^2} - u''(x_i) \tag{2.34}$$

Remember Taylor expansion of $u(x-h) = \sum_{i=0}^{\infty} \frac{1}{i!}\frac{d^i}{dt^i}u(x)(-h)^i$

$$R = \frac{u(x_i) + u'(x_i)\Delta x + 0.5u''(x_i)\Delta x^2 + \frac{1}{6}u^{(3)}(x_i)\Delta x^3) + \frac{1}{24}u^{(4)}(x_i)\Delta x^4 + \mathcal{O}(\Delta x^5)}{\Delta x^2} -$$
$$\frac{2u(x_i)}{\Delta x^2} +$$
$$\frac{u(x_i) - u'(x_i)\Delta x + 0.5u''(x_i)\Delta x^2 - \frac{1}{6}u^{(3)}(x_i)\Delta x^3) + \frac{1}{24}u^{(4)}(x_i)\Delta x^4 + \mathcal{O}(\Delta x^5)}{\Delta x^2}$$
$$- u'(x_i)$$
$$R = u''(x_i) + \frac{1}{12}u^{(4)}(x_i)\Delta x^2 + \mathcal{O}(\Delta x^5) - u''(x_i) = \mathcal{O}(\Delta x^2)$$

There are discretizations that can reduce this residual even further (although a second order scheme is usually considered adequate), but this time the stability criterion on the time derivative 2.2.2 will always be of the order $\mathcal{O}(\Delta x^2)$ and so we will never get a smaller error than this unless we change the time derivative.
For the Random walk we know from statistical mechanics that the fluctuations around a steady state is related to the number of walkers through eq. 2.35.

$$\langle \Delta u \rangle \propto \frac{1}{\sqrt{N}} \tag{2.35}$$

In the combined solver, we assume that equation 2.35 still holds for the RW-part of the solution even though we can only say for certain that it is correct for the first time-step. The number of walkers, N is now given by the defined conversion factor Hc as

$$N(x, y, t) = Hc \cdot U(x, y, t) \tag{2.36}$$

and the total number of walkers is the sum of the walkers on all the mesh-points. In each mesh-point the fluctuations are of the order $\sqrt{N(x_i, y_j, t_n)}^{-1}$, meaning that the convergence rate in each mesh-point is $\frac{1}{2}$.

2.2.4 Tridiagonal linear systems

The implicit discretization gives us a set of linear equations, or a linear system, to solve at each timestep. The physics of the system gives us a special form of linear system, namely a band diagonal system, where the number of non-zero bands on the matrix is dependent on the number of spatial dimensions we are in. The one dimensional case gives us a tridiagonal system, which can be solved extremely efficiently by the "tridiag" function listed above. In two spatial dimensions we are not quite as fortunate as in one dimension. We get a banded matrix with 2n bands and five non-zero bands, where n is the spatial resolution (which is equal in x and y direction). Rewriting the assembled matrix (see eq. A.2) to a block-matrix form gives us a tridiagonal matrix, where the entries are $n \times n$ matrices.

The fool-proof way to solve a linear equation $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is not a sparse matrix, by Gaussian elimination.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \mathbf{x} = \mathbf{b} \tag{2.37}$$

Is reduced to

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & (a_{22} - \frac{a_{21}a_{12}}{a_{11}}) & (a_{23} - \frac{a_{21}a_{13}}{a_{11}}) & (a_{24} - \frac{a_{21}a_{14}}{a_{11}}) \\ 0 & (a_{32} - \frac{a_{31}a_{12}}{a_{11}}) & (a_{33} - \frac{a_{31}a_{13}}{a_{11}}) & (a_{34} - \frac{a_{31}a_{14}}{a_{11}}) \\ 0 & (a_{42} - \frac{a_{41}a_{12}}{a_{11}}) & (a_{43} - \frac{a_{41}a_{13}}{a_{11}}) & (a_{44} - \frac{a_{41}a_{14}}{a_{11}}) \end{pmatrix} \mathbf{x} = \tilde{\mathbf{b}} \tag{2.38}$$

and further to

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & (a_{22} - \frac{a_{21}a_{12}}{a_{11}}) & (a_{23} - \frac{a_{21}a_{13}}{a_{11}}) & (a_{24} - \frac{a_{21}a_{14}}{a_{11}}) \\ 0 & 0 & (\tilde{a}_{33} - \frac{\tilde{a}_{32}a_{23}}{\tilde{a}_{22}}) & (\tilde{a}_{34} - \frac{\tilde{a}_{32}a_{24}}{\tilde{a}_{22}}) \\ 0 & 0 & (\tilde{a}_{43} - \frac{\tilde{a}_{42}a_{23}}{\tilde{a}_{22}}) & (\tilde{a}_{44} - \frac{\tilde{a}_{42}a_{34}}{\tilde{a}_{22}}) \end{pmatrix} \mathbf{x} = \tilde{\mathbf{b}} \tag{2.39}$$

until we have an upper triangular matrix. We then do a backwards sweep to solve for one element of the unknown vector, $\mathbf{x}$ at a time.

Since most entries are zero we can easily get away with only doing one forward sweep down the matrix, eliminating all the sub-diagonal matrix-entries, and then one backward sweep, which calculates the unknown vector $\mathbf{x}$. The algorithm is listed below as a function implemented in C++.

```
void tridiag(double *u, double *f, int N, double *a, double
    *b, double *c){
    double *temp = new double[N];
    for(int i=0;i<N;i++){
      temp[i] = 0;
    }
    double btemp = b[0];
    u[0] = f[0]/btemp;
    for(int i=1; i<N; i++){
        //forward substitution
        temp[i] = c[i-1]/btemp;
        btemp = b[i]-a[i]*temp[i];
        u[i] = (f[i] -a[i]*u[i-1])/btemp;
    }
    for(int i=(N-2);i>=0;i--){
        //Backward substitution
        u[i] -= temp[i+1]*u[i+1];
    }
    delete[] temp;
}
int main()
{
    return 0;
}
```

We can also modify the tridiagonal solver from the one-dimensional case so we can use it on block-tridiagonal systems. The modified algorithm for the block-tridiagonal matrix 2.40 is listed in **??**, and is in fact only the linear algebra version of the "tridiag" function, replacing the $1.0/btmp$ -terms with $(B_i + A_i H_{i-1})^{-1}$. The result of rewriting the $2n$-band diagonal matrix is the block matrix in equation 2.40.

$$\begin{pmatrix} B_0 & C_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A_1 & B_1 & C_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \ddots & \ddots & 0 & 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & A_i & B_i & C_i & 0 & & 0 & 0 \\ 0 & \ddots & 0 & \ddots & \ddots & \ddots & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & A_{n-1} & B_{n-1} \end{pmatrix} \begin{pmatrix} \mathbf{u}_0^{n+1} \\ \mathbf{u}_1^{n+1} \\ \vdots \\ \mathbf{u}_i^{n+1} \\ \vdots \\ \mathbf{u}_n^{n+1} \end{pmatrix} = \mathbf{u}^n \quad (2.40)$$

which we can also express as $M\mathbf{x} = \mathbf{k}$. Block-matrices named B_i are tridiagonal, and the ones named A_i or C_i are strictly diagonal.

There is a forward substitution

$$\begin{aligned} H_1 &= -B_1^{-1}C_1 \\ H_i &= -\left(B_i + A_i H_{i-1}\right)^{-1} C_i \\ \mathbf{g}_1 &= B_1^{-1}\mathbf{k}_1 \\ \mathbf{g}_1 &= \left(B_i + A_i H_{i-1}\right)^{-1}\left(\mathbf{k}_i - A_i \mathbf{g}_{i-1}\right) \end{aligned}$$

Followed by a backward substitution

$$\begin{aligned} \mathbf{x}_{n-1} &= \mathbf{g}_{n-1} \\ \mathbf{x}_i &= \mathbf{g}_i + H_i \mathbf{x}_{i+1} \end{aligned}$$

The algorithm requires inverting approximately 3n $n \times n$ matrices, which might be expensive. However, we only need to to the inversion once as long as the mass-matrix, M is unchanged so the expense is reduced. This should give us a computational intensity of around $\mathcal{O}(n^2)$ seeing as we only need to do one matrix-matrix multiplication where one matrix is diagonal, and two matrix-vector multiplications. All of which demand $\mathcal{O}(n^2)$ operations. This reduction in computational cost makes the implicit scheme as effective as the implicit FE scheme.

In three dimensions we are even more unfortunate and get an $2n^2$-banded matrix and seven non-zero bands. As we can see in the appendix (eq. A.5) we can write this so it looks like the block tridiagonal linear system we got in 2d, and that could be solved by the block-tridiagonal solver. The difference is that the entries i n the block-matrices A_i, B_i and C_i are block-matrices themselves meaning that we must work with $n^2 \times n^2$ matrices. All in all we should get a performance of around $\mathcal{O}(n^3)$ if we take care to save the inverted matrices from the forward substitution. This is about the same performance as the explicit scheme gives us, but again without the stability issue.

2.3 Combining the two solvers

This section will deal with the actual combination of the two models.

2.3.1 The basic algorithm

The basic structure of the program is rather similar to the physical problem. There is one dendrite-object which contains the PDE-solver for the normal

diffusion equation, with the possibility to use a random walk solver instead. On the dendrite object spines can be placed, which in the physical world are the receiving end of a synapse. Depending on what is being modeled, synaptic input is modeled by randomly added spikes of some random number of molecules which spawn at the far end of the spine. In the overlapping points where the spine is located on the dendrite mesh, the coupling is done as follows: If a random walker in the spine comes in contact with the position labeled as the "end" of the spine it is moved from the list of active walkers to a list of walkers which have moved out of the spine. Similarly, at each time-step a part of the PDE-solution corresponding to one walker will diffuse into the spine with a certain probability. It might be desirable for a walker to only be able to diffuse out of the spine with some probability as well, or for the walkers which diffuse into the spine to have some drift term, but these are minor updates and might be added later if needed.

There is also the possibility of modeling parts of the dendrite-mesh as random walk (This can be done in 2d as well as 1d). This is done by choosing some points on the mesh and sending the to the "AddWalkArea" method which will map them to an index and set the initial condition for the walk. Although anisotropy will follow into the random walk solver, by the method provided by Farnell and Gibson [1]. At each time-step the solve-method of the combined solver is called, which in turn calls the solve method for the PDE-solver. The solution from the PDE-solver is used to calculate the number of walkers by eq. **??** in each mesh-point on the PDE-mesh, and then give each walker a random position in a square around its mesh-point ($\pm\frac{\Delta x}{2}$). Because the sum of the PDE-solution over the random walk area of the mesh might be different from one time-step to the next (eq. 2.41) the conversion from PDE-solution to random walker distribution must be done at every time-step. The alternatives are to remove or add the difference at each time-step, but this will require checking that each mesh-point has the "correct" number of walkers and updating the number to correspond with the solution from the PDE-solver. *which is what we are doing already?* Or the conversion factor could be adjusted at each time-step. The latter is largely a bad solution because it ruins transparency and might introduce even more fluctuations in the solution.

$$\sum u_{i,j}^{n} \neq \sum u_{i,j}^{n+1} \tag{2.41}$$

After the random walk integration the two solutions are combined by a simple average. A few other methods have been tested (see chapter **??**) but discarded. The average of the two solutions is then set as the new "initial

condition" for the next time-step, and the process repeats itself.

The results of these are inserted in the solution from the PDE using some routine (e.g. the average of the two) and the time-step is done. A schematic of the algorithm is provided in figure 2.1.

Figure 2.1: Schematic diagram of the algorithm.

2.3.2 Convergence rate

In chapter 2.2.3 we discussed the error that arises as a result of adding random walkers on parts of the mesh. We found that the amplitude of the fluctuations per mesh-point is proportionate to $\frac{1}{\sqrt{N}}$ where N is the number of walkers related to the mesh-point. Combining the two models means adding fluctuations to the approximation to the exact solution. Seeing as we send this combined solution to the PDE-solver as an "initial condition" for the next time-step we have made a compromise in accuracy. The error-estimate we will define later is still dependent of Δt, but the dependency is now of the order $\mathcal{O}(\sqrt{\Delta t})$. This also further supports the claim that there is no need to find a very precise scheme to solve the PDE.
We will test this by doing a convergence test in time keeping the number of walkers constant.

2.3.3 Potential problems or pitfalls with combining solutions

In this chapter we will identify and discuss a few obvious difficulties we can expect to run into in our planned project. As far as possible we will also explain how to solve the problems or how to live with them.

- Different timescales
  The PDE-solver will be operating with some time-step Δt which will, depending on the discretization of the PDE, have some constraints and will definitely have an impact on the error. The walkers will, as we have just seen, solve the diffusion equation as well, but with some different $\Delta\tilde{t}$ which is smaller than the time-step on the PDE level. Depending on the coupling chosen between the two models this difference will have some effect or a catastrophic effect on the error. Running some number of steps, N, on the random-walk level should eventually sum up to the time-step on the PDE level, $\sum_{i=0}^{N} \Delta\tilde{t} = \Delta t$. It turns out, as we will see in

section 2.3.4 that we can make sure the coupling is as good as it gets by restricting the step length of the walkers.

- Boundary conditions
  To combine the two models we will need to put restricting boundary conditions on the random walks. This is not usually done (as far as I have seen), but not very difficult. Finding a boundary condition that accurately models the actual system turns out to be quite straightforward. We can assume that the number of walkers in the walk-domain is conserved for each PDE time-step, and thus no walkers can escape the domain. Implementing perfectly reflecting boundaries solves this quite well. This means that the flux of walkers out of a boundary is zero, which is the same as Neumann boundary conditions on the PDE level.
  Dirichlet boundaries can (probably) be implemented by adding or removing walkers on the boundaries (or in a buffer-zone around them) until we have the desired concentration of walkers.

- Negative concentration of walkers
  The concentration of walkers is calculated as $NP(x,t)$ where $P(x,t)$ is really only an estimate of the actual probability distribution, calculated by dividing the number of walkers in one area $x \pm \frac{\Delta x}{2}$ by the total number of walkers. Seeing as negative probabilities does not make sense, and neither does a negative number of walkers, we will eventually run into some problems if the solution of the PDE takes negative values (which it most likely will not do). I can find no good solutions to this problem, but a workaround consists of storing the sign of the solution over each time-step, converting the absolute value to a distribution of random walkers and multiplying back with the sign after the RW solution is done. This workaround has a problem in that a transition from positive to negative value will lead to a "valley" in the absolute-value solution. A normal PDE solution of this kind of initial condition will very rapidly even out the "valley", and so a value which should have been zero (a node-point) will get some other value (say some fraction of the conversion factor). This again leads to a larger discontinuity when the solution from the RW model is multiplied by the sign again.

- Smooth solutions
  A diffusion process is very effective when it comes to dampening fast fluctuations, and so any solution of the diffusion equation will be smooth. When we introduce a stochastic process, we will potentially also introduce fast fluctuations from one time-step to the next. In this case we

are faced with a dilemma; on the one hand there is the smoothness of the solution to consider, on the other hand we have introduced the stochastic term believing that it adds detail to our model. The approach we tried to use to this was to do some curve-fitting using both of the solutions. A polynomial regression model was implemented in 1d, but regardless of degree and what points were used, the result was a lot worse than just the average of the two solutions. Another idea is to implement a cubic spline interpolation over the area, but this too has its problems. An interpolation forces the solution to have a value at the interpolating points, and seeing as we cannot say for certain which value is correct how shall we pick the interpolating points?

- Number of time-steps on the random walk level
  As the time-step on the PDE level is increased above the stability criterion of the FE scheme towards more efficient sizes we are faced with the problem of whether or not to increase the number of time-steps on the RW level. Strictly speaking we do not have to do this, seeing as we adjust the step-length of the walkers with respect to the time-step (see eq 2.45). As an initial value we put the number of time-steps to 100, but this was more a guess of how many are necessary for the central limit theorem to have effect than anything else. The question really boils down to how we define our model, which we have yet to do in an accurate way.

- Random walks in 3D
  Both 1 and 2 dimensional space are spanned completely by a random walk, but space of 3 or more dimensions is not. This does not have to be a problem, seeing as we have proved that the random walk fulfills the diffusion equation (chapter 2.1.2) and we are not trying to span the complete 3d space, but we could potentially meet some difficulties as a result of this property of the random walk.

2.3.4 Probability distribution and time-steps

As we saw in section 2.1.2 the probability of finding a walker at a position x_i after some N time-steps (on the walk-scale) is (in the limit of large N) given as the Gaussian distribution. In our application, however, we are not interested in finding the walker at an exact position, but in an interval around the mesh-points sent to the walk-solver. This interval is (for obvious reasons) $x_i \pm \frac{\Delta x}{2}$ where Δx is the mesh resolution on the PDE level. We will also run the walk solver for some N time-steps on the random-walk scale (where N

steps on the random walk scale is the same as one step on the PDE scale). This slightly modifies our distribution into

$$P(x_i \pm \Delta x, t_{n+1}) = \frac{1}{\sqrt{4\pi DN\Delta\tilde{t}}} \exp\left(\frac{(x \pm \Delta x)^2}{4DN\Delta\tilde{t}}\right) \tag{2.42}$$

This makes the concentration of walkers $C(x,t) = MP(x,t)$

$$C(x_i \pm \Delta x, t_{n+1}) = \frac{M}{\sqrt{4\pi DN\Delta\tilde{t}}} \exp\left(\frac{(x \pm \Delta x)^2}{4DN\Delta\tilde{t}}\right) \tag{2.43}$$

For each PDE time-step we reset the walkers to have some new initial condition. We do this because the concentration over the "walk-area" will change with each PDE time-step. The point is that $C(x_i \pm \Delta x, t_{n+1})$ will be dependent on the initial condition $C(x_i \pm \Delta x, t_n)$.

Looking at the difference in time-step size between the two length scales we see from equation 2.4 that the step-size on the random walk scale is dependent on the variance in the actual steps (This is in principle the Einstein relation).

$$\sigma^2 = \langle \Delta x^2 \rangle = 2DN\Delta\tilde{t} \implies \Delta\tilde{t} = \frac{\langle \Delta x^2 \rangle}{2DN} \tag{2.44}$$

Equating this with 2.8 gives us a first order approximation to the step-length, l

$$\begin{aligned} \langle \Delta x^2 \rangle &= 4pqNl^2 = 2DN\Delta\tilde{t} \\ l &= \sqrt{2D\Delta\tilde{t}}. \end{aligned} \tag{2.45}$$

Of course this is assuming that we use a random walk algorithm of fixed step-length.

Equation 2.45 is proportional to the square-root of the adjusted time-step. We have already suggested that the error term from the RW simulation depends on the number of walkers we use (or the conversion rate). This equation suggests that the error term also depends on the time step. Though this might seem a bit frustrating at first glance, but it answers a question we asked earlier; how many time-steps do we need to take at the RW-level. We have the intuitive relation between the RW time-step, $\Delta\tilde{t}$ and the PDE time-step through the number of steps at the RW level, T:

$$\Delta\tilde{t} = \frac{\Delta t}{T}$$

Further we suggested that the error from the RW simulation is proportional to the root of the time-step $\epsilon \propto \sqrt{\Delta\tilde{t}} = \sqrt{\frac{\Delta t}{T}}$. We want to force this error

to behave as $\mathcal{O}(\Delta t)$, and see that we can adjust the number of time-steps at the RW level in order to make this happen.

$$\mathcal{O}(\Delta t) > \sqrt{\frac{\Delta t}{T}} \implies T > \frac{1}{\Delta t} \tag{2.46}$$

Combined with the demand to the number of walkers we will quickly end up with an extreme computational demand if we want to force our model to have first order convergence (not to mention second order convergence). Fortunately we will ignore this demand outside of verification because there is little physical meaning left if we use the demanded number of walkers.

2.4 Geometry

Any finite difference method is problematic to solve on anything else than a rectangular grid. When we additionally use an implicit FD method we will add a "demand" of having a square grid as well. Fortunately the implicit solvers we use are stable.
In the physical scope of this thesis we do not find any rectangular shapes to apply our system to. Actually we do not have any well defined geometry to apply our system on, but that is of less importance. The purpose of this project is to investigate the actual coupling of two models for the same problem.
If we want to model diffusion on a general geometry by a FD method we could transform the grid to a unit-square through a general transform.

$$\mathbf{r} = \mathbf{T}(\mathbf{q})$$

Where $\mathbf{r} = (x, y)$ is the position in physical space, $\mathbf{q} = (\xi, \eta)$ is the position on the unit-square that is the computational space and $\mathbf{T}$ is the transformation. The transformation is achieved by the functions $x(\xi, \eta)$ and $y(\xi, \eta)$. After a lot of math, including some differential geometry we find that the diffusion equation in computational space still can be written as

$$\frac{\partial C}{\partial t} - \nabla \cdot \mathbf{J} = 0$$

but the total flux vector $\mathbf{J} = \mathbf{f} + \phi$ now has the properties

$$\nabla \cdot \mathbf{f} = \frac{1}{g} \cdot \left(\mathbf{f} \cdot \frac{\partial}{\partial \xi} \left(\frac{\partial y}{\partial \eta}, \frac{-\partial x}{\partial \eta} \right) + \mathbf{f} \cdot \frac{\partial}{\partial \eta} \left(\frac{-\partial y}{\partial \xi}, \frac{\partial x}{\partial \xi} \right) \right)$$
$$\nabla \phi = \frac{1}{g} \left(\phi \cdot \frac{\partial}{\partial \xi} \left(\frac{\partial y}{\partial \eta}, \frac{-\partial x}{\partial \eta} \right) + \phi \cdot \frac{\partial}{\partial \eta} \left(\frac{-\partial y}{\partial \xi}, \frac{\partial x}{\partial \xi} \right) \right)$$

where g is the Jacobian of the transformation $\mathbf{T}$. In other words we would need to discretize a completely new and much more complicated equation in order to solve for a general geometry. We would also need to have some idea of what the functions $x(\xi, \eta)$ and $y(\xi, \eta)$ are, which is not necessarily something we have access to. Furthermore we already have access to Finite Element software which can take any geometry as a mesh, and so a simpler way of using a more interesting geometry would be to change the PDE-solver in the developed software to a FEniCS solver.

Chapter 3

Analysis

3.1 Some discussion

This chapter will contain most of the numerical error analysis and some of the discussion of this analysis as well as a recap of the methods used for error analysis in general, and how they are adapted to this particular problem.

In the numerical setup chosen for this project we will potentially introduce several new error sources in addition to the normal errors introduced by numerical solution of any equation (see section 2.2). When a part of the solution acquired is replaced by the solution from a stochastic model, we will change the initial condition to the next iteration in time. This might have a number of effects on our final solution. Figure **??** we can see the typical effects of solving an equation numerically. When we impose a stochastic solution on top of this again, we will get fluctuations around the approximations to the solution at the new time-step which most likely worsens our approximation. The aim of this chapter is to verify our implementation and investigate the effects of adding the stochastic solution to the normal PDE solution.

3.1.1 The error estimate

Before we do anything we should specify what we use to measure the error which will be denoted as ϵ. Throughout this thesis the term error is used quite lazily, but unless something else is specified we refer to the expression

$$\epsilon(t_i) = ||u_e(t_i) - u(t_i)||_2 \tag{3.1}$$

where u_e is the exact (manufactured) solution to the equation, and u is the result from the numerical simulation. We use this error-estimate because it is time-dependent, thus letting us explicitly see how the error evolves over time. The error is calculated over the entire mesh, letting us see clearly if the error from the random-walk areas are dominating, or (otherwise) how the PDE-scheme is holding up. Another error estimate we will use later is the maximum value of the error. This will be used for the convergence tests to make sure that we overlook any effects of simulating for a long time.

$$\epsilon = max(\epsilon(t_i)) \tag{3.2}$$

3.1.2 Verification techniques

There are quite a few ways to verify an implementation. In this thesis we will focus on three which I think are adequate for no particular reason.

- Method of manufactured solutions
  A normal way of checking that our scheme of choice is implemented correctly is by making an exact solution to the equation and checking that the error is of the expected order. Since the software contains both an explicit FE scheme and an implicit BE scheme we will verify the two of them alongside one another. We will start with the simplest diffusion equation (eq. 2.13) and add complexity until we have the desired expression. Both schemes are expected to have an error-term of the order of Δt, which in the FE case is limited by a stability criterion. The next thing we do is decide that the solution to the diffusion equation should be equation 3.3 which satisfies our equation if we set the diffusion constant to 1.

- Exact numerical solutions
  At least for the explicit schemes we can find an exact solution to the scheme itself seeing as the scheme is a difference equation. The calculations are shown step-by-step in chapter 3.2.3.

- Convergence tests
  This must be combined with the manufactured solution, but takes a slightly different approach to quantifying the error estimate. We start bu calculating some form of error estimate, and chose a value to represent the error of the entire simulation. This could be the maximum error for the entire simulation, or an integrated measure. We do multiple simulations while improving the parameter we want to investigate, say how the size of the time-step influences the error. Finally, using equation **??** we get a number indicating the improvement in the error estimate by improving the parameter in question. This number indicates the order of convergence which is one for FE and BE since the error goes as $\mathcal{O}(\Delta t)$, and two for our approximation to the second derivative in space since the error goes as $\mathcal{O}(\Delta x^2)$. A convergence rate of 1 means that halving the parameter will (roughly) halve the error, while the same reduction for a second order convergence will reduce the error by 4.

3.2 Verification of PDE solvers

To verify the implementation of the PDE-solvers we will now run through the steps outlined in the previous chapter. We will also add complexity along the way, but this will be described when we get there. First of all we will test the simplest possible implementation which is equation **??**. Since both

the FE and BE discretization have been implemented we will test both of them, but mostly we will use the BE discretization because it has no stability criterion. For most of the tests we will force the solution to be equation 3.3 in 1d which satisfies our equation if we set the diffusion constant to 1.

Despite testing the simple cases first we will do all the tests on the same implementation (the full one) because it should be able to recreate the simple cases as well. Setting $D(x, y) = 0.5$ as an array with all entries equal and inserting this into the schemes will return the average of $D(x, y)$ at different mesh-points, and effectively this is the same as having only a constant. We will also do all verification without random walkers.

3.2.1 Manufactured Solutions

As mentioned we force the solution to be equation 3.3 because it fulfills the boundary conditions we have chosen.

$$u(x,t) = e^{-t\pi^2}\cos(\pi x) + 1 \tag{3.3}$$

Equations 3.4 and 3.5 prove that the manufactured solution we have chosen will indeed fulfill the diffusion equation (eq. 2.13).

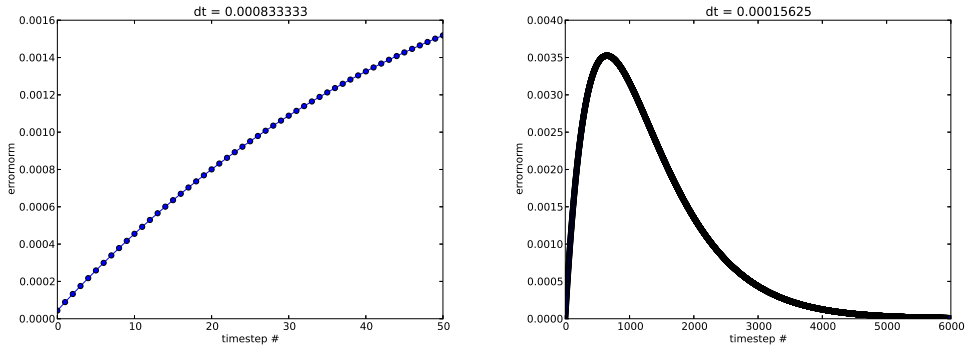
$$\frac{\partial}{\partial t}e^{-t\pi^2}\cos(\pi x) + 1 = D\frac{\partial^2}{\partial x^2}e^{-t\pi^2}\cos(\pi x) + 1 \tag{3.4}$$

$$-\pi^2 e^{-t\pi^2}\cos(\pi x) = -\pi^2 e^{-t\pi^2}\cos(\pi x) + 1 \implies 1 = 1 \tag{3.5}$$

The error in space is determined by two factors, the actual error caused by the approximation to the second derivative, which is of the order of Δx^2 and, in the FE case, the error term coming from the time derivative due to the stability criterion (eq. 2.2.2), which is also of the order Δx^2. In 1D we get the plots in figure 3.1a of error described in chapter 3.1.1, with the corresponding plots of the convergence rates in figure **??** that verify the order of the error term. For longer simulations, we expect the analytic solution to reach a steady state which we find in the limit of large t.

$$u(x, t \to \infty) \to e^{-\infty}\cos(\pi x) + 1 \to 1 \tag{3.6}$$

The numerical scheme should be able to represent this to machine precision (10^{-16}), meaning that the numerical solution should start converging to zero after some number of times steps, but this might depend on how the derivatives as estimated so we say that it should in the very least stabilize. Figures **??** and **??** seem to show the expected behaviour from the error estimate, which implies that the implementation is correct so far.



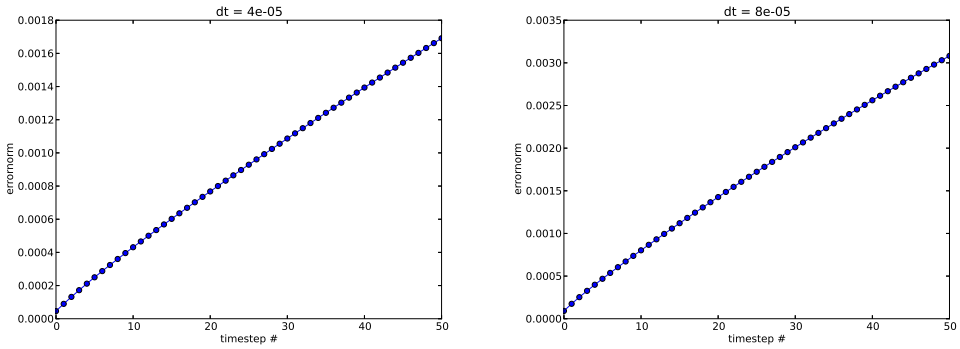
(a) (b)

Figure 3.1: Numerical error for 1D Forward Euler discretization of the PDE. Nothing else is done to the simulation.

As we discussed in section 2.1.4 we should also be able to model diffusion where the diffusion constant is not a constant. The scheme we derived in section 2.2.1 already takes this into account, and so all we need to do is to add the drift term which was discussed in section **??** to the scheme. Like before we should verify that the scheme solves the equation to the expected accuracy by using a manufactured solution 3.3 and tweaking the source term so this function solved the equation 2.20. When $v = 0$ and $D(x) = \pi x$ the source term becomes

$$\begin{aligned} -\pi^2 \exp\left(-\pi^2 t\right) \cos\left(\pi x\right) &= -\pi \exp\left(-\pi^2 t\right) \frac{\partial}{\partial x} \pi x \sin(\pi x) + f(x,t) \\ -\pi^2 \cos\left(\pi x\right) &= -\pi^2 \left(\sin(\pi x) + \pi x \cos(\pi x)\right) + \tilde{f}(x) \\ \tilde{f}(x) &= \pi^2 \left(\sin(\pi x) + \cos(\pi x)(\pi x - 1)\right) \end{aligned}$$

Once again $f(x,t) = \exp\left(-\pi^2 t\right) \tilde{f}(x)$. Figure 3.2 shows the error norm of the result of simulations of this equation with different values of Δt.



(a) (b)

Figure 3.2: Verification of anisotropic diffusion equation implementation

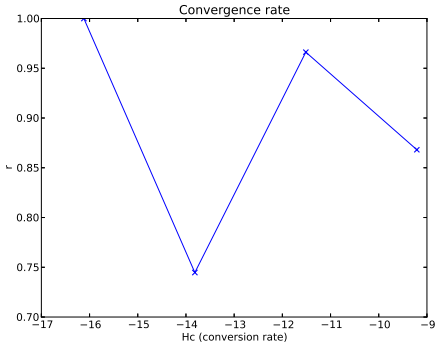
Again, the error is of the order of Δt and is roughly halved by halving Δt.

3.2.2 Convergence Tests

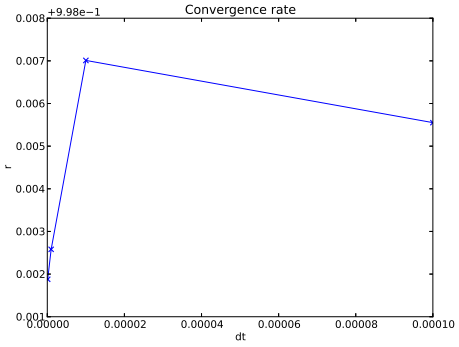
The next step in testing the implementation will be to perform a convergence test in time. We omit the spatial convergence test because an incorrect implementation of the spatial derivative would be clearly visible both in the visualization of the simulation (the solution blows up), and in the error plot seeing as the spatial error would dominate. We carry out the convergence test by doing several simulations with different values for Δt and comparing the errors by equation 3.7. A result of such an experiment for the FE scheme using the Δt values listed below is found in figure **??**. The expected value of r is approximately 1. The result is not perfect, but still close to 1. For the BE scheme we get the convergence rate shown in figure **??**. Again, the expected order of convergence is 1 and this time the result is almost perfect.

$$r = \frac{\ln(E_{i+1}/E_i)}{\ln(\Delta t_{i+1}/\Delta t_i)} \tag{3.7}$$

```
dt = [1e-4,1e-5,1e-6,1e-7,1e-8]
```



(a) Convergence test for the FE scheme. The x axis is $\ln(\Delta t)$.



(b) Convergence test for the BE scheme. The x axis is Δt, the y-axis (though a little hard to see) is zoomed in around r=1.

Figure 3.3: Convergence tests for explicit and implicit schemes solving the simple diffusion equation (eq. 2.13).

We can also do a convergence test, equal to the one we did in 1d, to check that the scheme converges to 1 (by equation 3.7) for smaller Δt. The results of this test are shown in figure 3.4 and it does converge nicely to one.

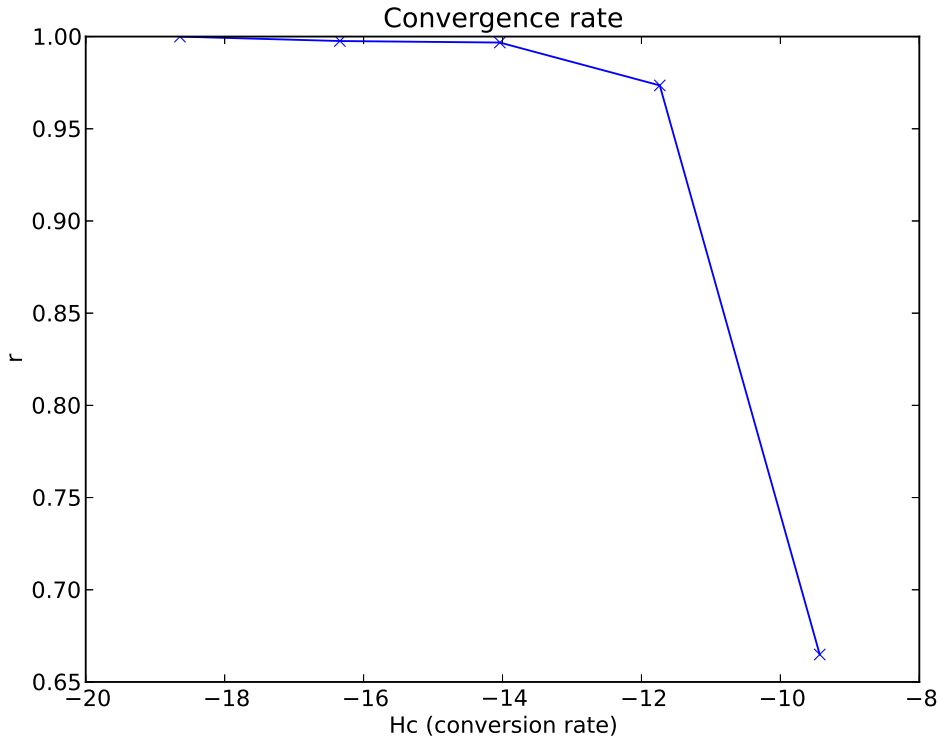


Figure 3.4: Convergence test for the FE scheme in 2d using Δt ranging from the stability criterion $\frac{\Delta x \Delta y}{5}$ to the same ratio divided by 100000 in increments of 10^{-1}.

3.2.3 Exact numerical solution

The FE scheme also has an exact solution seeing as it is in fact a difference equation. We can find this (for this exact initial condition, eq. 3.3) if we formulate the scheme as

$$u^{n+1} = D\Delta t u^n_{xx} + u^n \tag{3.8}$$

and insert for the first few iterations.

$$
\begin{aligned}
u^1 &= D\Delta t u^0_{xx} + u^0 \\
u^2 &= D\Delta t u^1_{xx} + u^1 = D\Delta t \left[D\Delta t u^0_{4x} + u^0_{2x} \right] + u^0 \\
&= (D\Delta t)^2 u^0_{4x} + 2D\Delta t u^0_{2x} + u^0 \\
u^3 &= D\Delta t u^2_{xx} + u^2 = D\Delta t \left[(D\Delta t)^2 u^0_{6x} + 2D\Delta t u^0_{4x} + u^0_{2x} \right] + (D\Delta t)^2 u^0_{4x} + 2D\Delta t u^0_{2x} + u^0 \\
&= (D\Delta t)^3 u^0_{6x} + 3 (D\Delta t)^2 u^0_{4x} + 3D\Delta t u^0_{2x} + u^0 \\
u^4 &= D\Delta t u^3_{xx} + u^3 = \ldots \\
&= (D\Delta t)^4 u^0_{8x} + 4 (D\Delta t)^3 u^0_{6x} + 6 (D\Delta t)^2 u^0_{4x} + 4D\Delta t u^0_{2x} + u^0
\end{aligned}
$$

Which we can generalize to

$$u^{n+1} = \sum_{i=0}^{n} \binom{n}{i} (D\Delta t)^i u^0_{2ix} \tag{3.9}$$

where we have

$$u^0_{2ix} = (-1)^i \pi^{2i} \cos(\pi x)$$

from the initial condition. This finally gives us the exact numerical solution of the FE scheme in time.

$$u^{n+1} = \sum_{i=0}^{n} \binom{n}{i} \pi^{2i} (-1)^i (D\Delta t)^i \cos(\pi x) \tag{3.10}$$

Notice that we have not used the approximation to the spatial derivative, but rather the analytical derivative. We find the approximation that the computer uses to the spatial derivative in the same way as for the time derivative.

$$
\begin{aligned}
u^0_{xx} &= \frac{1}{\Delta x^2} (\cos(\pi(x + \Delta x)) - 2\cos(\pi x) + \cos(\pi(x - \Delta x))) \\
&= \frac{2}{\Delta x^2} (\cos(\pi \Delta x) - 1) \cos(\pi x) \\
u^0_{4x} &= [u^0_x x]_{xx} = \frac{1}{\Delta x^2} \left[\frac{2}{\Delta x^2} (\cos(\pi \Delta x) - 1) (\cos(\pi(x + \Delta x)) - 2\cos(\pi x) + \cos(\pi(x - \Delta x))) \right] \\
&= \frac{4}{\Delta x^2} (\cos(\pi \Delta x) - 1)^2 \cos(\pi x) \\
&\ldots
\end{aligned}
$$

We immediately see that this pattern continues, and we get the general formula in equation 3.11 for the exact numerical solution.

$$u^{n+1} = \sum_{i=0}^{n} \binom{n}{i} (D\Delta t)^i \frac{2^i}{\Delta x^{2i}} (\cos(\pi \Delta x) - 1)^i \cos(\pi x) \tag{3.11}$$

We expect the FE scheme to represent this solution more or less to machine precision, at least to 15 digits. There are, however two issues with the solution 3.11:

- Δx^{2i} will quickly tend to zero, and the computer will interpret it as zero. This will cause division by zero, which again ruins the simulation. This can be fixed rather simply by testing if $\Delta x^{2i} > 0$ and returning zero if the test returns false.
- $\binom{n}{i}$ goes to infinity for large n and i. We will eventually (for $n >\sim 170$) meet overflow. Figure 3.5 shows that at least for relatively few time steps we can drop the troublesome terms, but effectively this limits ous simulation to some 170 steps.

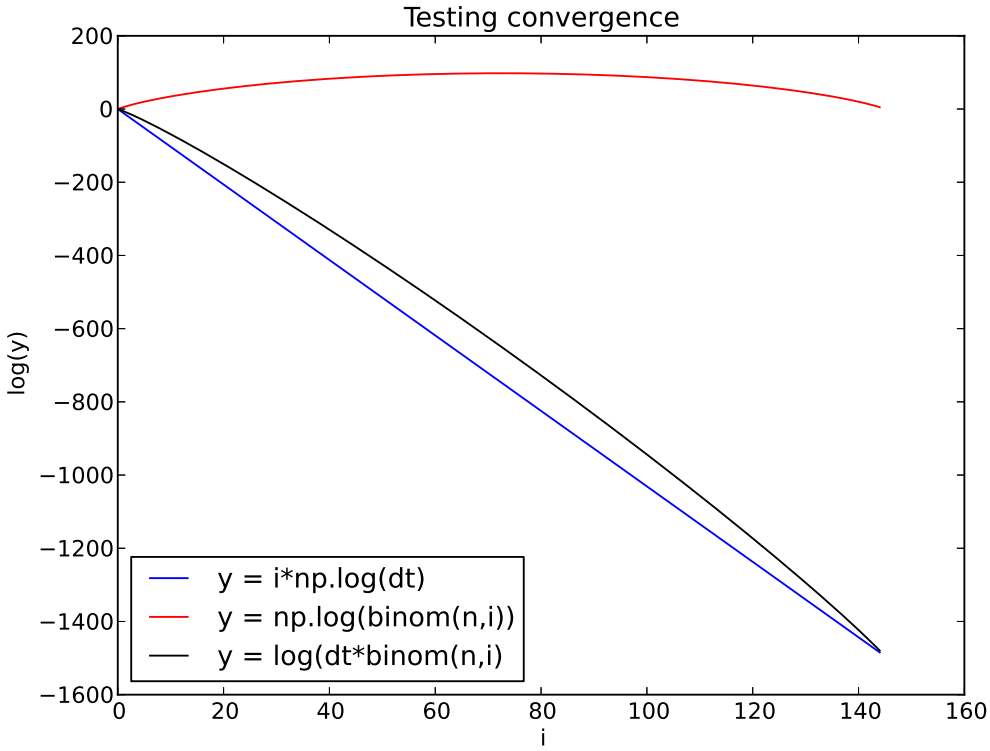


Figure 3.5: Testing the relation between $(D\Delta t)^i$ and the binomial coefficients.

The results from testing the FE scheme are found in figure 3.6. We see that the error in the worst case is about an order of magnitude worse than

we expected. This is most likely due to the fact that we are cutting part of the solution, and over several time steps the error we do might accumulate.

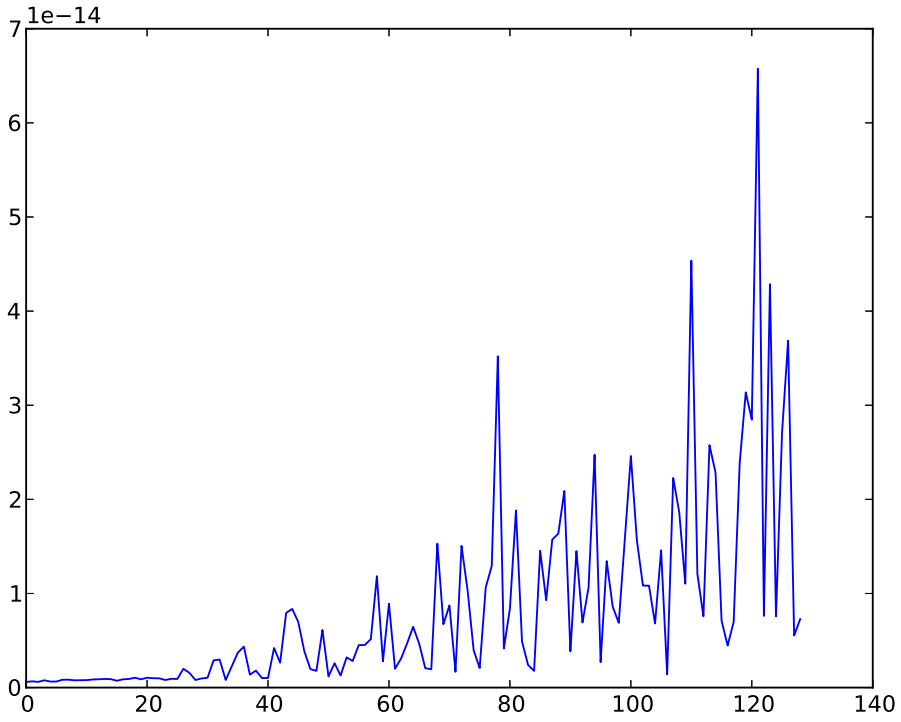


Figure 3.6: Error plot for 1d FE scheme compared to the exact numerical solution 3.11 with the modifications suggested earlier.

We find the exact numerical solution to the 2d diffusion equaiton with eq 3.12 as initial condition by using the same method as we did in 1d. The exact numerical solution of the FE scheme can be found in equation 3.13, and again we expect the scheme to be able to reproduce this to more or less machine precision.

$$u(x, y, t = 0) = \cos(\pi x)\cos(\pi y) \tag{3.12}$$

The result of a test simulation of this is shown in figure 3.7. Again, as in we did in 1d, we see that although the error is very small, and start out with machine precision, it does increase and even more than in the 1d case. This is *probably* because of the terms we have do drop in the exact numerical solution due to overflow and so on, which accumulate in the numerical solution from

the scheme. We should, in other words, be pleased that the error starts out with machine precision, and stays small for the amount of time steps we can simulate and still have something to compare it with.

$$u^{n+1} = \sum_{i=0}^{n} \binom{n}{i} (D\Delta t)^i \left[2^{i-1} \cos(\pi x) \cos(\pi y) \left(\frac{(\cos(\pi \Delta x))^i}{\Delta x^{2i}} + \frac{(\cos(\pi \Delta y))^i}{\Delta y^{2i}} \right) \right] \tag{3.13}$$

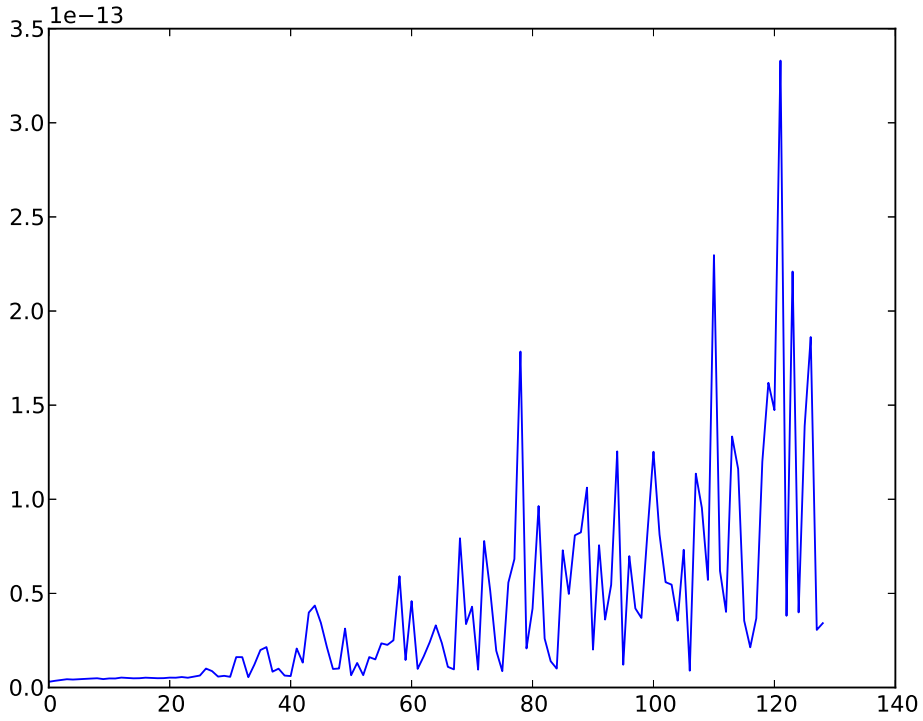


Figure 3.7: Numerical solution from the FE scheme versus the exact numerical solution of the FE scheme in 2d. we have used a Δt which is almost on the stability criterion, $\Delta t = \frac{\Delta x \Delta y}{5} = 8e - 05$.

3.3 Testing the Random walk implementation

To verify our implementation, and perhaps gain some new insight we will do some testing on the random walks implementation as well. This means that we have to find a solution of the diffusion equation 2.13 for an initial condition

we can recreate with random walkers to the best possible precision. The absolute simplest initial condition to recreate is the Heaviside step function, defined in equation 3.14.

$$H(x-a) = \begin{cases} 1 & x \geq a \\ 0 & x < a \end{cases} \tag{3.14}$$

In order to verify our implementation we must solve the diffusion equation 2.13 for this new initial condition. This is most easily done through separation of variables. We have

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}; \quad \frac{\partial u(0,t)}{\partial x} = \frac{\partial u(1,t)}{\partial x} = 0$$
$$u(x,0) = H\left(x - \frac{1}{2}\right); \quad D = 1$$

and

$$u(x,t) = F(x)T(t) \implies \frac{T'(t)}{T(t)} = \frac{F''(x)}{F(x)}$$

where the primes denotes the respective derivatives. We separate the equation using a separation constant λ

$$T'(t) - \lambda T(t) = 0 \implies T(t) = C\exp(\lambda t)$$
$$F''(x) - \lambda F(x) = 0 \implies F(x) = C_1\exp(\sqrt{\lambda}x) + C_2\exp(-\sqrt{\lambda}x)$$

Where C, C_1 and C_2 are arbitrary constants. Choosing $\lambda = -\mu^2$ lets us rewrite the spatial solution in terms of sines and cosines. There are really three choises here; $\lambda = -\mu^2$, $\lambda = \mu^2$ and $\lambda =?$ but we chose the former because the results of the other choises are unphysical.

$$F(x) = a\cos(\mu x) + b\sin(\mu x)$$

The boundary conditions gives us

$$F'(0)T(t) = F'(1)T(t) = 0$$

Since the time dependent solution cannot be exactly zero and is independen of position by construction ($C\exp(\lambda t)|_{x=0} = C\exp(\lambda t) \neq 0$), the first derivative of the spatial solution must be zero at the boundaries

$$F'(x) = -a\mu\sin(\mu x) + b\mu\cos(\mu x)$$
$$F'(0) = -a\mu\sin(0) + b\mu\cos(\mu x) = b\mu\cos(\mu x) \implies b = 0$$
$$F(1) = a\cos(\mu) \implies \mu = n\pi$$

Telling us that a Fourier series in cosines is the solution to the equation, and it will look like this.

$$u(x,t) = a_0 + \sum_{n=1}^{\infty} a_n \exp\left(-(n\pi)^2 t\right) \cos(n\pi x) \tag{3.15}$$

The coefficients are found by approximating the initial condition

$$\begin{aligned}
a_0 &= \int_0^1 H(x-0.5)dx = \frac{1}{2} \\
a_n &= 2\int_0^1 H(x-0.5)\cos(n\pi x)dx = 2\int_{0.5}^1 \cos(n\pi x)dx \\
&= \frac{2}{n\pi}\left[sin(n\pi x)\right]_{0.5}^1 = \frac{2}{n\pi}\sin(n\pi) - \sin(\frac{n\pi}{2}) \\
a_n &= \frac{2\sin(\frac{n\pi}{2})}{n\pi}
\end{aligned}$$

which gives us the final solution

$$u(x,t) = \frac{1}{2} + \sum_{n=1}^{\infty} \frac{2\sin(\frac{n\pi}{2})}{n\pi} \exp\left(-(n\pi)^2 t\right) \cos(n\pi x) \tag{3.16}$$

This will be the manufactures solution we will test our simulations against for the verification of the RW implemetation.

We can now perform a convergence test to find the convergence rate for the random walkers. We will modify it slightly by testing for the number of walkers rather than the time step.

Using the maximum of the error measure already in use, we have tested the convergence rate measure for the following measures of Hc:

```
Hc = [200,1400,5600,10400,32000]
```

The convergence test suggests that the convergence rate for random walks follows the proportionality in equation 3.17. This relation tells us just what we have been expecting the whole time; while increasing the number of walkers will reduce the error, the convergence is slow. Should we wish to do so, we can force the error to $\mathcal{O}(\Delta t^2)$, but this will be extremely inefficient. In fact we can find the relation as $Hc \sim \Delta t^{-2}$ for $\epsilon \sim \mathcal{O}(\Delta t)$, and $Hc \sim \Delta t^{-4}$ for $\epsilon \sim \mathcal{O}(\Delta t^2)$. Clearly we will have enough trouble for the simpler cases.

$$err \propto Hc^{\frac{-1}{2}} \tag{3.17}$$

Figure 3.8: A convergence test for the isotropic random walk implementation using different conversion factors, Hc. The x axis (conversion rate) is log transformed and ranges from 100 to 50000 in real numbers.

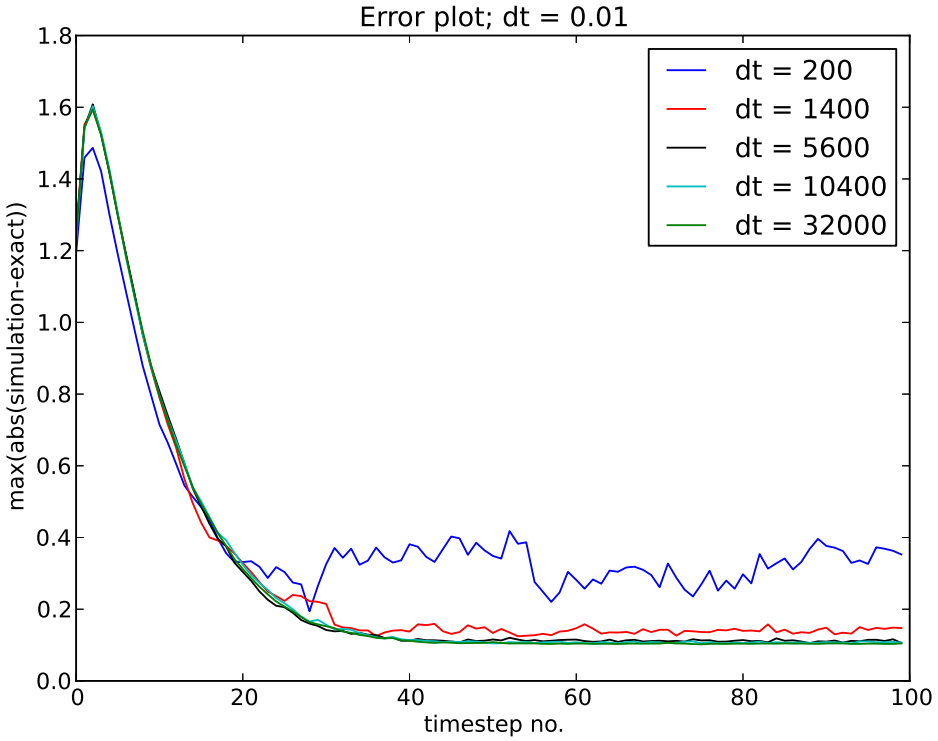


Figure 3.9: A "normal" error plot for the same simulation as in figure 3.8. The Δt in question is used to couple the RW simulation and the exact solution. For each Δt the RW simulation does 100 steps with a step length calculated from equation 2.45

3.4 Testing the combined solution

In this chapter we will combine the PDE-solution on some part of the mesh with the result from a random walk simulation using the same initial condition as the PDE-solver was given. As we will discuss in chapter **??** there are many candidates as to the combination of the solutions, but we will use one of the simplest ones; the average of the two. Before we start with the verification, however, we will take a quick look at a simplified problem which in principle is the same.

3.4.1 A simplified version of the algorithm

Monte Carlo methods are immensely important in modern computational science (*reference*), and can be used to solve integrals as well as random walks. As a simplified analogy to our method for solving the diffusion equation we ca look to the Ordinary Differential Equation (ODE) in equation 3.18.

$$\frac{\partial f}{\partial x} = a(x) \ , \ x \in [a, c] \tag{3.18}$$

Equation 3.18 is easily solvable (see eq. 3.19). For the sake of illustration we also specify that $a(x) = \frac{1}{x}$ and divide the integral in two parts, introducing $b \in (a, c)$.

$$f(x) = \int_a^b a(x)\, dx + \int_b^c a(x)\, dx \tag{3.19}$$

This is a case where we have complete control over all parts of the solution which is $f(x) = f(b) - f(a) + f(c) - f(b)$, and we can solve the two parts of the integral in two different ways; by the midpoint method and by MC integration. The convergence rates of these methods are 2 and 0.5 respectively. By the relation we found in chapter **??** (eq. **??**) the number of MC samples should be proportionate to the resolution used by the midpoint-rule to the power of four.

$$\frac{1}{\sqrt{N}} \simeq \Delta x^2$$
$$N \simeq \frac{1}{\Delta x^4} = N_x^4$$

The following output is from a program generously donated by Hans Petter Langtangen which does the required integration and calculates the convergence rates. It uses the relation described, but multiplies with a constant,

giving us

$$N = 2000 \times N_x^4$$

```
  N_x     N_MC      error        MC_error    MP_error
   1        2000 (    1)  2.650E-02   2.648E-02   2.391E-05
   2       32000 (    1)  7.392E-03   7.433E-03  -4.136E-05
   4      512000 (    1)  1.918E-03   1.927E-03  -8.954E-06
   8     8192000 (    8)  4.683E-04   4.866E-04  -1.832E-05
  16   131072000 (  131)  1.176E-04   1.220E-04  -4.320E-06

  Convergence rates
total  MP     MC
-1.84 -1.83   0.20
-1.95 -1.95  -0.55
-2.03 -1.99   0.26
-1.99 -2.00  -0.52
```

We see that the convergence rate for the whole integral is roughly 2 which is what we expect. This suggests that the idea behind the algorithm is sound. Another thing to notice it the convergence rate of the MC method which is sort off all over the place, this illustrates how difficult it is to verify the simulations. As the listed output and equation **??** shows, the number of walkers or MC samples grows very fast making it computationally very demanding to do the calculations.

3.4.2 Introducing walkers

First of all, using random walkers on parts of the mesh will have a considerable, negative impact on the error estimate. As we have discussed before, the solution from the random walkers will fluctuate around the "correct" (it is in fact correct while verifying) solution with amplitude proportional to $\frac{1}{\sqrt{N_{ij}}}$ which will depend on the PDE-solution in the mesh-point. It will also, as demonstrated in chapter 3.4.1, be possible to force the combined solution to have the desired properties in terms of error-estimates but at considerable computational cost. While doing the various error-testing for the combined solution we will therefore stick to the implicit scheme since we can choose the time-step more freely and by extension reduce the number of random walkers required.
Keeping in mind that the spatial error goes like Δx^2 we should at least choose the time-step so that $\Delta t > \Delta x^2$ to make sure the error from the time derivative is dominant. Starting off, we have a 2d simulation of isotropic diffusion with $\Delta t = 0.01$ and $\Delta x = \frac{1}{75}$ over the full interesting course of the solution,

meaning that the solution more or less at steady state the last time-steps. The error-plot from this simulation is shown in figure 3.10. We can see that although the error-term from the combined solution does not converge onto the error-curve from the deterministic solution, the error term is of the same order.

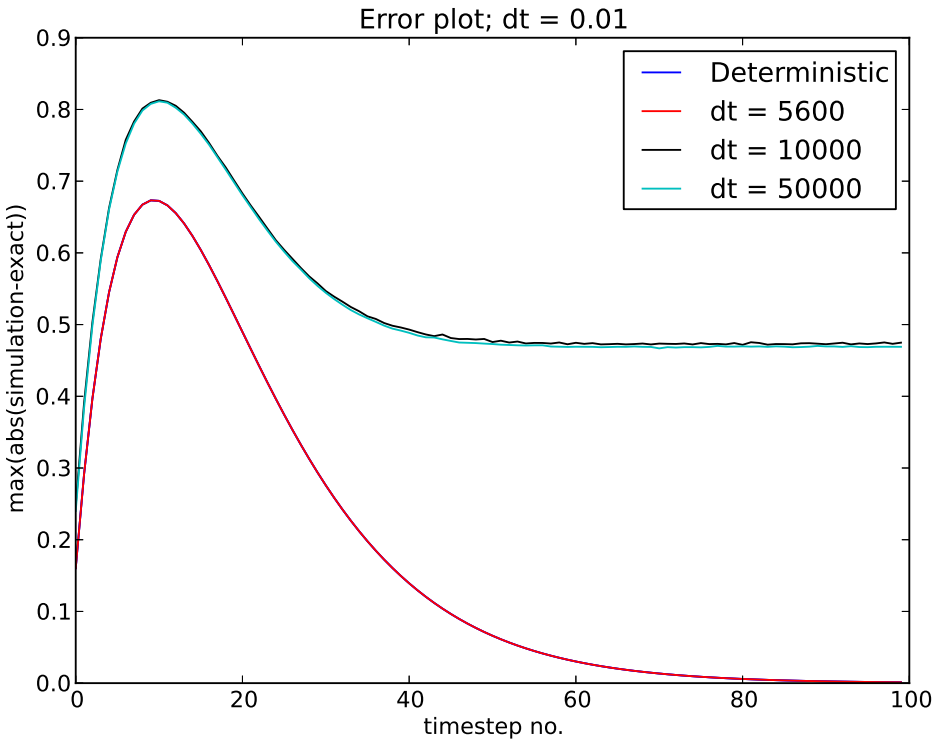


Figure 3.10: Error-plot of the combined solution using an increasing number of walkers. Parameters of importance are $\Delta t = 0.01$, $\Delta x = \frac{1}{75}$. Simulations doen with the BE scheme for 100 steps.

We then introduce an area on the domain where we switch models from the normal PDE to an average of the PDE solution and the result of a random walk simulation where the initial condition is the last time step from the PDE converted to walkers by the conversion rate given in equation 3.20. In this case we have used the parameters $a = 3$, $\Delta t = \frac{\Delta x^2}{3.0}$, $\Delta x = \frac{1}{20}$. These parameters makes one unit of $u(x,t)$ equal to some 1000 walkers.

$$C_{ij} = \frac{a}{\Delta t} U_{ij} \tag{3.20}$$

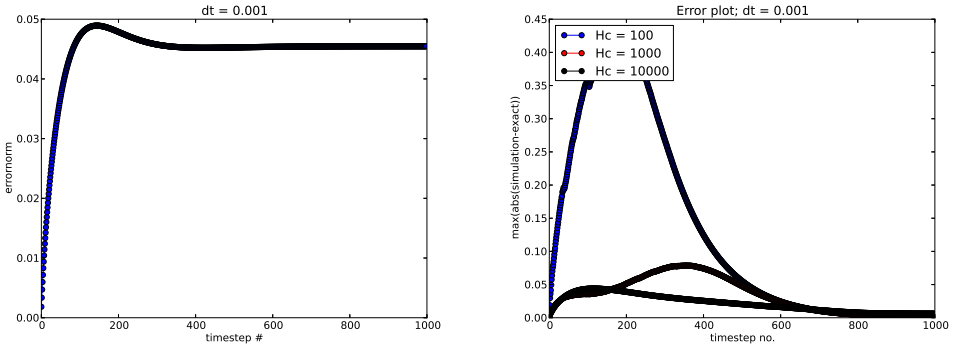
Mostly we will rewrite equation 3.20 to just one conversion factor times

the PDE solution, giving us some flexibility should we want to add more dependencies in the conversion. As of now, the conversion factor, Hc, is defined in equation 3.21. One "unit" of U_{ij} will directly correspond to Hc random walkers.

$$Hc = \frac{a}{\Delta t} \implies C_{ij} = Hc \cdot U_{ij} \tag{3.21}$$

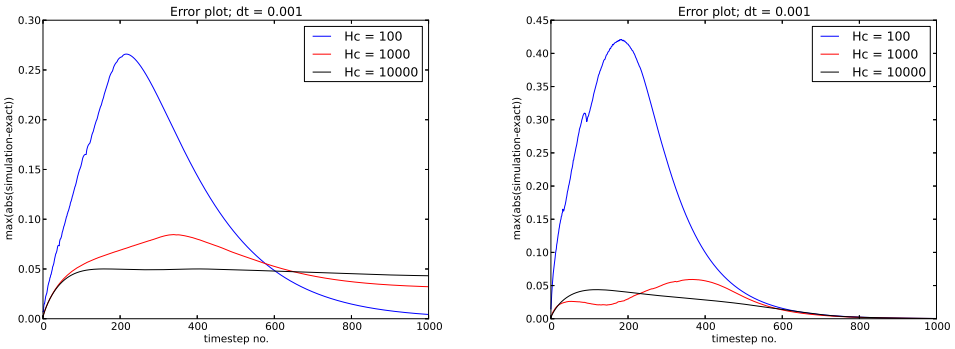
3.4.3 Increasing the time step and the relative size of walk-area

Now that we have an estimate of how to adjust the step length of the walkers in order to adjust for the time step, Δt, on the PDE level we would like to investigate the actual effects of running the simulation with a larger time step to verify our calculations. First off all, figure 3.11a shows the error norm of a simulation of the simplest diffusion equation 2.13 discretized by the Backward Euler scheme **??** using a time step which would make the Forward Euler discretization unstable (There is something strange about its convergence). Figure 3.11b shows the same simulation for various conversion parameters for the random walk. These simulations have input from the random walk model on some 20% of the mesh points. As a comparison we can turn to figures 3.12a and 3.12b which have 5% and 35% of the mesh points affected by walkers.



(a) (b)

Figure 3.11: Numerical error for 1D Backward Euler discretization of the PDE. In figure b there has been added walkers to the solution in the area $x \in [0.5, 0.7]$.

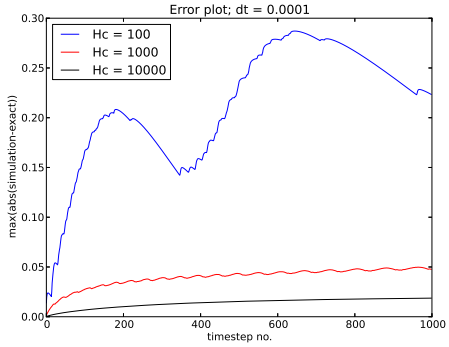


(a) Having walkers on 5% of the mesh points.

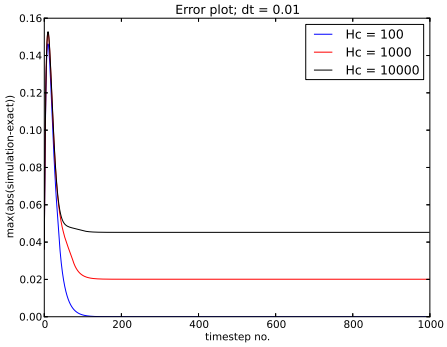
(b) Having walkers on 35% of the mesh points.

Figure 3.12: The effect of increasing the size of the walk area for a fixed $\Delta t = 0.001$ using the BE discretization.

These experiments have been done using the Backward Euler discretization so that we can simulate for a longer time and still see the effects to their full extent. We have also investigated the effects of changing the time step (also using the BE discretization to avoid instabilities and be able to do longer simulations). The results are summarized in figure 3.13. We notice something a bit unexpected in figure 3.13b. Unlike almost all the other comparable plots, it seems that using the least amount of walkers gives the best result here. This might be because the system quickly reaches its steady state, and will then be very well described by the continuum model. Having a small conversion factor, Hc, will mean that very quickly there will be no walkers which sort of ruins the point. This particular equation has steady state $u(t \to \infty, x) = 0$, and so not having any walkers will be perfect. What we should read from this figure is rather that the simulation with the most walkers converges to an acceptable error, and that this is achieved just as fast as for the other two simulations.

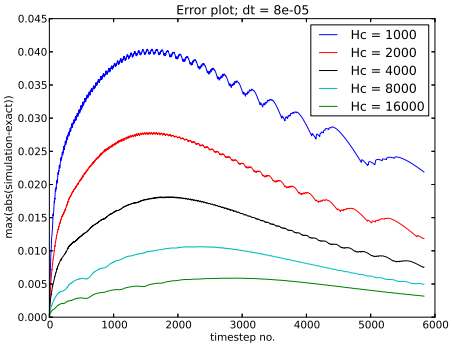


(a) Using a normal $\Delta t = 0.0001$.

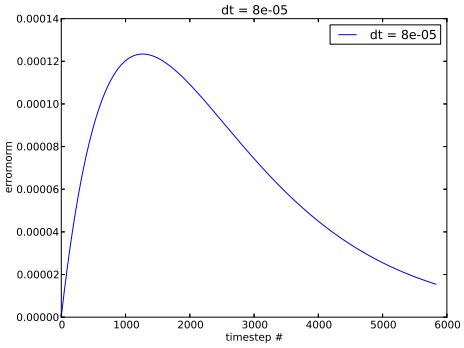


(b) Using a large $\Delta t = 0.01$.

Figure 3.13



(a)



(b)

Figure 3.14: The deterministic error and the error from simulations with walkers for long simulations.

Chapter 4

Software

4.1 About

The current version of the software (14.02.14) works in the following way. Various parameters are specified in a python-script which calls the program. After initialization, where anisotropic diffusion constant, initial condition and areas of combined solution are set up we start solving the equation (using the implicit BE scheme). At each time-step we call the Solve-method of the combine-class which in turn calls the solve-method of the PDE-solver, calculates the number of walkers by the conversion-rate, gives them a random position in proximity to the coarse-grained PDE-mesh-point we are looking at, and writes this position to a file. The random walk program is then called with some parameters. This program reads the file with walker-positions, advances some specified number of time-steps and writes the final positions to the same file it read. The main program now reads this file, maps the walker-positions back to the coarse-grained PDE-mesh and converts the spatial distribution of walkers to a concentration distribution. Here we are faced with a few choices with respect to the combination of the two solutions we have for the same area:

- Some form of least squares fitting could be done. A polynomial regression was developed, but tests indicate that it is not a good solution.
- Cubic spline interpolation might be slightly better seeing as we force the derivatives to be equal at the end-points. Interpolation forces the fitted curve to take the measured values at the interpolation-points, and so we must chose only some of the points to be used by the interpolation in order for there to be a difference. Immediately we are faced with the problem of which points to use and which to throw away. If we consistently chose the points which are closest to the PDE-solution (and the end-points which must be included for smoothness) we might as well not include a random walk model, and vice versa.
- We could use only the result from the random walk model
- Some sort of average might also help us. There are many to chose from, both arithmetic and geometric and with different kinds of weighting.

Intuitively, I find it better to use either an average or only the random walk result. At present (14.02.14) the simplest arithmetic mean is implemented at each mesh-point.
After each time-step the program also writes the combined solution on the coarse-grained mesh to a unique file.

The python script also does some other more or less clever stuff at each run. This is described in the appendix as a form of debugging A.2.

4.1.1 Limitations

As with any software there are limitations. The limitations discussed in this chapter will be with regard to physical problems, and not memory or CPU limitations which are described later.
Assuming the dendritic spines stuff will be the final application:
First of all, to specify a source function, one must program it in the Diffusion class (Diffusion::f()). This might be fixed in a later version using inheritance.
The geometry is, currently, limited to a square. Of course we can argue that we have scaled the size to a unit square, but as discussed in chapter **??** the actual geometry remains a unit square. Furthermore the same issue arises on the random walks.
As of now, it is unclear if 3d-modeling is supported. This is actually a smaller issue since there is usually little more to learn from switching from 2d to 3d. In any case, the extension to 3d is most likely not very hard, provided we use an array of slices from the cubic matrix describing the solution.

4.2 Adaptivity

There are two adaptive parts of the software. First of all, the number of walkers which depend on the concentration, or the solution the the PDE in the relevant area. This must change in order to keep physical meaning and give results. Without this adaptivity, the results would either be wrong, or the model would not make physical sense.
Since a diffusion process in general has rapid changes in the beginning where for example high frequency variations are dampened and very slow convergence to a steady state later, we have introduced a test of the amount of change between two subsequent time-steps. If this amount is smaller than some limit, we will increase the time-step.
This increase should be done in a more elegant manner(linearly?)

4.3 Computational cost

This chapter will consider the expensive parts of the code and look at possible improvements.

4.3.1 Memory

The memory-expensive parts of the code include storing the decomposed matrices, and storing the random walkers. None of these pose any problems.

4.3.2 CPU time

The program as it stands now (v 1.∞) uses the BE discretization and a highly specialized tridiagonal solver. The random walk-part of the software has been excreted as a stand-alone program which communicates with the main-program through a binary .xyz file containing the positions of all walkers in 3d space. This makes it easier to change solver, and implement more advanced solvers like the Direct Simulation Monte Carlo (DSMC) Molecular Dynamics (MD) simulator written by Anders Hafreager (see chapter **??**). There are three expensive operations in the algorithm as it stands now with the BE discretization using standard LU-decomposition.

- Random walks are expensive if there are many walkers. The number of calls to the random number generator follows eq. 4.1 and for the verification process, which required a lot of walkers, this represented a considerable cost. As an additional problem we will encounter some overhead upon calling the program, initializing variables and instances and so on. The computationally most demanding function seems to be the round-off function, which is used to place the walkers on the coarse-grained PDE-mesh.

- Communicating the positions of all the walkers between the two programs each time-step is very costly.

- Translating the positions of the walkers from the unit-square they are walking on to the coarse-grained PDE grid requires calling the "round" function from the math library in C++. This function is rather slow, and the program suffers from it.

$$N_{\text{calls}} \propto Hc\frac{(x_1 - x_0)}{\Delta x}\frac{(y_1 - y_0)}{\Delta y}\tilde{T} \tag{4.1}$$

where $\tilde{T}$ denotes the number of timesteps on the PDE level times the number of time-steps one PDE-step corresponds to on the RW level. Note that this expression will NOT be zero in 1d, and that it is dependent on the PDE-solution.

4.3.3 Parallelizability

In the final algorithm there are the following stages

- Initialization
  Read parameters from command-line, initial condition and diffusion "tensor" from file. Setup instances of solvers etc. Practically no point in parallelizing this.

- LU-decomposition
  The actual LU-decomposition is a sort of Gaussian Elimination which is costly ($\mathcal{O}(N^3)$). Although the decomposition is pre-implemented at this point, the plan is to implement my own version of this. This step should be possible to parallelize.

- Solving
  This step includes a back-transform of the decomposed matrix which is expensive for $d > 1$. This step should be parallelizable. It also includes the random walk part which is both expensive (depending on the number of walkers left) and highly parallelizable. We also write stuff to file which is quite costly. This is probably not possible to parallelize.

Parallelization of the random walk solver should scale linearly because the only form of communication required is shared memory. The LU decomposition and back-substitution require some communication and will not scale linearly, but will still benefit from parallelizing.

4.3.4 Some fancy title about changing stuff

It should be rather simple to replace parts of the code as long as certain conditions are met. Perhaps the simplest part to replace is the random walk part. Requirements for this part are:

- Locating executable main-file of the program in the folder "stochastic" and naming it "walk_solver".

- This executable should read the filename of the ini-file containing the positions of all walkers , and the local diffusion "tensor" in the relevant area.

- Upon completion, all positions must be written to the same file.

The PDE-solver should also be rather simple to replace, but some more programming will be required. First of all your solver must be included in the header-file. Next, the "Combine" class has an instance of the PDE-solver which it calls the advance-method of at each time-step. Seeing as this method is most regularly named "solve" you will either have to rename the method or the call. There are really very few dependencies on the PDE-solver, seeing as it is mostly left alone, but in addition to being able to respond to function calls it must:

- Have its own Δt attribute named "dt".
- Work on "double**" data types for all spatial dimensions (or implement some form of workaround)
- Be able to respond to increasing the time-step. In practice this means that the solver should be implicit.

As discussed, the implementation of random walks on parts of the mesh will reduce the convergence-rate to 0.5, and so there are really only two reasons to implement a new PDE solver. The current one only implements Neumann boundaries, and consequently must either be modified or replaced in order to work with other boundary conditions. It also only works on a square mesh. As discussed in section **??** it will be immensely complicated to implement a grid transformation, and this is already implemented in most finite element software.

Chapter 5

Results

5.1 Validity of the model

As we saw in the introduction to the analysis chapter it is possible to introduce enough walkers so that there is no mathematical difference between the combined model and the PDE-model. For all practical purposes, however we will not use that many walkers, because it is not the reason we introduced them in the first place. The simulations done in the analysis-chapter are largely done for verification purposes (see chapter A.2 in the appendix). In other words, our model converges to the continuum model in the limit of sufficient walkers.

Appendix A

A.1 Various calculations

In this appendix some more tedious and rather boring, but no less important calculations can be found. We will also list some algorithms that are important, but not quite in the scope of this thesis.

A.1.1 Backward Euler scheme in 2D

Using the BE discretization on the simple 2D diffusion equation will yield the general scheme in equation A.1.

$$u_{i,j}^n = \underbrace{\frac{-D\Delta t}{\Delta x^2}}_{\alpha}\left(u_{i+1,j}^{n+1} + u_{i-1,j}^{n+1}\right) + \underbrace{\left(1 + \frac{2D\Delta t}{\Delta x^2} + \frac{2D\Delta t}{\Delta y^2}\right)}_{\gamma} u_{i,j}^{n+1} \underbrace{-\frac{2D\Delta t}{\Delta y^2}}_{\beta}\left(u_{i,j+1}^{n+1} + u_{i,j-1}^{n+1}\right) \tag{A.1}$$

This can, again, be written as a linear problem where the vectors are simply the matrices u^n and u^{n+1} written as column vectors. The matrix is written out for a 3×3 grid with no-flux Neumann boundary conditions in equation A.2. We see that it is a five-band diagonal matrix, and so the tridiagonal solver cannot be used in this case. It is fully possible to use for example a Gaussian elimination in order to solve this equation, but it will require $\frac{2}{3}\mathcal{O}(n^3)$ operations per time step, where n is the size of the matrix (in this case $n = 9$). Another way to solve this equation, and by extension use the BE scheme, is to use some form of sparse LU decomposition.

$$\begin{pmatrix} \gamma & -2\beta & 0 & -2\alpha & 0 & 0 & 0 & 0 & 0 \\ -\beta & \gamma & -\beta & 0 & -2\alpha & 0 & 0 & 0 & 0 \\ 0 & -2\beta & \gamma & 0 & 0 & -2\alpha & 0 & 0 & 0 \\ -\alpha & 0 & 0 & \gamma & -2\beta & 0 & -\alpha & 0 & 0 \\ 0 & -\alpha & 0 & -\beta & \gamma & -\beta & 0 & -\alpha & 0 \\ 0 & 0 & -\alpha & 0 & -2\beta & \gamma & 0 & 0 & -\alpha \\ 0 & 0 & 0 & -2\alpha & 0 & 0 & \gamma & -2\beta & 0 \\ 0 & 0 & 0 & 0 & -2\alpha & 0 & -\beta & \gamma & -\beta \\ 0 & 0 & 0 & 0 & 0 & -2\alpha & 0 & -2\beta & \gamma \end{pmatrix} \mathbf{u}^n = \mathbf{u}^{n+1} \tag{A.2}$$

When we try to implement Neumann boundary conditions for grids that are larger than 3×3 we come across a problem. Doing the matrix-vector multiplication in equation A.2 reproduces the BE scheme with boundary conditions perfectly. However, if we extend to a 4×4 grid using a matrix on the same form we will start producing equations which will not arise from the scheme. This is illustrated in eqs. A.3 and A.4. Moving the off-diagonal entries with α one more column to the right and left will solve the problem,

but this will force us to use some more general solver of a sparse linear system. All in all we will probably be better off using another scheme (at least in 2D). The first equation that arises from the BE scheme in 2D (where $i = j = 0$) is

$$u_{0,0}^n = \gamma u_{0,0}^{n+1} - 2\alpha u_{1,0}^{n+1} - 2\beta u_{0,1}^{n+1} \tag{A.3}$$

while the first equation produced by the linear system in the 4×4 case is

$$u_{0,0}^n = \gamma u_{0,0}^{n+1} - 2\alpha u_{0,3}^{n+1} - 2\beta u_{0,1}^{n+1} \tag{A.4}$$

which is an equation that will never be produced by the BE scheme. In the 3×3 grid-case the off-diagonal matrix entries with α are on the third column before and after the diagonal.

Moving the corresponding entries to the fourth column in the 4×4 case, and similarly to the n'th column in the $n \times n$ case will fix the problem, but also increase the complexity of the matrix seeing as it will be $n + 2$ band diagonal.
Extending the model to three spatial dimensions gives a very similar matrix to the 2d-case.

$$\begin{pmatrix} D_{00} & -2\beta I & 0 & -2\alpha I & 0 & 0 & 0 & 0 & 0 \\ -\beta I & D_{01} & -\beta I & 0 & -2\alpha I & 0 & 0 & 0 & 0 \\ 0 & \ddots & \ddots & 0 & 0 & \ddots & 0 & 0 & 0 \\ -\alpha I & 0 & 0 & D_{10} & -2\beta I & 0 & -\alpha I & 0 & 0 \\ 0 & \ddots & 0 & \ddots & \ddots & \ddots & 0 & \ddots & 0 \\ 0 & 0 & 0 & -2\alpha I & 0 & 0 & D_{n0} & -2\beta I & 0 \\ 0 & 0 & 0 & 0 & \ddots & 0 & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & 0 & 0 & -2\alpha I & 0 & -2\beta I & D_{nn} \end{pmatrix} \begin{pmatrix} u_{00k}^{n+1} \\ u_{01k}^{n+1} \\ \dots \\ u_{10k}^{n+1} \\ \dots \\ u_{n0k}^{n+1} \\ \dots \\ u_{nnk}^{n+1} \end{pmatrix} = \mathbf{u}^n \tag{A.5}$$

In equation A.5 I denotes the $n \times n$ identity, D_{ij} denotes the tridiagonal $n \times n$ matrix with entries similar to the ones in eq. A.1, and off-diagonal entries similar to the ones in eq. A.1. All 0's denote the $n \times n$ zero-matrix. The values α and β are the relevant coefficient matrices for the calculations in question. These will be diagonal as well (or simly numbers in the isotropic case). Note also that the vector entries u_{ijk}^{n+1} are column vectors, making the vector $\mathbf{u}^{n+1}$ have the shape $1 \times n^3$.

A.2 Debugging

In any project which involves programming one is bound to do some debugging. This project is no exception. Debugging can be extremely frustrating

because no-one sees all the hours that go into finding the bugs, only the ones that do not (when the bug is not fixed). This section will deal with some general techniques for debugging finite difference solvers and random-walk implementations and some special words on how to debug the software developed to combine the two solvers.

A.2.1 Compiler/syntax errors

If you are programming in a compiled language like fortran or C/C++ you will forget some syntax, or misspell it, use a compiler-flag that outputs as much info as possible to terminal (-Wall for the gnu compiler), and start with the errors you understand. If you are building a larger project which requires linking, remember that packages must be linked in the correct order. For example; the armadillo linear algebra library is backened by LAPACK and BLAS. Both these libraries must be linked as well and they must be linked in the following order:

```
g++ *.cpp -o myprog -O2 -larmadillo -llapack -lblas
```

Anything else will give very cryptic compiler errors.
If you are using an interpreted language like python or MatLab the interpreter usually gives extensive information about the errors you have done, read them thoroughly!

A.2.2 Segmentation faults

In an interpreted language you will be told exactly where and what is wrong, in compiled languages you will not unless you are using extensions that give you some more information like armadillo. Segmentation faults are often quite simple to find, and most compilers have some sort of debugger which can help you find them. The gnu-compiler has an environment called gdb in which you can run your program which will catch seg.-faults and tell you where they are. If you are using some advanced editor like qt creator you can also easily place breakpoints in your code where you can get information about the various variables, instances, attributes etc. of your code at the exact time of the break. You can also step through the code. thoroughly Some times though the thing that works best is to print things at various places. I like having the possibility that every function in my code can print its name when it is called. There are even some python modules which tells you where it was called from. This will make it very easy to find out when the code went wrong, and what function is the problem.

A.2.3 Finite difference methods

First and foremost: Have a correct discretization. There are (probably) webpages which can discretize your equation(s) for you, but it is almost always useful to do this by hand. It will help you in your further debugging. There is one very important rule in programming in general: "First make it work, then make right, then make it fast". For implementing FDMs this means that you should start coding as soon as you have a clear image of what to be implemented, and what dependencies are needed. You will need a well defined initial condition (preferably one where you have the exact solution of the equation) and boundary conditions before you start coding. Personally, I like starting with the simplest Dirichlet boundary conditions $u|_{\partial\Omega} = 0$ and make them work before I go any further. You should note, however, that implicit schemes will be greatly influenced by the choice of boundary conditions.
Visualization is invaluable during debugging, seeing as a plot will let you see when and where the error occurs. *Show some example* Most likely you will now have something wrong with your solution (if not, cudos). This is where you look over your discretization again to make sure that it is correct, and then look over your implementation to check that it actually does what you think it does. At this point I would like to introduce rubber-duck debugging which was invented by the C-developer Dennis Ritchie. The story goes that he would keep a rubber duck at his desk and whenever he was stuck, would describe the code in detail (what each statement did and was supposed to do) to the rubber duck. Asking questions often reveals a lot of information. Personally I like my rubber duck to challenge me, so I prefer to involve a friend, but the concept is the same.
When your code seems to reproduce the intended results it is time to start the verification. This is where we make an error estimate and do some numerical analysis (you should of course have checked for the numerical stability of your chosen scheme when you discretized it). Making sure your implementation is correct is a lot harder than it sounds, but there are a few points that should be fulfilled:

- Manufactured solution
  Find some function which fulfills the equation you are working with. Remember that you have a source term which can be whatever you want it to be at this point, meaning that you can more or less decide what solution you want to your equation.

- Stationary solution
  This boils down to energy-conservation. If the initial condition is a con-

stant, there should be no time-dependencies (assuming your boundary conditions match; an initial condition $u = 1$ with Dirichlet boundaries $u|_{\partial\Omega} = 0$ will not work), and the solution will be constant.

- Exact numerical solution
  For a fitting initial condition (and discretization) you will be able to find an exact solution to the discretized equation you are implementing. An example of this is found in chapter 3.2.3. Your scheme should reproduce this solution to more or less machine precision. Note that you might run into round-off errors and overflow here in some cases (again, see chapter 3.2.3).

- Convergence test
  The discretization that is implemented will have some error term dependent on a discretization parameter (usually Δt, Δx or some parameter h used to determine the other discretization parameters) to some power. This power will determine the convergence rate of the numerical scheme, and you should verify that your implementation has the expected convergence. A convergence test is another way of saying that reducing the discretization parameter should reduce the error by the expected ammount. For a first order scheme the error should be halved by halving the time-step where as a second order scheme will get a reduction of $\frac{1}{4}$ for the same havlving of the discretization parameter.

There are probably more ways to make sure that a finite difference scheme is working properly, but the ones listed will usually give a good implication.

A.2.4 Random walk and Monte Carlo methods

The main difference between debugging a MC based solver and a deterministic solver is the fact that you often do not have a clear idea of what the results of the intermediate steps should be. What you might know (or should know during development), however is the result of the complete MC integration, and some statistical properties of your random numbers. Using uniformly distributed random numbers will give you a certain mean and standard deviation, and a Gaussian distribution will give you another. You should check that the random number generator (RNG) you chose actually reproduces these properties to a reasonable precision. If you are working with random walkers it also helps to look at the behavior of a small number of walkers, to check that they behave more or less as you expect. One thing to look out for is the fact that a random walker in both one and two spatial dimensions will

fill all space given enough steps. Of course enough steps is infinitely many, but if you also implement reflecting boundaries and use some 4-5 walkers you will see a tendency after approximately 10^4 steps.
As we have discussed earlier the fluctuations in a MC-model are usually of a magnitude $\frac{1}{\sqrt{N}}$ this is also smart to verify.
Finally, you should absolutely have the possibility to set the random seed and check that two runs with the same random seed produces the exact same result and makes sure you are using a RNG with a large enough period. The xor-shift algorithm by Geroge Marsagla [] has a period of around 10^{48} which usually is more than adequate.

A.2.5 The developed software

For some 2 months while working with this project I got really good results which seemed to verify all the important parts of the theory. Unfortunately it turned out that, while individually both parts of the program did exactly what they were supposed to do (verified by various tests), the combination of the two parts was implemented wrong. What actually happened was a finer and finer round-off rather than taking some number of steps with random walkers and combining the two models. It turned out that I sent an empty array to the random walk class as a new initial condition for the current time-step.
The moral behind this little story is that you should make 100% sure that every part of every function you write does exactly what you think it does, and nothing else. Furthermore, if you rewrite your code, you should remove the old parts as soon as possible. If you use some kind of version control software, which you definitely should, you will have older versions saved in the version control anyways. Do not be nostalgic and simply comment out the old parts just in case something, this makes your code very messy, and leaves the possibility of something slipping past you.

Another point to be made is that it will probably be helpful to construct the different parts of your code in such a way that they can be run as independently of each other as possible. As an example, both the PDE-solver, its tridiagonal linear system solver and the spine object can with relatively small changes to the main-fine be run independently. This allows for easier testing of the various parts of the code, and makes it more likely that the code will be reused in other projects.

A.2.6 When you cannot find the bug

While debugging (or any other repetitive task involving your own work) it is remarkably easy to become blind to your mistakes. The psychology behind this is (probably) that you have a clear idea of what should happen in each statement, and so you read that in stead of what the statement actually says. When it comes to proof-reading you can supposedly read backwards word by word, but can you do something similar when reading code? While I have never tried reading my code backwards because a statement usually depends on the previous statement, I have tried doing hand-calculations for almost every statement. Although hand calculations do not always show where things go wrong, they point out what variable or array entry etc. is wrong, and so the previous calculations can be checked. For finite difference schemes one can reduce the number of spatial mesh points to something manageable like three or four, and then do the same calculations that you think the computer does. If the solution is a matrix you can pinpoint the invalid matrix-entries with this method.

Another very important point if you are stuck is to never use "nice" values. If a parameter is set to zero or one just because it needs to be something, the probability that a potential problem disappears because it cancels out increases dramatically. Similarly, never do matrix calculations for 3×3 matrices. Use 4×4 matrices instead. The reasoning behind this is that banded matrices might fool you on 3×3 matrices, making you think your problem is tridiagonal when it in fact is n-band diagonal for example.

Bibliography

[1] L Farnell and WG Gibson. "Monte Carlo simulation of diffusion in a spatially nonhomogeneous medium: A biased random walk on an asymmetrical lattice". In: *Journal of Computational Physics* 208.1 (2005), pp. 253–265.

[2] Charles Nicholson. "Diffusion and related transport mechanisms in brain tissue". In: *Reports on progress in Physics* 64.7 (2001), p. 815.