NTNU

Norwegian University of
Science and Technology

Monte Carlo Simulations of Diffusion MRI in Restricted Geometries

Ane Nordlie Johansen

Master of Science in Physics and Mathematics
Submission date: March 2017
Supervisor: Pål Erik Goa, IFY

Norwegian University of Science and Technology
Department of Physics

Summary

The main part of the thesis has been to implement an environment in Matlab, suitable for simulating diffusion MRI. The implementation consists of a simulation of self-diffusion, wall detection and interaction for the diffusing particles, and of applying a gradient to generate the MRI-signal. The part of the implementation that needed the most consideration was the part including the possibility of using obstacles with arbitrary shapes. The diffusion and wall interaction parts of the program has been tested out quite thorough and seems to work satisfying. The diffusion MRI signal is also tested, but should probably be investigated to some higher extent before trusting the results completely.

Sammendrag

Hoveddelen av oppgaven har vært å implementere et program i Matlab, som egner seg til å simmulere diffusjons MRI. Implementeringen består av simmulering av diffusjon ved hjelp av Monte Carlo metoder, deteksjon av hindringer og kollisjons-interaksjon, og inkludering av gradienter for å generere diffusjons-MRI-signalet. Den delen av implementeringa som var mest arbeidskrevende var å inkludere muligheten for deteksjon og kollisjon med hindere av variert fasong. Denne delen av programmet er mye testet, og ingen abnormalteter er oppdaget. MRI-delen er også testet, men ikke godt nok til å kunne brukes uten ytterligere testing.

Contents

Contents

Chapter 1

Introduction

Magnetic resonance imaging (MRI) is a widely used imaging technique in clinical settings. MRI has the advantage of being suitable for differentiating between soft tissues, and unlike many other imaging techniques, it does not require exposure of high energy radiation [8]. Diffusion MRI has existed since the 1980's and uses self-diffusion of water in the body/sample to probe the geometrical structure. This is possible because the geometrical structure influences the mobility of the particles, and therefore the apparent diffusion coefficient, which influences the MRI-signal [9]. The benefit of this comes from the diffusing particles' sensitivity to the environment, making the apparent diffusion coefficient sensitive to cell density and permeability [1].

For being able to access the benefits of diffusion MRI, the connection between the diffusive behavior and the signal has to be known. This is a complex connection, and much research has been done regarding this. A method for doing this is simulating the diffusion and MRI-sequence for a wide range of different environments and analyzing the resulting signal. This may be done using Monte Carlo methods, which are methods simulating statistical phenomenons using random numbers.

The purpose of this thesis is to establish an environment appropriate for this kind of simulations. That is, using Monte Carlo methods to implement a diffusion process, and add the possibility of including obstacles to disrupt free diffusion. The obstacles interactions is implemented with intentions of being able to use a large variety of different

shapes. The sequences needed for generating the MRI signal is also included, together with calculations of the apparent diffusion coefficient.

Chapter 2

Theory

2.1 Diffusion

Free Diffusion

Diffusion is transport of particles, and will be present in a fluid even if the fluid is in equilibrium. This is called Brownian motion and arises from the fact that there is a high occurrence of collisions between the particles in the fluid. This motion is described by the diffusion equation, which may be derived using the continuity equation given by [4]

$$\frac{\partial q}{\partial t} + \nabla \cdot \boldsymbol{j} = k. \tag{2.1}$$

Here, q is the concentration of the system in question, $\boldsymbol{j}$ is the flux in or out of the system, and k is a source or sink term, and is zero in this case. Including the diffusion coefficient D, a size closely related to a fluids viscosity, and describing the degree of mobility for the particles is done using Ficks law [3];

$$\boldsymbol{j} = -D\nabla q. \tag{2.2}$$

Substituting this into the continuity equation gives the diffusion equation

$$\frac{\partial q}{\partial t} = D\nabla^2 q. \tag{2.3}$$

This is a partial differential equation, and solving this for the case of free diffusion results in the Green function given by [5]

$$G(\boldsymbol{r}, \boldsymbol{r}', t) = \frac{\exp(\frac{-|\boldsymbol{r}-\boldsymbol{r}'|^2}{4Dt})}{(4\pi Dt)^{d/2}}. \tag{2.4}$$

Here, $\boldsymbol{r}$ is the final position and $\boldsymbol{r}'$ the initial position of a particle, d is the dimension of the system and t is the time. From the Green function, statistical moments describing the diffusion is found [5]. The first moment is the expectation value of the particles position, and is given by

$$\langle \boldsymbol{r} \rangle = \boldsymbol{r}'. \tag{2.5}$$

The second moment gives the expectation value of the squared position

$$\langle \boldsymbol{r}^2 \rangle = |\boldsymbol{r}'|^2 + 2dDt. \tag{2.6}$$

From this one can see that the mean square displacement a particle has moved during time t becomes

$$\langle (\boldsymbol{r} - \boldsymbol{r}')^2 \rangle = \langle \boldsymbol{r}^2 \rangle + \boldsymbol{r}'^2 - 2\boldsymbol{r}' \langle \boldsymbol{r} \rangle = 2dDt. \tag{2.7}$$

Non Free Diffusion

For non-free diffusion, that is, when the particles are either trapped inside something (restricted diffusion), surrounded by obstacles that they may collide with (hindered diffusion), or a combination of these, the diffusion coefficient will differ from the one discussed in the previous section and have a time dependence (except in some limiting cases). By investigating this new diffusion coefficient $D(t)$ (for short time behavior) one can in fact find characteristics for the environment the particle is located in, such as shape and characteristic lengths [6] .

Analytical results for the time dependence of $D(t)$ varies with respect to the diffusion length relative to the characteristic lengths of the surroundings, that is, it differs whether it is the long time behavior, the short time behavior or somewhere in between, that is in question. This may be explained by how each particle perceives the wall e.g. letting a particle diffuse inside a sphere of radius R for a time much shorter than

the time it would need to diffuse across the sphere, results in a case where the particle does not see the wall on the other side. However, if the particle diffuses for a much greater time, it will soon or later have been reflected from all directions, this lead to a case where the particle forgets its initial position. This long time limit leads to a time dependent diffusion coefficient decreasing towards zero by $\frac{1}{t}$. On the other hand, if there are holes in the sphere wall for the particle to escape through, making it connected with the outside, the diffusion coefficient will not approach zero [6].

For both restricted diffusion and diffusion inside a connected obstacle, the short time behavior of $D(t)$ may be described by [6] [7]

$$D(t) = D_0\Big(1 - k\frac{S}{V}(D_0 t)^{1/2}\Big) + \mathcal{O}(D_0 t). \tag{2.8}$$

Here $k = \frac{4}{9\sqrt{\pi}}$ for pulsed gradient sequences and $k = \frac{32(2\sqrt{2}-1)}{105\sqrt{\pi}}$ for constant gradients (see next section). $\frac{S}{V}$ is the surface to volume ratio, and t is the diffusion time.

2.2 Magnetic Resonance Imaging

MRI uses the protons' intrinsic property that it is a 1/2-spin particle. Due to this a proton in an external magnetic field $\boldsymbol{B_0}$ has two available energy states $\pm\mu_p B_0$, where μ_p is the magnitude of the protons magnetic moment. When the spins have reached equilibrium in the magnetic field, the lower energy state will be slightly more occupied, leading to an resulting magnetization vector $\boldsymbol{M}$. But, because the spins are not in phase, $\boldsymbol{M}$ is static aligned with $\boldsymbol{B_0}$ and no signal is present.

The spin's precession frequency is given by $\omega = -\gamma B_0$ and is know as the Larmor frequency, γ is the gyromagnetic ratio. To get the spins to be in phase with each other, a radio frequency (RF) signal with this exact frequency is applied. The fact that the spins are now precessing in phase, means the magnetization vector are tilted away from $\boldsymbol{B_0}$ and now rotates around this, and this rotating magnetization vector is the MR signal. Keeping the RF-signal on for the time it takes $\boldsymbol{M}$ to reach down to the plane perpendicular to $\boldsymbol{B_0}$ defines $T1$, which is an intrinsic property of the material being scanned.

Contrasts in MRI are detected from differences in the time it takes before the signal vanishes after the RF-signal are removed. In a $T1$-weighted image it is the time for the magnetization vector to relax from the plane perpendicular to $\boldsymbol{B_0}$ back to aligning with $\boldsymbol{B_0}$ (spin-lattice relaxation) that is weighted, while for a $T2$-weighted image it is the dephasing of the spins coherence in the plane (spin-spin relaxation)[8].

In a spin echo MR-sequence, another RF-signal is applied after letting the spins dephase for a time say t_{se}. This RF-signal is set to tilt the magnetization vector 180°, which results in an inversion of the dephasing caused by local magnetic field inhomogeneities [9]. In this way, a signal echo appears at time t_{se} after the 180° tilt. Without this second RF-signal, no echo will occur, and the dephasing then results in a signal called a free induction decay (FID).

Diffusion MRI

For a spin precessing with frequency ω, the phase at time t will be given by $\phi = \omega t + \phi_0$, where ϕ_0 is the phase at time $t = 0$, and in the MR-sequence, this is the same for all spins at the moment the RF-signal is turned off. Then, as mentioned in the previous section, the spins will start to dephase. In diffusion MRI, a magnetic gradient with duration δ is applied. This leads the spins at different locations to experience slightly different magnetic field strengths, which leads to variation in the frequencies and thus, dephasing. A time $t = \Delta - \delta$ after the first gradient is ended, a new gradient is applied, this gradient has the same duration and strength as the first one, but with opposite sign. So, if a spin had been positioned at the exact same location from the first gradient was applied, to the second gradient ended, the impact from the second gradient will be inverting the impact from the first on, and thereby the spins phase will be back at the initial state. But the spins do not stay a the same position, they move around, and the more they move, the more the gradients will lead to dephasing and signal loss. The result is that the signal will decrease with increasing diffusion, and therefore be dependent on the geometrical structure in the sample [6]. In a spin echo sequence the gradients are applied without sign change because of the 180° tilt.

For a spin assumed to be located at position x for a short time $\mathrm{d}t$, the phase change resulted from being exposed to a gradient G_x in the x-direction is given by

$$\phi(x) = -\omega(x)\mathrm{d}t = -\gamma(B_0 + G_x x)\mathrm{d}t. \tag{2.9}$$

If the gradient duration is short, the spins movement δ may be neglected, the phase change caused by the two gradients are then described by

$$\begin{aligned} \phi_2 - \phi_1 &= -\gamma\delta\big[B_0 + G_x x_1 - B_0 - G_x x_2\big] \\ &= -\gamma\delta G(x_2 - x_1). \end{aligned} \tag{2.10}$$

In this case, the total diffusion signal attenuation caused by a gradient G_x is found by integrating over the spins' movement and corresponding phase change. This is done using the initial spin density $\rho(x_i)$, the diffusion propagator $P(x_i, x_f, \Delta)$ (given by 2.4 for free diffusion) and the attenuation corresponding to a spin's movement from x_1 to x_2 given by $\exp(-i\gamma\delta G_x(x_2 - x_1))$. The expression for the attenuation $E(q) = S(q)/S_0$, where $q = \gamma\delta G_x$, $S(q)$ is the signal and S_0 is the signal in absence of magnetic gradients, then becomes

$$E(q) = \iint \rho(x_1)P(x_1, x_2, \Delta)\mathrm{e}^{-iq(x2-x1)}\mathrm{d}x_1\mathrm{d}x_2. \tag{2.11}$$

For free diffusion, this results in a Gaussian phase distribution

$$E(q) = \mathrm{e}^{-q^2 D\Delta}. \tag{2.12}$$

A generalization of this to include gradients with longer duration, introduced by Stejskal-Tanner, is given by

$$E(b) = \mathrm{e}^{-bD}, \tag{2.13}$$

where b is known as the b-value, and is given by $b = (\gamma\delta G_x)^2(\Delta - \delta/3)$ for a spin-echo sequence [9].

What size regarded as the diffusion time in a diffusion MRI-sequence is somewhat ambiguously, Δ, $\Delta + \delta$ and $\Delta - \delta/3$ is used [11]. Though it seems quite settled to use $\Delta - \delta$ for the pulsed gradient cases.

High b-Values

For high b-values the signals behavior will greatly deviate from the Gaussian form and singularities occurs. A complete analysis of this is given in [10], here only the parameters and equations needed to investigate if a signal fulfill this are cited.

Defining t_D as the time needed to diffuse the distance R, that is $T_D \equiv \frac{R^2}{D}$, and t_C as the time the signal needs to dephase in the case of no diffusion $t_c \equiv \frac{1}{\gamma G R}$. Measuring the signal as a function of the dimensionless parameter $\tau = \frac{t}{t_C}$, the signal will depend on p given by

$$p = \frac{t_c}{t_d} = \frac{D}{\gamma G_x R^3}. \tag{2.14}$$

In this way, decreasing p will lead to increased number of singularities.

2.3 Reflection and Surface Detection

Given a particle with initial position $\boldsymbol{r}$ and a constant velocity $\boldsymbol{v}$, its trajectory is described by $\boldsymbol{r'} = \boldsymbol{r} + \boldsymbol{v}t$. If a sphere of radius R is present and centered in origin, the time it takes before the particle hits the sphere wall is found by realizing that a collision happens at the time when $\boldsymbol{r'}$ lies on a distance R from the origin, i.e. $R^2 = |\boldsymbol{r'}|^2$. Substituting this into the equation for the particle trajectory and solving for t gives the following equation for the collision time;

$$t_c = \frac{-\boldsymbol{r}\cdot\boldsymbol{v} \pm \sqrt{(\boldsymbol{r}\cdot\boldsymbol{v})^2 - \boldsymbol{v}^2(\boldsymbol{r}^2 - R^2)}}{\boldsymbol{v}^2}. \tag{2.15}$$

For intersection with a plane, the corresponding equation is given by [14]:

$$t_c = \frac{(\boldsymbol{p} - \boldsymbol{r})\cdot\hat{\boldsymbol{n}}}{\boldsymbol{v}\cdot\hat{\boldsymbol{n}}} \tag{2.16}$$

Here $\boldsymbol{p}$ is a point in the plane, and $\hat{\boldsymbol{n}}$ is the unit normal. If the actual obstacle is just a part of this plane, described by the corners of a lattice face (squared or other), the following test can be used to check whether the particle hits inside this face or not [13]

$$(\boldsymbol{p}_j - \boldsymbol{p}_i) \times (\boldsymbol{r}_c - \boldsymbol{p}_i)\cdot\hat{\boldsymbol{n}} \geq 0. \tag{2.17}$$

$\boldsymbol{r}_c$ is here the point where the particle hits the plane, and $\boldsymbol{p}_i$ is the faces' corners. Labeling the points counter clockwise from one side of the face, gives clockwise labeling on the other side. This test is valid in both cases, as long as the normal used is the one defined outwards to the counter clockwise labeling side for both. This test has to be done between all the corners, with cyclic permutation. When a particle collides with a wall, resulting in an elastic collision, the new velocity is given by mirroring the initial velocity vector around the outward pointing surface normal in the collision point. From Fig. 2.1 it is easy to see that the velocity after the collision is given by $\boldsymbol{v'} = v_x\hat{\boldsymbol{x}} - v_z\hat{\boldsymbol{z}}$, where v_x and v_z are the x- and z-components of the velocity before the collision. This expression can be done independent of the coordinate system recognizing v_z as $\boldsymbol{v}\cdot\hat{\boldsymbol{n}}$ and $v_x = |\boldsymbol{v}||\hat{\boldsymbol{n}}|\sin(\theta) = |\boldsymbol{v}\times\hat{\boldsymbol{n}}|$. Noticing $\boldsymbol{v}\times\hat{\boldsymbol{n}}$ points in the negative y direction this is forced in the correct direction by crossing it with $-\hat{\boldsymbol{n}}$. Following, the new velocity can be expressed $\boldsymbol{v'} = (\boldsymbol{v}\times\hat{\boldsymbol{n}})\times(-\hat{\boldsymbol{n}}) - (\boldsymbol{v}\cdot\hat{\boldsymbol{n}})\hat{\boldsymbol{n}}$. Using suffix notation [15] and rearrange the double cross product may be simplified to

$$\begin{aligned}(\boldsymbol{v}\times\hat{\boldsymbol{n}})\times\hat{\boldsymbol{n}} &= \epsilon_{ijk}(\boldsymbol{v}\times\hat{\boldsymbol{n}})_j n_k = \epsilon_{ijk}\epsilon_{jlm}v_l n_m n_k = \epsilon_{kij}\epsilon_{jlm}v_l n_m n_k \\ &= (\delta_{kl}\delta_{im} - \delta_{km}\delta_{il})v_l n_m n_k = v_k n_i n_k - v_i n_k n_k \\ &= (\boldsymbol{v}\cdot\hat{\boldsymbol{n}})\hat{\boldsymbol{n}} - \boldsymbol{v}.\end{aligned} \tag{2.18}$$

Here ϵ is the alternating tensor and δ_{ij} is the Kronecker delta. Now substituting this into the expression for $\boldsymbol{v'}$ gives

$$\boldsymbol{v}' = -((\boldsymbol{v}\cdot\hat{\boldsymbol{n}})\hat{\boldsymbol{n}} - \boldsymbol{v}) - (\boldsymbol{v}\cdot\hat{\boldsymbol{n}})\hat{\boldsymbol{n}} = \boldsymbol{v} - 2(\boldsymbol{v}\cdot\hat{\boldsymbol{n}})\hat{\boldsymbol{n}}. \tag{2.19}$$

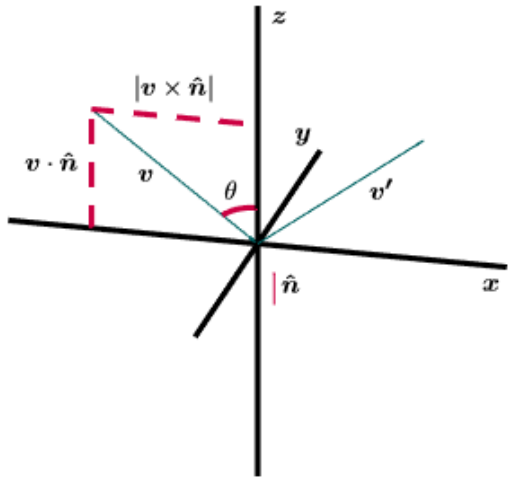


Figure 2.1: A particle with initial velocity $\boldsymbol{v} = v_x\hat{\boldsymbol{x}} + v_z\hat{\boldsymbol{z}}$ is reflected elastically by a plane overlapping the xy-plane.

Chapter 3

Method

The simulations are done using Matlab R2014a on a Linux cluster and most of them are done in parallel using four workers. The input parameters are the diffusion coefficient D_0, the gyromagnetic ratio γ, the number of spins N, the gradient strengths G and the total simulation time T or the gradient time specifications δ and Δ.

3.1 Random Walk

The Brownian motion is modeled using an uncorrelated and unbiased random walk. That is, a random walk generated by letting the particle's movement be determined by a sequence of uncorrelated steps with equal probability for going in any direction. From the central limit theorem one gets that the particle movement will lead to a Gaussian with increasing time [2].

Using a Monte Carlo simulation, the random walk is implemented directly in 3d, using a constant step size and a randomly chosen direction. This method leads to a Gaussian distribution due to the Central Limit Theorem [12], and only requires two generated random numbers per step, opposite to generating a random step length in three directions. The step size is given by $dr = \sqrt{6Ddt}$ in accordance with Eq.2.7. For the direction, the azimuthal angle ϕ is determined by a random number between zero and 2π, while the polar angle θ is determined by giving $\cos(\theta)$ a random number on the interval $(-1, 1)$. This comes

from the fact that the infinitesimal surface element for a sphere is given by $dS = d\cos(\theta)d\phi$, and is therefore the way of getting a random distribution over the sphere. It should be noted that the random number generator used here is Matlab's rand(). This function returns numbers in an open interval, and for spherical coordinates it is only the azimuthal angle's upper limit that should not be included in the interval. Though, the results does not seem to be significantly affected.

3.2 Detection and Interaction With Obstacles

Wall Detection for Spherical Cell

To investigate the diffusive behavior of spins inside a spherical cell with impermeable walls, a program is set up with a sphere of radius R centered in origin. The wall detection is done by checking whether the new position generated are more than a distance R from origin, if so, the collision time t_c is found by Eq.2.15. To avoid doing this check for each step during the diffusion, the shortest distance between the particles position and the sphere wall is found. Dividing this distance on the step size gives the number of steps the particle can take before a new collision check/distance calculation is necessary.

Detection of Grid Based Obstacles

This implementation is set up for handling particle interaction with obstacles of various shapes. To achieve this, the obstacles have to be defined by grids, and the grid points have to be sorted in a consistent way, independent of shape. The choice of set up for these points are the same design as Matlab's function sphere(). That is, they are closed, and defined by the equal sized matrices X, Y and Z, where each point in the grid is defined by the vector $X(i,j)\hat{\boldsymbol{x}} + Y(i,j)\hat{\boldsymbol{y}} + Z(i,j)\hat{\boldsymbol{z}}$. All the grid faces are squares, except at the top and bottom of the obstacle, where they are triangles (see Fig. 3.1). For the grid indices (i,j), the first one specifies the "height" on the grid, the maximum and minimum of this are at the end points of the obstacle i.e. the point where the triangles in the top and bottom of the obstacle are connected. The second index specifies the polar angle and here the minimum and maximum index

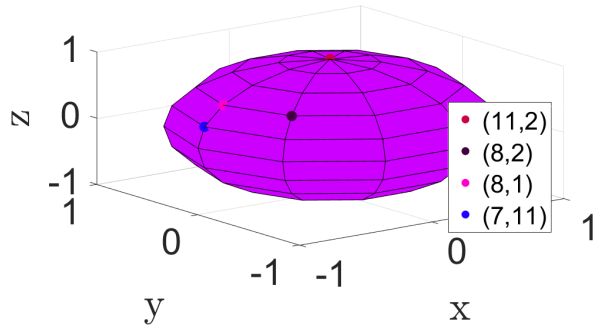


Figure 3.1: Overview over how the indices are sorted in matlab's sphere function, here called with input parameter 10.

overlaps, this is shown in Fig. 3.1. To check for wall interactions, a function is implemented with the purpose of looking through all grid points, identifying the indices belonging to the three or four grid points that most likely belong to the nearest face, and sort these to be in counter clockwise direction seen from outside the obstacle. The reason this is not guaranteed to be the closest face is due to the possibility of differences in the face sizes as shown in Fig. 3.2, and the sorting is needed for the collision check as mentioned in the theory section (Eq. 2.17).

Having found this face, the collision check is done (this is describe below). If the particle do not collide, another function is called, which finds the indices defining all the neighboring faces to the original one. If the face is a square there are eight neighbors, and if it is a triangle the three neighboring squares and all the triangles on that side of the obstacle are included. These neighboring points are also sorted in counterclockwise direction, and the check for collision. Including this neighbors are crucial for avoiding holes in the obstacle.

The function generating the collisions uses the three first grid points to set up two vectors $\boldsymbol{v}1$ and $\boldsymbol{v}2$ from the first to the second point and from the second to the third point respectively. Crossing this gives a surface normal directed out from the obstacle. Then using that the particle has velocity $\boldsymbol{v} = d\boldsymbol{r}/dt$, the collision time is computed using

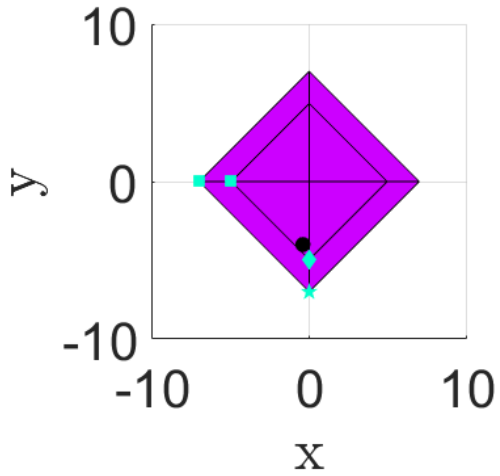


Figure 3.2: An obstacle seen from above with all eight faces located in the xy-plane. A particle (the black dot) lies above a triangular face on the obstacle, so this is the closest face. The first grid point the function will find is the blue diamond, and this belongs to the correct face, but the next closest is the point marked with a blue star, and this does not belong to that face. So is this case, the face found (fulfilled on the figure with the two blue squares) would not be the one with nearest location to the particle.

Eq.2.16.

Because $\hat{\boldsymbol{n}}$ appears both in the numerator and the denominator it is not a problem that the normal is pointing out from the obstacle even though the particle is inside. If t_c is positive, and less than dt, the particle will reach the plane defined by $\boldsymbol{v}1$ and $\boldsymbol{v}2$. If so, the program checks if the particle actually hits the plane inside the face in question. This is done using the test given by Eq.2.17, three tests for the triangles and four for the squares. This test holds for particles reflecting both on the inside and the outside of the obstacle because the normal is pointing out from the obstacle in both cases, while the grid points is seen to be in clockwise order for a particle inside and counterclockwise from outside.

Wall Interaction

The collision points are found letting the particle move with its original velocity for the time t_c and the reflection is done by mirroring this velocity around the surface normal using 2.19. For the case inside the sphere, the surface normal is given by a vector starting in the collision point and pointing towards origin. For the grids one has to use the outward pointing normal seen from the particles position, so before computing the new velocity, the normal changes sign if $\hat{\boldsymbol{n}} \cdot \boldsymbol{v} > 0$. The new position is found by setting the length of the new velocity equal the old, and let the spin move $\boldsymbol{v}t_c + \boldsymbol{v}'(dt - t_c)$. Also, there is a while loop allowing multiple collisions in one time step, in that case $dt' = dt - t_c$ is used instead of dt. This multiple collision check is important to avoid significant errors [12].

Initializing Positions

To give the particles random initial positions inside the sphere, a random number between zero and the sphere radius is generated together with a random direction found in the same way as for the direction in the random walk. A cylinder is used to test the case with grid based obstacles, and for this, the initialization is done setting the z-position by a random number between the minimum and maximum z-value, while x and y is set between plus and minus the maximum radius (the variation in radius comes from that the cylinder is set up by a grid so it do have edges). Though, this may result in a position outside the cylinder, so this is tested and new positions are generated until all the particles are located inside the cylinder.

3.3 Gradient and Signal

Using the equations regarding diffusion MRI given in the theory section, the phase shift of one particle located at position x during time $\mathrm{d}t$, and exposed to a gradient G_x is given by $d\phi = \gamma \mathrm{d}t G_x x$. In the implementation, all x-positions are saved to an array during the random walk, and when all positions for one particle are generated, all the phase shifts are calculated. Simulating a FID signal corresponds to compute

the phase shift for all the x-positions using the same positive gradient. A pulsed gradient spin echo sequence is achieved by using $+G_x$ on the first x-position, and $-G_x$ on the last one. The constant gradient spin echo comes from including the δ/dt first and last x-positions instead of only the first and last one. The final phase for each particle is then found by summing up all the phase shifts, and the total signal generated from N particles is then calculated using

$$S = \sum_{i=1}^{N} \mathrm{e}^{i\phi_i}. \tag{3.1}$$

Several gradients, δ and Δ are set in the same simulations. For the time parameters this is done by only using fractions of the position array for the shorter sequences. Further, the phases are saved in a matrix keeping track of which parameters belonging to each signal. The signal densities are found by taking the absolute values of these results and normalized by dividing on the total number of particles.

3.4 Simulations

All the simulations are done using the time step $\mathrm{d}t = 0.001$ ms, and the gyromagnetic ratio for protons is given by $\gamma = 2.675 \cdot 10^8$ rad/(Ts). For the MR simulations the gradient strengths are mainly chosen to ensure signals with b-values in the range $10\ \mathrm{s/mm^2} < b < 800\ \mathrm{s/mm^2}$, for the pulsed gradient this means using values way too high to be relevant in an actual MR-scan. Considering the evaluation of the results, the b-values belonging to each signal are found using $b = (\gamma\delta G_x)^2(\Delta - \delta/3)$, and the ADC comes from solving Eq. 2.13 with respect to D. Using Eq. 2.8 and plotting the ADCs as funcion of the square root of the diffusion time t, the surface to volume ratio, and hence also the sphere radius are found from the resulting slope.

Testing Free Diffusion and Gradient

The free random walk is tested with the diffusion coefficient $D = 1$ $\mu\mathrm{m}^2/\mathrm{ms}$ which is taken from [16], where it is used for modeling extracellular water. The total diffusion time is $T = 100$ ms, and the

simulation is done for 10 000, 50 000, 100 000 and 400 000 particles. The results are then compared to the statistical properties given in the theory section. To ensure correct behavior in all directions separately, histograms for the particle densities in the x-, y-, and z-direction are compared with Eq. 2.4 in one dimension. This is also done because it is tricky to get a nice histogram for the density in three dimensions around origin due to the $\frac{1}{r^3}$-factor appearing from the volume scaling in the density calculation. To check the statistical properties of r, the distribution of the final positions are compared with the expected curve given by $\rho(r) \times r$, here $\rho(r)$ is the density function given by Eq. 2.4.

The signal/gradient-part of the program is tested by applying a pulsed gradient to the free diffusion. 100 000 particles are set to diffuse, varying the simulation time from 30 ms to 200 ms, and the gradient from 26434 mT/m to 610480 mT/m, using the diffusion coefficient $D_0 = 1.65\ \mu\text{m}^2/\text{ms}$.

Spin Echo Simulations of Spins Inside Sphere

Based on the results from the random walk, 100 000 particles are used in the simulations. The environment is inside a sphere with radius 75 μm with impermeable walls. The intrinsic diffusion coefficient is set to $D_0 = 1.65\ \mu\text{m}^2/\text{ms}$. This situation is chosen in terms of being able to compare the simulation results to the results given in [18], where diffusion MR-measurements using a rutabaga where performed. Four different setups for the spin echo sequences are used, pulsed gradient, constant gradient with no break between the positive and the negative part, constant gradient with a 5 ms break, constant gradient with varying gradient and break duration and one with a constant positive gradient over the whole simulation time.

The pulsed gradient is given the duration $\delta = 0.001$ms corresponding to adding the gradient over only one time step, this is done to let the gradient approach a Dirac delta function and gives $T \simeq \Delta$. In this case applying the gradient and adding up the phase is considerably less time consuming than for the constant gradient. Therefore the number of different diffusion times N_T and gradient strengths N_G is both high for the pulsed gradient. For the constant gradient two simulations are done, one with small N_T and high N_G, and one opposite. This was

done to get a decent amount of points both for the signal as a function of b, and for the ADC.

For the case of both varying gradient duration and varying break between the positive and negative gradient, δ and Δ are chosen in terms of being able to compare the results with the experimental results given in [18].

FID Simulation of Spins Inside a Sphere

A FID simulation is done to investigate the behavior of the signal for high b-values. This is only done for 10 000 particles due to the long diffusion time (17500 ms) needed to achieve results easily compared to [10] without changing the environment considered in the previous section, and because of the high number of measurement points needed to get a decent result around the singularities.

Chapter 4

Results

4.1 Random Walk

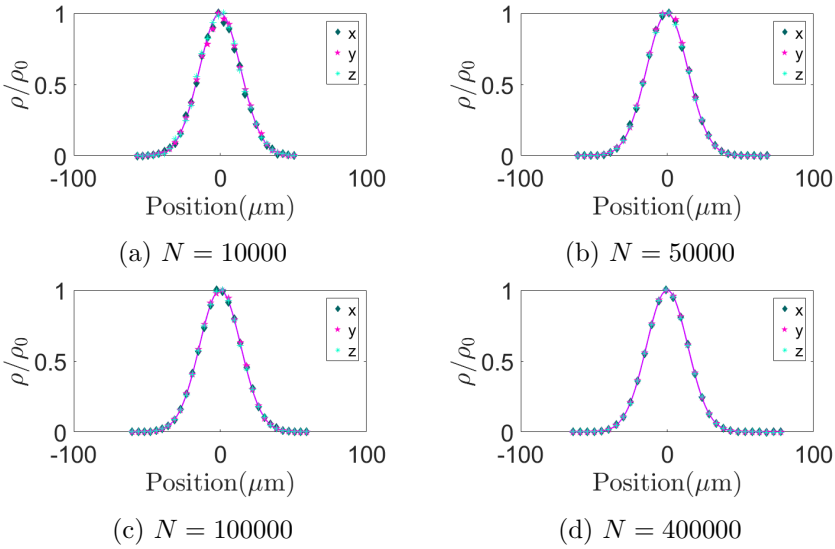
Fig. 4.1 shows the particle density after 100 ms, for 10 000, 50 000, 100 000 and 400 000 particles. They are compared to the probability density in one dimension given by Eq. 2.4. In Fig. 4.2 the distribution of the final positions is shown, and compared to the expected statistical curve given by $\rho(r) \times r$. The actual values for the mean position, standard deviation and kurtosis, which is the fourth statistical moment, is given in Table 4.1.

4.2 Reflection Inside a Non-Spherical Obstacle

To test the reflection for the non-spherical obstacles, 10 particles are set to diffuse inside and outside a cylinder, this is shown in Fig. 4.3. As seen, the cylinders shape appears even when the cylinder itself is not plotted.

4.3 Pulsed Gradient on Free Diffusion

Spins experiencing free diffusion are exposed to a pulsed gradient spin echo sequence. The simulation includes 59 different gradient strengths



(a) $N = 10000$ (b) $N = 50000$

(c) $N = 100000$ (d) $N = 400000$

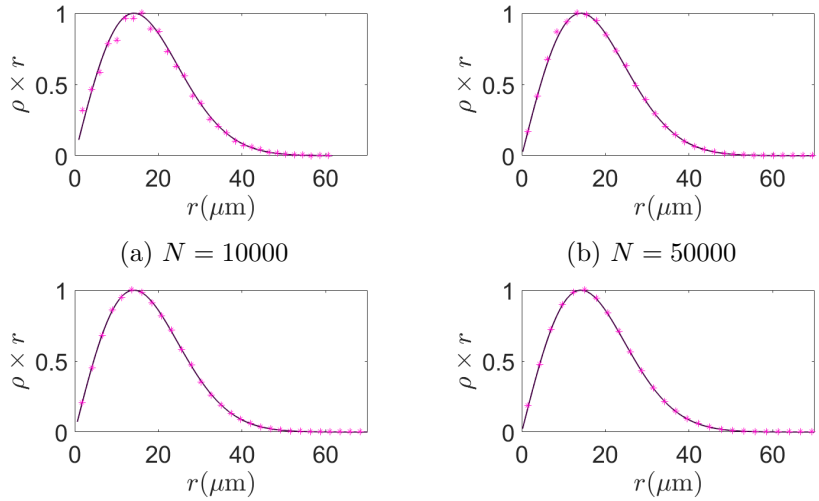
Figure 4.1: Analytical (plottet) and numerical (scattered) particle density for N particles experiencing free diffusion, with total diffusion time $T = 100$ms.

and 35 different simulation times, with maximum simulation time $T = 200$ ms. The normalized signal as function of b, for three different diffusion times, are shown in Fig. 4.4. Calculating the ADC values using Eq. 2.8 for $b \leq 800$ s/mm^2, and meaning over these gave the diffusion coefficient $D_0 = (1.656 \pm 0.005)$ μm^2/ms.

4.4 Spin Echo for Diffusion in Sphere

Pulsed Gradient

The calculation for the pulsed gradient includes 35 different simulation times with $T_{min} = 30$ ms and $T_{max} = 200$ ms, and 60 different gradient strengths with $G_{min} = 26434$ mT/m and $G_{max} = 606434$ mT/m. Normalized signal and the logarithm of the signal as function of b, for $T = 30$ ms, $T = 125$ ms and $T = 200$ ms are shown in Fig. 4.5. In Fig. 4.6 the apparent diffusion coefficient is plotted as a function of



(a) $N = 10000$ (b) $N = 50000$

(c) $N = 100000$ (d) $N = 400000$

Figure 4.2: Analytical (plotted) and numerical (scattered) position distribution for N particles experiencing free diffusion, total diffusion time $T = 100$ms.

the diffusion time $\Delta - \delta/3$. The results included are all the signal amplitudes associated with a b-value less than 800 s/mm^2, that is 1125 points. Calculating the sphere radii from this gives the radii given in Table 4.2. $r = 72.1$ μm and $r = 72.5$ μm including and not including D_0 respectively. Doing a rerun with the same parameters only changing the sphere radius to 85.0 μm gave $r = 95.6$ μm and $r = 96.0$ μm respectively.

Constant Gradient

For both the constant gradient and the constant gradient with a 5 ms break, two different simulations are done. One including simulation times 30 ms, 100 ms and 200 ms, varying the gradient strength from 1 mT/m to 71 mT/m with steps of 2 mT/m, and one with gradient strengths 1 mT/m, 30 mT/m and 71 mT/m, varying the simulation time from 30 ms to 200 ms with a step of 5 ms.

		Mean	Standard deviation	Kurtosis
Analytical		0	14.14	3.00
$N = 10K$	x	-0.23	14.15	2.98
	y	0.01	14.21	2.94
	z	-0.13	14.06	3.00
$N = 50K$	x	0.01	14.14	3.00
	y	-0.00	14.13	3.03
	z	0.00	14.06	2.99
$N = 100K$	x	0.06	14.19	3.00
	y	0.04	14.16	3.03
	z	0.00	14.09	3.02
$N = 400K$	x	0.01	14.17	3.01
	y	0.01	14.15	3.01
	z	0.01	14.11	3.00

Table 4.1: Mean value, standard deviation and kurtosis measured for simulation of free diffusion for N particles, using total diffusion time 100 ms.

	Not including D_0	Including D_0
R = 65 μm	55.0	55.3
R = 75 μm	72.1	72.5
R = 75 μm	73.0	73.4
R = 85 μm	95.6	96.0

Table 4.2: Radii in μm for the pulsed gradient simulations, calculated from the ADC regression lines. ADC for simulation 1, 75 μm, is shown in Fig. 4.6

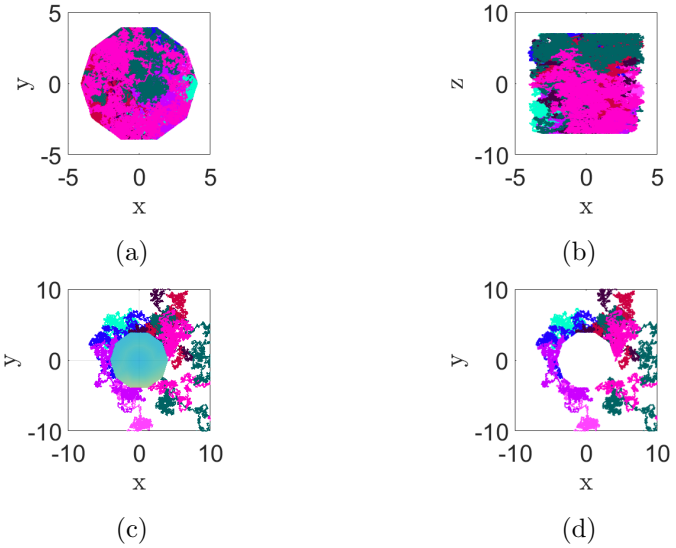


Figure 4.3: Trajectory of 10 particles diffusing inside and around a cylinder. In (a) and (b) the particles are placed with initial positions inside the cylinder, while in (c) and (d) they are positioned right outside. In (c) a transparent version of the cylinder i added to increase visibility.

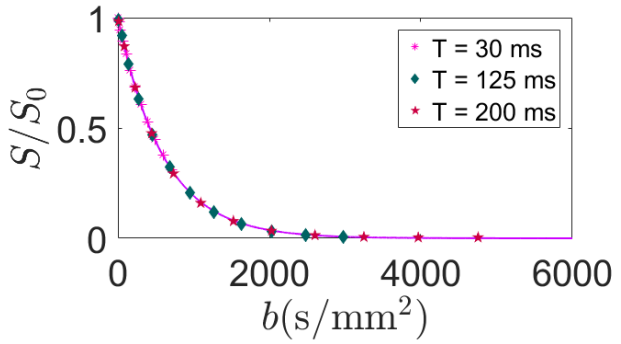


Figure 4.4: Normalized signal as function of b is scattered for three different simulation times, for the pulsed gradient case on free diffusion. The plotted line is the Gaussian given in 2.13, where D is the intrinsic diffusion coefficient used in the simulation.

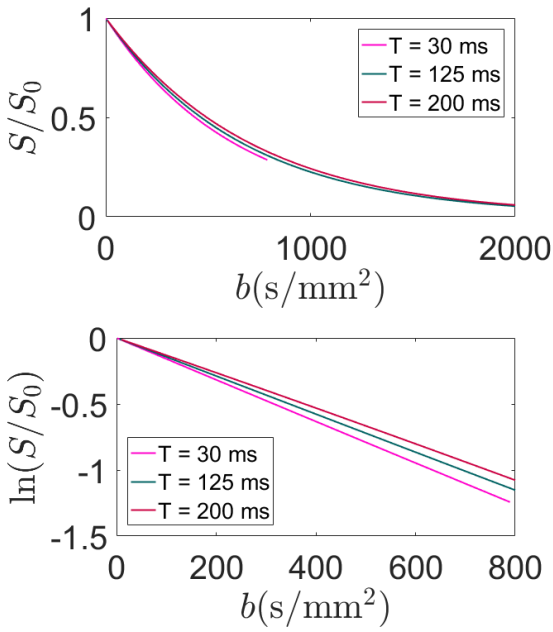


Figure 4.5: Normalized signal and logarithm of signal as function of b for the pulsed gradient. Three different diffusion times are included.

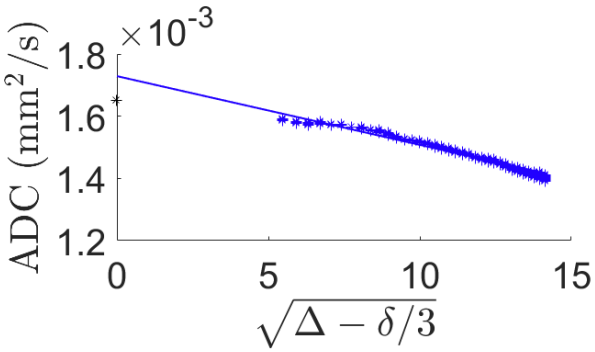


Figure 4.6: Measurement points and resulting regression line for the apparent diffusion coefficient as function of diffusion time for the pulsed gradient simulation. The intrinsic diffusion coefficient D_0 is scattered with a black star.

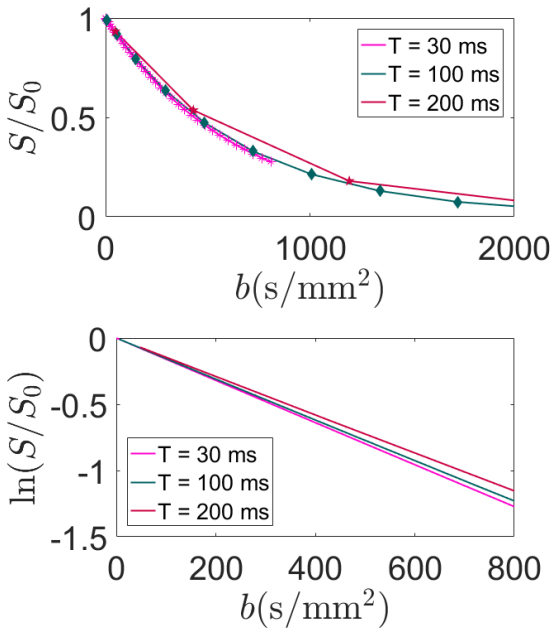


Figure 4.7: Normalized signal and logarithm of signal for constant gradient without break. Three different simulation times and gradient varying from 1 mT/m to 71 mT/m. Scattered points are included to clarify difference in distance between measurement points.

The signal and the logarithm of the signal for the three different diffusion times are plotted in Fig.4.7 and Fig.4.8 for the first and second case respectively. Using the results from the simulation with three different gradient strengths gives the apparent diffusion coefficient plot in Fig.4.9. The calculated radii for both cases, using Eq. 2.8 and the regression lines from the ADC-plots is given in table 4.3. For gradients without break

Varying δ and Δ

In Fig. 4.10 $\ln(S/S_0)$ is plotted for both the simulated and the experimental ([18]) data with the following values: $\delta = 7.8$ ms, 21.8 ms, 26.8 ms and $\Delta = 16.6$ ms, 27.6 ms, 32.6 ms. These values corresponds to

Diffusion time	Without break	With 5 ms break	
$\Delta - \delta/3$	65.9 , 68.0	63.0	Not including D_0
	71.9 , 74.1	70.8	Including D_0
Δ	80.7 , 83.3	79.4	Not including D_0
	88.0 , 90.8	87.5	Including D_0
$\Delta + \delta$	114.1 , 117.7	115.8	Not including D_0
	124.5 , 128.4	125.3	Including D_0

Table 4.3: Radii in μm for the constant gradient simulations, calculated from the ADC regression lines, for the 75 μm sphere. ADC for simulation 1, is shown in Fig. 4.9. The calculations includes 40 measurement points for the case without break, 41 points for the case with a 5 ms break. The two values given for the no break case are from the two separate simulations.

Diffusion time	Without break	With 5 ms break	
$\Delta - \delta/3$	82.6 , 74.1	78.4	Not including D_0
	89.9 , 81.2	87.9	Including D_0
Δ	101.2 , 90.8	98.6	Not including D_0
	110.1 , 99.5	108.6	Including D_0
$\Delta + \delta$	143.1 , 128.4	143.7	Not including D_0
	156.7 , 140.7	155.5	Including D_0

Table 4.4: Radii in μm for the constant gradient simulations, calculated from the ADC regression lines, for the 85 μm sphere. The calculations includes 40 measurement points for the case without break, 41 points for the case with a 5 ms break. The two values given for the no break case are from the two separate simulations.

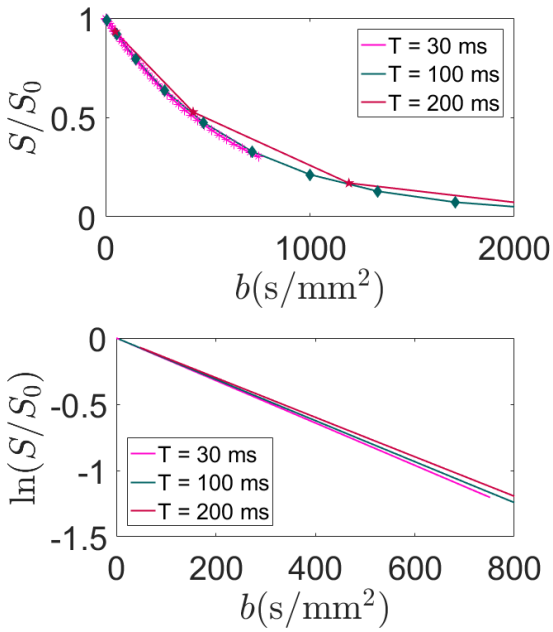
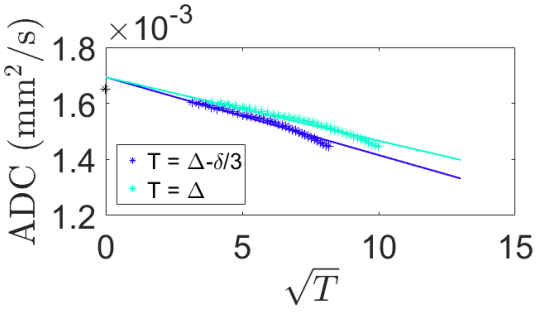


Figure 4.8: Signal and logarithm of signal for constant gradient with 5 ms break between the gradients. Three different simulation times and gradient varying from 1 mT/m to 71 mT/m. Scattered points are included to clarify difference in distance between measurement points.

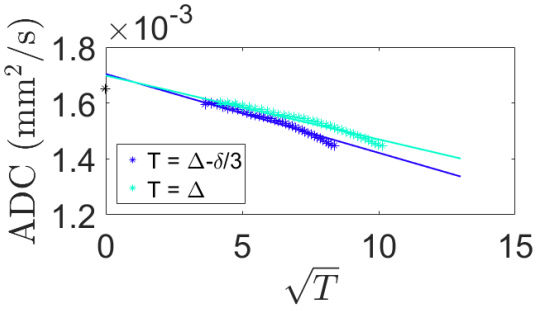
the experimental echo times: 38 ms, 60 ms and 70 ms.

4.5 FID Signal for Diffusion in Sphere

The resulting logarithmic plot for the FID simulation is shown in Fig. 4.11 as a function of the dimensionless parameter τ, for three different values of p.



(a) Constant gradient without break.



(b) Constant gradient with 5 ms break.

Figure 4.9: Measurement points and resulting regression line for the apparent diffusion coefficient as function of two different diffusion times, D_0 is included in the regression lines and D_0 is scattered with a black star.

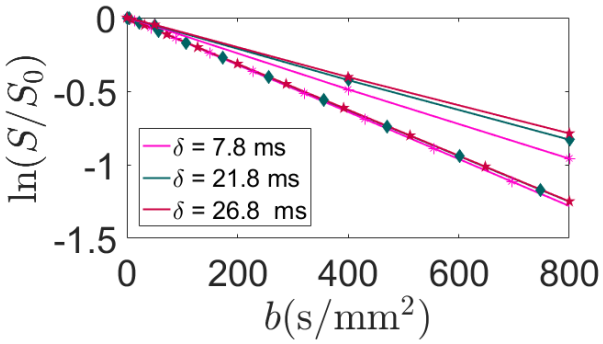


Figure 4.10: Logarithmic plot of simulated and experimental signal. The lines with the same color and scattering symbols corresponds to the same δ and Δ, and the densest lines corresponds to the simulated results.

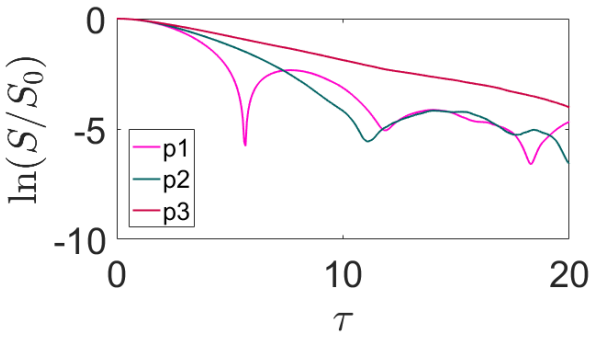


Figure 4.11: FID signal plotted as function of the dimensionless parameter τ for hign b-values. Three different values for p (see Eq. 2.14) are included; $p_1 = 0.05$, $p_2 = 0.15$ and $p_3 = 0.25$.

Chapter 5

Discussion and Conclusions

5.1 Performance and Execution Time

For the particle diffusion in the sphere, it is the multiplication of the gradient with the position vector and the summation of this that is the most time-consuming part of the code. This means that for short gradient durations like the pulsed gradient, the execution time is much shorter than for a constant gradient with high duration. The case of constant gradient with and without break, done in the same simulation had an execution time of 10 hours, including three different gradients and 35 different diffusion times, where the maximum diffusion time was 200 ms. The pulsed gradient case had an execution time 3.7 hours, and here 59 different gradients and 35 different diffusion times were included.

For the cylinder case, the wall detection is a much more complex process and unlike the sphere case, the collision check is done for each step. It should be possible to include a requirement to avoid this, but as the program is meant to work for a variation of obstacles this may be a somewhat cumbersome affair. Also the walls here are defined by the grid points, meaning normals from the planes to the particle have to be found. Though, very similar code do already exits in the program and may be reused for this. Concerning the execution time, this is way higher than for the sphere, and the pulsed gradient simulation done for 10 000 particles with 5 different gradient strengths, 31 different diffusion

times and maximum diffusion time 50 ms needed 74 hours. Though, investigating the utilization of the four workers during this execution, it seems like this code for the moment have too much communication with none parallelized parts, so improvements of this should be done.

As mentioned, the intentions were to make this work for a large variation of shapes. It should be noted that exact criteria for the obstacles are not set, but the most obvious one is that all grid faces should have sides large compared to the particle step size. Sharp edges leading different sides of the obstacle to be spaced close together compared to face size may also cause problems, this is due to the fact that the particle only is tested for hitting one face with neighbors, and because this face is chosen by the distance to its grid points and not to the actual plane. From this, it might be a suggestion to rewrite the code to generate all the normals from each face from the start, instead of using the distances to the grid points. This would demand a bit more calculations in the first part, but would probably simplify the process of avoiding collision check for each step, and also allow for an even bigger variation of obstacles. Anyhow, in introducing a new obstacle to the program it would be sensible to run a test, plotting the particle trajectory to ensure no holes are present, also one may want to always save the final position for each particle to check this.

Using the Linux cluster for the simulation has been necessary due to the possibility of screening the sessions, that said, only four worker are used in the parallelizations and the cluster seems a bit slow, so it should be possible to reduce the execution time quite a bit. In comparison executing the same code both on the cluster and with an Intel CORE i5-6200U, 2.4GHz CPU on a relatively new, but cheap, computer resulted in an execution time being 815 sec on the cluster and 159 sec on the PC, both without parallelization.

In conclusion, it should be possible to use this code for setting up a more complex environment for a diffusion MRI simulation. The simples case would be to put two obstacles inside or next to each other, and in essence this would mean looping over each of the obstacles for the collision check, though this has yet not been tested. For something more complex than this, the need for including a check to avoid wall detection in every diffusion step is crucial. Also, it may be a good idea

to include subvolumes as done in [17], here one avoids including all obstacles in the wall detection by restrict the check to the subvolume the spin in question is located in.

Finally, doing the implementations the main focus has been on avoiding errors, little time has been used to improve the execution time, so the possibility for improvements are absolutely present. Also, managing to re-implement this code using fortran would probably be a good way to decrease the execution time, but this would require quite a big amount of work.

5.2 Random Walk and Obstacle Interaction

Considering the free random walk, the results given in Fig. 4.2 and Fig. 4.1 indicates that the diffusion simulation in the implementation works as it should. The plots are a decent tool to use when deciding how many particles to include in a simulation, but remembering that these plots are generated for a random walk with total diffusion time 100 ms are important. E.g for a 50 ms simulation, more than 50 000 particle should be included.

The wall detection and interaction also seem to be correct. The implementation for the grid based obstacles are done only considering the grid points, and not the shape, so choosing the cylinder as the test obstacle was arbitrary. This shows that the interaction works for at least angles of 90° and up. Saving the final position of each particle was done for the long test run mentioned in the previous section, and the maximum position og these was still inside the cylinder (centered at the origin).

5.3 Signal and ADC

For the radii, it may at first glance look like at least some of the results are quite decent. But it is important to be aware that Eq. 2.8 is only valid for short time diffusion. In the derivation of this equation [6], one uses that the fraction of particles that have seen the wall is given the total surface area timed with the diffusion length. This means the maximum diffusion length should be much smaller than the sphere

radii, and in the simulations done for this thesis, this fact seems to have been somewhat neglected somewhere in between keeping the b-values at a decent level and making sure the particles have no holes to escape through. A rerun using shorter diffusion times was tried in the last minute, but for these times some oscillations occur in the ADC-plots. This probably means more particles should be included for simulations considering short diffusion times.

Considering the ADC plots, all of them seem to have a tendency change somewhere in the first half of the time range. This implies that the ADC's time dependence is dependent on the spin's diffusion length versus the size of the obstacle, as it should. Also, this implies that computing D_0 from the ADC results will give a more accurate result than the regression does.

It should be noted that there may be a systematic error either in the signal generation, or in the ADC calculations. This suspicion comes from the fact that the calculated radii are as close to the actual radii as they are, and that the expected ADC line for the correct time regime has to have a smaller slope than this, resulting in a higher radius.

5.4 Other

Considering the results for high b-values, this seem to be quite correct comparing with results given in [10]. Though, this run was mostly done as an extra check for the signal generation, and further investigation of the high b-value behavior is not done.

Appendix A

Code

Some excerpts and explanation of the code is included, some of the code are unfortunately in lack of a decent amount of comments at the moment.

The method for the random walk implementation is shown, this is for one particle and K time steps.

```
%****************************************
d_cos_theta = 2*rand(1, K) - 1;
d_theta = acos(d_cos_theta);
d_phi   = 2*pi*rand(1, K);

x = cumsum([0 dr*sin(d_theta).*cos(d_phi)]);
y = cumsum([0 dr*sin(d_theta).*sin(d_phi)]);
z = cumsum([0 dr*d_cos_theta]);
%****************************************
```

The function Get4ClosestDist() uses the particle's position and the grid matrices and finds the three or four pairs of indices belonging to the face most likely to be nearest the particle, and return these in counter clockwise direction seen from outside the obstacle.

```
%*********************************************************************
%Returns the indexes belonging to square/triangle in Obstacle [X,Y,Z]
%with grid points nearest (x,y,z).
%Gives out the points counterclockwise from outside obstacle
%clockwise from inside obstacle
%*********************************************************************
function MinDist = Get4ClosestDist(x, y, z, X, Y, Z, ObsSize)

Shape = 0;      %Shape = 0 => square
                %Shape = 1 => triangle
                %Shape = 2 or 3 => first: square or triangle
                %                  then : triangle, top or bottom
```

```
13
14  %********** Initilizing **********
15  InitialIndex1 = round(ObsSize/2);
16  InitialIndex2 = InitialIndex1 + 1;
17
18  index1_1 = InitialIndex1;
19  index1_2 = InitialIndex1;
20  index2_1 = InitialIndex2;
21  index2_2 = InitialIndex2;
22
23  Dist1 = (X(InitialIndex1, InitialIndex1) - x)^2 + ...
24          (Y(InitialIndex1, InitialIndex1) - y)^2 + ...
25          (Z(InitialIndex1, InitialIndex1) - z)^2 + 1;
26  %******************************
27  %******* Finds the closest gridpoint *******
28  for j = 1 : ObsSize
29      for k = 1 : ObsSize - 1
30          Dist = (X(j,k) - x)^2 + ...
31                 (Y(j,k) - y)^2 + ...
32                 (Z(j,k) - z)^2 ;
33          if Dist < Dist1
34              index1_1 = j;
35              index1_2 = k;
36              Dist1 = Dist;
37          end
38      end
39  end
40  %*****************************************
41  %****** Find the closest of the neighboring points ******
42  I1 = index1_1 - 1;
43  I2 = index1_1 + 1;
44  I3 = index1_2 - 1;
45  I4 = index1_2 + 1;
46
47  if index1_2 == 1
48      I3 = ObsSize - 1;
49  end
50
51  if index1_1 == 1
52      Shape = 1;
53      index2_1 = 2;
54      I1 = index2_1;
55  end
56
57  if index1_1 == ObsSize
58      Shape = 1;
59      index2_1 = ObsSize - 1;
60      I1 = index2_1 ;
61  end
62
63  if index1_1 == 2
64      Shape = 2;
65  end
66  if index1_1 == ObsSize - 1
67      Shape = 3;
68  end
69
70  if Shape == 0
71      DistNeighbor1 = (X(I1,index1_2) - x).^2 + (Y(I1,index1_2) - y).^2 + ...
72                      (Z(I1,index1_2) - z).^2;
73      DistNeighbor2 = (X(I2,index1_2) - x).^2 + (Y(I2,index1_2) - y).^2 + ...
```

```
74                      (Z(I2,index1_2) - z).^2;
75      DistNeighbor3 = (X(index1_1,I3) - x).^2 + (Y(index1_1,I3) - y).^2 + ...
76                      (Z(index1_1,I3) - z).^2;
77      DistNeighbor4 = (X(index1_1,I4) - x).^2 + (Y(index1_1,I4) - y).^2 + ...
78                      (Z(index1_1,I4) - z).^2;
79      [M,I] = min([DistNeighbor1 DistNeighbor2 DistNeighbor3 DistNeighbor4]);
80      if I == 1
81          index2_1 = I1;
82          index2_2 = index1_2;
83      elseif I == 2
84          index2_1 = I2;
85          index2_2 = index1_2;
86      elseif I == 3
87          index2_1 = index1_1;
88          index2_2 = I3;
89      else
90          index2_1 = index1_1;
91          index2_2 = I4;
92      end
93  elseif Shape == 1
94      Dist = zeros(1,ObsSize-1);
95      for k = 1:ObsSize-1;
96          Dist(k) = (X(I1,k) - x).^2 + (Y(I1,k) - y).^2 + (Z(I1,k) - z).^2;
97      end
98      [M,I] = min(Dist);
99      index2_2 = I;
100 elseif Shape == 2
101     DistNeighbor1 = (X(I1,index1_2) - x).^2 + (Y(I1,index1_2) - y).^2 + ...
102                     (Z(I1,index1_2) - z).^2;
103     DistNeighbor2 = (X(I2,index1_2) - x).^2 + (Y(I2,index1_2) - y).^2 + ...
104                     (Z(I2,index1_2) - z).^2;
105     DistNeighbor3 = (X(index1_1,I3) - x).^2 + (Y(index1_1,I3) - y).^2 + ...
106                     (Z(index1_1,I3) - z).^2;
107     DistNeighbor4 = (X(index1_1,I4) - x).^2 + (Y(index1_1,I4) - y).^2 + ...
108                     (Z(index1_1,I4) - z).^2;
109     [M,I] = min([DistNeighbor1 DistNeighbor2 DistNeighbor3 DistNeighbor4]);
110     if I == 1
111         index2_1 = I1;
112         index2_2 = index1_2;
113         Shape = 1;
114     elseif I == 2
115         index2_1 = I2;
116         index2_2 = index1_2;
117         Shape = 0;
118     elseif I == 3
119         index2_1 = index1_1;
120         index2_2 = I3;
121         Shape = 2;
122     else
123         index2_1 = index1_1;
124         index2_2 = I4;
125         Shape = 2;
126     end
127 elseif Shape == 3
128     DistNeighbor1 = (X(I1,index1_2) - x).^2 + (Y(I1,index1_2) - y).^2 + ...
129                     (Z(I1,index1_2) - z).^2;
130     DistNeighbor2 = (X(I2,index1_2) - x).^2 + (Y(I2,index1_2) - y).^2 + ...
131                     (Z(I2,index1_2) - z).^2;
132     DistNeighbor3 = (X(index1_1,I3) - x).^2 + (Y(index1_1,I3) - y).^2 + ...
133                     (Z(index1_1,I3) - z).^2;
134     DistNeighbor4 = (X(index1_1,I4) - x).^2 + (Y(index1_1,I4) - y).^2 + ...
```

```
135                     (Z(index1_1,I4) - z).^2;
136     [M,I] = min([DistNeighbor1 DistNeighbor2 DistNeighbor3 DistNeighbor4]);
137     if I == 1
138         index2_1 = I1;
139         index2_2 = index1_2;
140         Shape = 0;
141     elseif I == 2
142         index2_1 = I2;
143         index2_2 = index1_2;
144         Shape = 1;
145     elseif I == 3
146         index2_1 = index1_1;
147         index2_2 = I3;
148         Shape = 2;
149     else
150         index2_1 = index1_1;
151         index2_2 = I4;
152         Shape = 2;
153     end
154 end
155 %*********************************************************
156 %***************Finding the two (one) other points*****************
157 if Shape == 0                       %square
158     if index1_1 == index2_1
159         I1 = index1_1 - 1 ;
160         I2 = index1_1 + 1 ;
161         Dist1 = (X(I1,index1_2)-x)^2 ...
162         + (Y(I1,index1_2)-y)^2 ...
163         + (Z(I1,index1_2)-z)^2;
164         Dist2 = (X(I2,index1_2)-x)^2 ...
165         + (Y(I2,index1_2)-y)^2 ...
166         + (Z(I2,index1_2)-z)^2;
167         Dist3 = (X(I1,index2_2)-x)^2 ...
168         + (Y(I1,index2_2)-y)^2 ...
169         + (Z(I1,index2_2)-z)^2;
170         Dist4 = (X(I2,index2_2)-x)^2 ...
171         + (Y(I2,index2_2)-y)^2 ...
172         + (Z(I2,index2_2)-z)^2;
173         [M,I] = min([Dist1 Dist2 Dist3 Dist4]);
174         if I == 1 || I == 3
175             index3_1 = I1;
176             index3_2 = index1_2;
177             index4_1 = I1;
178             index4_2 = index2_2;
179         else                                %dvs I == 2 || I == 4
180             index3_1 = I2;
181             index3_2 = index1_2;
182             index4_1 = I2;
183             index4_2 = index2_2;
184         end
185     elseif index1_2 == index2_2
186         I1 = index1_2 - 1 ;
187         I2 = index1_2 + 1 ;
188         if index1_2 == 1
189             I1  = ObsSize -1 ;
190         elseif index1_2 == ObsSize
191             I2  = 2 ;
192         end
193         Dist1 = (X(index1_1,I1)-x)^2 ...
194         + (Y(index1_1,I1)-y)^2 ...
195         + (Z(index1_1,I1)-z)^2;
```

```
196         Dist2 = (X(index1_1,I2)-x)^2 ...
197         + (Y(index1_1,I2)-y)^2 ...
198         + (Z(index1_1,I2)-z)^2;
199         Dist3 = (X(index2_1,I1)-x)^2 ...
200         + (Y(index2_1,I1)-y)^2 ...
201         + (Z(index2_1,I1)-z)^2;
202         Dist4 = (X(index2_1,I2)-x)^2 ...
203         + (Y(index2_1,I2)-y)^2 ...
204         + (Z(index2_1,I2)-z)^2;
205         [M,I] = min([Dist1 Dist2 Dist3 Dist4]);
206         if I == 1 || I == 3
207             index3_1 = index1_1;
208             index3_2 = I1;
209             index4_1 = index2_1;
210             index4_2 = I1;  %test
211         else                 %dvs I == 2 || I == 4
212             index3_1 = index1_1;
213             index3_2 = I2;  %test
214             index4_1 = index2_1;
215             index4_2 = I2;
216         end
217     end
218     MinDist = [index1_1, index1_2, index2_1, index2_2, ...
219     index3_1, index3_2, index4_1, index4_2 ];
220 end
221 if Shape == 1                    %triangle
222     if index1_1 == 1
223         index3_1 = 2;
224         I1 = index2_2-1;
225         I2 = index2_2+1;
226         if index2_2 == 1
227             I1 = ObsSize - 1;
228         end
229         if index2_2 == ObsSize
230             I2 = 2;
231         end
232         dist = [(X(2,I1)-x).^2 + (Y(2,I1)-y).^2 + (Z(2,I1)-z).^2 ...
233                 (X(2,I2)-x).^2 + (Y(2,I2)-y).^2 + (Z(2,I2)-z).^2 ];
234         [M,I] = min(dist);
235         if I == 1
236             index3_2 = I1;
237         else
238             index3_2 = I2;
239         end
240     elseif index2_1 == 1
241         index3_1 = 2;
242         I1 = index1_2-1;
243         I2 = index1_2+1;
244         if index1_2 == 1
245             I1 = ObsSize - 1;
246         end
247         if index1_2 == ObsSize
248             I2 = 2;
249         end
250         dist = [(X(2,I1)-x).^2 + (Y(2,I1)-y).^2 + (Z(2,I1)-z).^2 ...
251                 (X(2,I2)-x).^2 + (Y(2,I2)-y).^2 + (Z(2,I2)-z).^2 ];
252         [M,I] = min(dist);
253         if I == 1
254             index3_2 = I1;
255             else
256             index3_2 = I2;
```

```
257             end
258         elseif index1_1 == ObsSize
259             index3_1 = ObsSize-1;
260             I1 = index2_2-1;
261             I2 = index2_2+1;
262             if index2_2 == 1
263                 I1 = ObsSize - 1;
264             end
265             if index2_2 == ObsSize
266                 I2 = 2;
267             end
268             dist = [(X(index3_1,I1)-x).^2 + (Y(index3_1,I1)-y).^2    ...
269                   + (Z(index3_1,I1)-z).^2 ...
270                     (X(index3_1,I2)-x).^2 + (Y(index3_1,I2)-y).^2 + ...
271                     (Z(index3_1,I2)-z).^2 ];
272             [M,I] = min(dist);
273             if I == 1
274                 index3_2 = I1;
275             else
276                 index3_2 = I2;
277             end
278         elseif index2_1 == ObsSize
279             index3_1 = ObsSize-1;
280             I1 = index1_2-1;
281             I2 = index1_2+1;
282             if index1_2 == 1
283                 I1 = ObsSize - 1;
284             end
285             if index1_2 == ObsSize
286                 I2 = 2;
287             end
288             dist = [(X(index3_1,I1)-x).^2 + (Y(index3_1,I1)-y).^2 + ...
289                     (Z(index3_1,I1)-z).^2 ...
290                     (X(index3_1,I2)-x).^2 + (Y(index3_1,I2)-y).^2 + ...
291                     (Z(index3_1,I2)-z).^2 ];
292             [M,I] = min(dist);
293             if I == 1
294                 index3_2 = I1;
295             else
296                 index3_2 = I2;
297             end
298         end
299         MinDist = [index1_1, index1_2, index2_1, index2_2, ...
300         index3_1, index3_2];
301     end
302     if Shape == 2          %square or triangle
303         if index1_1 == 2
304             dist = [(X(1,1)-x).^2 (Y(1,1)-y).^2 (Z(1,1)-z).^2 ...
305                     (X(3,index1_2)-x).^2 (Y(3,index1_2)-y).^2 (Z(3,index1_2)-z).^2 ...
306                     (X(3,index2_2)-x).^2 (Y(3,index2_2)-y).^2 (Z(3,index2_2)-z).^2 ];
307             [M,I] = min(dist);
308             if I == 1
309                 index3_1 = 1;
310                 index3_2 = 1;
311                 Shape = 1;
312             else
313                 index3_1 = 3;
314                 index3_2 = index1_2;
315                 index4_1 = 3;
316                 index4_2 = index2_2;
317                 Shape = 0;
```

```
318             end
319         else        %dvs index1_1 = ObsSize-1
320             a = ObsSize;
321             b = ObsSize - 2;
322             dist = [(X(a,1)-x).^2 (Y(a,1)-y).^2 (Z(a,1)-z).^2 ...
323                     (X(b,index1_2)-x).^2 (Y(b,index1_2)-y).^2 (Z(b,index1_2)-z).^2 ...
324                     (X(b,index2_2)-x).^2 (Y(b,index2_2)-y).^2 (Z(b,index2_2)-z).^2 ];
325             [M,I] = min(dist);
326             if I == 1
327                 index3_1 = a;
328                 index3_2 = 1;
329                 Shape = 1;
330             else
331                 index3_1 = b;
332                 index3_2 = index1_2;
333                 index4_1 = b;
334                 index4_2 = index2_2;
335                 Shape = 0;
336             end
337         end
338         if Shape == 1
339             MinDist = [index1_1, index1_2, index2_1, index2_2, ...
340             index3_1, index3_2];
341             if index1_1 == 1 || index1_1 == 2
342                 Shape = 3;
343                 else
344                 Shape = 2;
345             end
346         else
347             MinDist = [index1_1, index1_2, index2_1, index2_2, ...
348             index3_1, index3_2, index4_1, index4_2];
349         end
350     end
351     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
352     %***** Sorts the points to be counterclockwise*****
353     if Shape ~= 0
354         if index1_1 == 1 || index1_1 == 2
355             Shape = 3;
356         elseif index1_1 == ObsSize || index1_1 == ObsSize - 1
357             Shape = 2;
358         end
359     end
360     if Shape == 0       %square
361         I1 = index1_1 - 1;
362         I2 = index1_1 + 1;
363         I3 = index1_2 - 1;
364         I4 = index1_2 + 1;
365         if I3 == 0
366             I3 = ObsSize-1;
367         end
368         if I4 == ObsSize + 1
369             I4 = 2;
370         end
371         for k = 1:3
372             if MinDist(2*k+1) == I1 && MinDist(2*k+2) == I3
373                 MinDist(3:8) = [index1_1, I3, I1, I3, I1, index1_2];
374             elseif MinDist(2*k+1) == I1 && MinDist(2*k+2) == I4
375                 MinDist(3:8) = [I1, index1_2, I1, I4, index1_1, I4];
376             elseif MinDist(2*k+1) == I2 && MinDist(2*k+2) == I3
377                 MinDist(3:8) = [I2, index1_2, I2, I3, index1_1, I3];
378             elseif MinDist(2*k+1) == I2 && MinDist(2*k+2) == I4
```

```
379             MinDist(3:8) = [index1_1, I4, I2, I4, I2, index1_2];
380         end
381     end
382     elseif Shape == 2                   %triangle, top
383         if index1_1 == ObsSize
384             if index2_2 == 2 && index3_2 == ObsSize
385                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
386             elseif index2_2 == 1 && index3_2 == ObsSize-1
387                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
388             elseif index2_2 < index3_2
389                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
390             elseif index2_2 == ObsSize-1 && index3_2 == 1
391                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
392             elseif index2_2 == ObsSize && index3_2 == 2
393                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
394             else
395                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
396             end
397         elseif index2_1 == ObsSize
398             if index3_2 == ObsSize-1 && index1_2 == 1
399                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
400             elseif index3_2 == ObsSize && index1_2 == 2
401                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
402             elseif index1_2 < index3_2
403                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
404             elseif index1_2 == ObsSize-1 && index3_2 == 1
405                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
406             elseif index1_2 == ObsSize && index3_2 == 2
407                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
408             else
409                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
410             end
411         else                        %index3_1 == ObsSize
412             if index2_2 == ObsSize-1 && index1_2 == 1
413                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
414             elseif index2_2 == ObsSize && index1_2 == 2
415                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
416             elseif index1_2 < index2_2
417                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
418             elseif index1_2 == ObsSize-1 && index2_2 == 1
419                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
420             elseif index1_2 == ObsSize && index2_2 == 2
421                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
422             else
423             MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
424         end
425     end
426     elseif Shape == 3                                   %triangle, bottom
427         if index1_1 == 1
428             if index3_2 == ObsSize-1 && index2_2 == 1
429                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
430             elseif index3_2 == ObsSize && index2_2 == 2
431                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
432             elseif index2_2 < index3_2
433                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
434             elseif index2_2 == ObsSize-1 && index3_2 == 1
435                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
436             elseif index2_2 == ObsSize && index3_2 == 2
437                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
438             else
439                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
```

```
440             end
441         elseif index2_1 == 1
442             if index1_2 == 1 && index3_2 == ObsSize-1
443                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
444             elseif index1_2 == 2 && index3_2 == ObsSize
445                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
446             elseif index3_2 < index1_2
447                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
448             elseif index3_2 == 1 && index1_2 == ObsSize-1
449                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
450             elseif index3_2 == 2 && index1_2 == ObsSize
451                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
452             else
453                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
454             end
455         else %index3_1 == 1
456             if index2_2 == ObsSize - 1 && index1_2 == 1
457                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
458             elseif index2_2 == ObsSize && index1_2 == 2
459                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
460             elseif index1_2 < index2_2
461                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
462             elseif index1_2 == ObsSize - 1 && index2_2 == 1
463                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
464             elseif index1_2 == ObsSize && index2_2 == 2
465                 MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
466             else
467                 MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
468             end
469         end
470     end
471 %*************************************************
472 end
```

GetNeighbours() uses the indices belonging to the first face and the size of the matrices defining the obstacle to find the neighboring faces and return these, with the indices given in counter clockwise direction seen from outside the obstacle.

```
1  function Neighbours = GetNeighbours(index1_1, index1_2, ...
2                                       index2_1, index2_2, ...
3                                       index3_1, index3_2, ...
4                                       ObsSize, varargin)
5
6  N = size(varargin);
7  N = N(2);
8
9  if N == 2
10     index4_1 = varargin{1};
11     index4_2 = varargin{2};
12     Shape = 4;
13 else
14     Shape = 3;
15 end
16
17 if Shape == 4
18     CN = zeros(8,8);
19     %sorts the grid points
20     if index4_1 > index1_1
21         Index1_1 = index4_1;
```

```
22          Index1_2 = index4_2;
23          Index2_1 = index1_1;
24          Index2_2 = index1_2;
25          Index3_1 = index2_1;
26          Index3_2 = index2_2;
27          Index4_1 = index3_1;
28          Index4_2 = index3_2;
29      elseif index3_1 > index4_1
30          Index1_1 = index3_1;
31          Index1_2 = index3_2;
32          Index2_1 = index4_1;
33          Index2_2 = index4_2;
34          Index3_1 = index1_1;
35          Index3_2 = index1_2;
36          Index4_1 = index2_1;
37          Index4_2 = index2_2;
38      elseif index2_1 > index3_1
39          Index1_1 = index2_1;
40          Index1_2 = index2_2;
41          Index2_1 = index3_1;
42          Index2_2 = index3_2;
43          Index3_1 = index4_1;
44          Index3_2 = index4_2;
45          Index4_1 = index1_1;
46          Index4_2 = index1_2;
47      else
48          Index1_1 = index1_1;
49          Index1_2 = index1_2;
50          Index2_1 = index2_1;
51          Index2_2 = index2_2;
52          Index3_1 = index3_1;
53          Index3_2 = index3_2;
54          Index4_1 = index4_1;
55          Index4_2 = index4_2;
56      end
57
58      I1 = Index1_2 - 1;
59      if I1 == 0
60          I1 = ObsSize-1;
61      end
62      CN(1,:) = [Index1_1 Index1_2 Index1_1 I1 Index2_1 I1 Index2_1 Index2_2];
63      I2 = Index2_1 - 1;
64      I3 = Index3_2 + 1;
65      if I3 == ObsSize + 1
66          I3 = 2;
67      end
68      if I2 == 1
69          CN(2,:) = [Index2_1 Index2_2 Index2_1 I1 1 1 0 0];
70          CN(3,:) = [Index2_1 Index2_2 1 1 Index3_1 Index3_2 0 0];
71          CN(4,:) = [Index3_1 Index3_2 1 1 Index3_1 I3 0 0];
72      else
73          CN(2,:) = [Index2_1 Index2_2 Index2_1 I1 I2 I1 I2 Index2_2];
74          CN(3,:) = [Index2_1 Index2_2 I2 Index2_2 I2 Index3_2 Index3_1 Index3_2];
75          CN(4,:) = [Index3_1 Index3_2 I2 Index3_2 I2 I3 Index3_1 I3];
76      end
77      CN(5,:) = [Index4_1 Index4_2 Index3_1 Index3_2 Index3_1 I3 Index4_1 I3];
78      I4 = Index4_1 + 1;
79      if I4 == ObsSize
80          CN(6,:) = [Index4_1 Index4_2 Index4_1 I3 ObsSize 1 0 0];
81          CN(7,:) = [Index4_1 Index4_2 ObsSize 1 Index1_1 Index1_2 0 0];
82          CN(8,:) = [Index1_1 Index1_2 ObsSize 1 Index1_1 I1 0 0];
```

```
83      else
84          CN(6,:) = [Index4_1 Index4_2 Index4_1 I3 I4 I3 I4 Index4_2];
85          CN(7,:) = [Index4_1 Index4_2 I4 Index4_2 I4 Index1_2 Index1_1 Index1_2];
86          CN(8,:) = [Index1_1 Index1_2 I4 Index1_2 I4 I1 Index1_1 I1];
87      end
88  else     %triangle
89      R = 0;
90      if index1_1 == 1
91          R = 1;
92          Index1_1 = index1_1;
93          Index1_2 = index1_2;
94          Index2_1 = index2_1;
95          Index2_2 = index2_2;
96          Index3_1 = index3_1;
97          Index3_2 = index3_2;
98      elseif index2_1 == 1
99          R = 1;
100         Index1_1 = index2_1;
101         Index1_2 = index2_2;
102         Index2_1 = index3_1;
103         Index2_2 = index3_2;
104         Index3_1 = index1_1;
105         Index3_2 = index1_2;
106     elseif index3_1 == 1
107         R = 1;
108         Index1_1 = index3_1;
109         Index1_2 = index3_2;
110         Index2_1 = index1_1;
111         Index2_2 = index1_2;
112         Index3_1 = index2_1;
113         Index3_2 = index2_2;
114     elseif index2_1 == ObsSize
115         Index1_1 = index2_1;
116         Index1_2 = index2_2;
117         Index2_1 = index3_1;
118         Index2_2 = index3_2;
119         Index3_1 = index1_1;
120         Index3_2 = index1_2;
121     elseif index3_1 == ObsSize
122         Index1_1 = index3_1;
123         Index1_2 = index3_2;
124         Index2_1 = index1_1;
125         Index2_2 = index1_2;
126         Index3_1 = index2_1;
127         Index3_2 = index2_2;
128     else
129         Index1_1 = index1_1;
130         Index1_2 = index1_2;
131         Index2_1 = index2_1;
132         Index2_2 = index2_2;
133         Index3_1 = index3_1;
134         Index3_2 = index3_2;
135     end
136     if R == 1    %bottom
137         for k = 1:ObsSize-1
138             CN(k,:) = [1 1 2 k+1 2 k 0 0];
139         end
140         I1 = Index2_2 + 1;
141         I2 = Index3_2 - 1;
142         if I1 == ObsSize + 1
143             I1 = 2;
```

```
144             end
145             if I2 == 0;
146                 I2 = ObsSize - 1;
147             end
148             CN(ObsSize,:) = [2 Index2_2 2 I1 3 I1 3 Index2_2];
149             CN(ObsSize+1,:) = [2 Index2_2 3 Index2_2 3 Index3_2 2 Index3_2];
150             CN(ObsSize+2,:) = [2 Index3_2 3 Index3_2 3 I2 2 I2];
151         else
152             for k = 1:ObsSize-1
153                 CN(k,:) = [ObsSize 1 ObsSize-1 k ObsSize-1 k+1 0 0];
154             end
155             I1 = Index2_2 - 1;
156             I2 = Index3_2 + 1;
157             if I1 == 0
158                 I1 = ObsSize - 1;
159             end
160             if I2 == ObsSize + 1;
161                 I2 = 2;
162             end
163             P1 = ObsSize-1;
164             P2 = ObsSize-2;
165             CN(ObsSize,:) = [P1 Index2_2 P1 I1 P2 I1 P2 Index2_2];
166             CN(ObsSize+1,:) = [P1 Index2_2 P2 Index2_2 P2 Index3_2 P1 Index3_2];
167             CN(ObsSize+2,:) = [P1 Index3_2 P2 Index3_2 P2 I2 P1 I2];
168         end
169     end
170
171     Neighbours = CN;
172
173     end
```

InteractObstacle() uses the particles position and velocity to check if the particle do collide and if it does, the wall interaction is generated. This function is used for the grid based obstacles.

```
1   function NewPos = InteractObstacle(x, y, z, v, dt, GP1, GP2, GP3, varargin)
2
3   if length(varargin) == 1
4       GP4 = varargin{1};
5       Shape = 4;
6   else
7       Shape = 3;
8   end
9
10  vec1 = GP2 - GP1;
11  vec2 = GP3 - GP2;
12
13  normal = cross(vec1,vec2);
14  normal = normal/norm(normal);
15
16  t = dot((GP1 - [x y z]),normal)/dot(v,normal);
17
18  if t <= dt && t > 0                                          %may collide
19      pos_int = [x y z] + t*v;                                 %intersection position
20      if Shape == 4
21          if dot(cross(GP2-GP1,pos_int-GP1),normal) >= 0 && ...   %do collide
22             dot(cross(GP3-GP2,pos_int-GP2),normal) >= 0 && ...   %
23             dot(cross(GP4-GP3,pos_int-GP3),normal) >= 0 && ...   %
24             dot(cross(GP1-GP4,pos_int-GP4),normal) >= 0          %
25              if dot(v,normal) > 0
```

```
                normal = - normal;
            end
            v_new = v - 2*dot(v,normal)*normal;
            v_new = v_new*norm(v)/norm(v_new);
            NewPos = [pos_int + (dt-t)*v_new ...
            pos_int                  ...
            v_new dt-t];
        else
        NewPos = 0;
        end
    else  %triangle
        if dot(cross(GP2-GP1,pos_int-GP1),normal) >= 0 && ...  %do collide
           dot(cross(GP3-GP2,pos_int-GP2),normal) >= 0 && ...  %
           dot(cross(GP1-GP3,pos_int-GP3),normal) >= 0         %
            v_new = v - 2*dot(v,normal)*normal;
            v_new = v_new*norm(v)/norm(v_new);
            NewPos = [pos_int + (dt-t)*v_new ...
                      pos_int                ...
                      v_new dt-t];
        else
            NewPos = 0;
        end
    end
    else                                                  %does not collide
        NewPos = 0;
    end

end
```

For the sphere case all code is included in one script without any self made external functions. The following script is for a pulsed gradient, this is put in a separate script for convenience, but the scrips for the other gradients is in essence the same.

```
%********** Parameters **********
N = 100000;                          %Total number of particles.
R = 75;                              %Radius sphere [\mu m]
D = 1.65;                            %Diffusion coefficient [(\mu m)^2 /ms]
gamma = 2.675*10^8;                  %Gyromagnetic ratio rad/(T*s)
G_x = 26434:10000:610480;            %Gradient strength [mT/m]
diff_times = 30:5:200;               %Diffusion times [ms]
dt = 0.001;                          %Time step [ms]
%******************************

dr = sqrt(6*D*dt);                   %Stepsize [\mu m]

N_G = length(G_x);                   %Number of different gradients.
N_time = length(diff_times);         %Number of different diffusion times.
N_time_N_G = N_time*N_G;             %Total number of resulting phases/file.

limit = round(diff_times/dt);           %Finding the step numbers
for i = 1:N_time                        %associated with the different
    if limit(i)/2 ~= round(limit(i)/2) %diffusion times.
        limit(i) = limit(i) + 1;        %And forces it to be even.
    end
end

K = max(limit);                         %Total number of steps.

phase_1 = zeros(N,N_time_N_G);       %Initilizing arrays
```

```
27
28  %***** Finding initial positions for all particles *****
29  cos_theta = 2*rand(1,N)-1;
30  theta = acos(cos_theta);
31  phi = 2*pi*rand(1,N);
32  r0 = rand(1,N)*R;
33  x0 = r0.*sin(theta).*cos(phi);
34  y0 = r0.*sin(theta).*sin(phi);
35  z0 = r0.*cos_theta;
36  %*******************************************************
37
38  string1 = ['Number of steps : ', num2str(K)];
39  disp(string1)
40
41  Crash_nr = 0;                          %Keeps track of number of collitions.
42
43  %(I)****************** Loop over all the particles **********************
44  parfor n = 1:N
45      %%
46      if n/1000 == round(n/1000)
47          string1 = ['ParticleNr : ', num2str(n), ' of ',num2str(N)];
48          disp(string1)
49      end
50      %%
51
52      x_pos = zeros(1,K);     %Initilizing arrays to
53      y_pos = x_pos;          %save all positions.
54      z_pos = x_pos;          %
55
56      x_pos(1) = x0(n);       %Adds the initial position.
57      y_pos(1) = y0(n);       %
58      z_pos(1) = z0(n);       %
59
60      %(II)*********** Finds position 2-K ***************
61      k = 2;
62      while k <= K
63
64          %***** Find number of steps before *****
65          %***** collition check is needed.  *****
66          dist = R - norm([x_pos(k-1) ...      %Distance to sphere wall.
67          y_pos(k-1) ...          %
68          z_pos(k-1)]);           %
69          dK = floor(dist/norm(dr));          %Number of steps to wall.
70          %*************************************
71
72          if dK > 0               %Enters to find dK new positions.
73
74              if dK > K-k+1                   %Hinders the program to run
75                  dK = K-k+1;                 %past total diffusion time.
76                  if dK == 0                  %
77                      k = K + 1;              %
78                  end                         %
79              end                             %
80
81              d_cos_theta = 2*rand(1, dK) - 1;        %Computes random
82              d_theta = acos(d_cos_theta);            %directions.
83              d_phi   = 2*pi*rand(1, dK);             %
84
85              dx_vec = dr*sin(d_theta).*cos(d_phi);   %Finds the
86              dy_vec = dr*sin(d_theta).*sin(d_phi);   %steps.
87              dz_vec = dr*d_cos_theta;                %
```

```
88
89              x_pos(k:k+dK-1) = cumsum(dx_vec) + x_pos(k-1);  %Finds and
90              y_pos(k:k+dK-1) = cumsum(dy_vec) + y_pos(k-1);  %saves the new
91              z_pos(k:k+dK-1) = cumsum(dz_vec) + z_pos(k-1);  %positions.
92
93              k = k + dK;
94
95          else                        %Enters when next step may give collition.
96
97              pos = [x_pos(k-1) y_pos(k-1) z_pos(k-1)];   %Current position.
98
99              d_cos_theta = 2*rand - 1;                   %Finds direction.
100             d_theta = acos(d_cos_theta);                %
101             d_phi   = 2*pi*rand;                        %
102
103             dx = dr*sin(d_theta).*cos(d_phi);           %Computes
104             dy = dr*sin(d_theta).*sin(d_phi);           %the step.
105             dz = dr*d_cos_theta;                        %
106
107             v = [dx dy dz]/dt;                          %Finds velocity.
108
109             ddt = dt;                       %Variable to get correct total
110                                             %stepsize during collition.
111
112             %(III)***** Loop in case of multiple collition **********
113             %********** in same time step.                 **********
114             collide = 1;
115             while collide == 1
116
117                 %***** Computes time until collition from current *****
118                 %***** position given the velocity v.             *****
119                 A = dot(pos,v);
120                 B = dot(v,v);
121                 C = sqrt(A^2 - B*(dot(pos,pos) - R^2));
122                 t_c = (-A + C)/B;
123                 if t_c < 0
124                     t_c = (-A - C)/B;
125                 end
126                 %*****************************************************
127
128                 if t_c >= ddt || t_c < 0        %The step do not
129                     collide = 0;                %lead to collition.
130                 end                             %
131
132                 if isreal(t_c) == 0                 %Hope it never
133                     disp('something is wrong:')     %enters this one.
134                     k = K + 1;                      %
135                 end                                 %
136
137                 if collide == 1                         %Collition
138                     Crash_nr = Crash_nr + 1;
139
140                     normal = pos + v*t_c;               %Surface normal in
141                     normal = normal/norm(normal);       %collition point.
142
143                     v_new = v - 2*dot(v,normal)*normal;   %Computes new
144                     v_new = v_new*norm(v)/norm(v_new);    %velocity.
145
146                     pos = pos + v*t_c + v_new*(ddt-t_c);  %New position.
147
148                     v = v_new;                            %Updates v.
```

```
149                     ddt = ddt-t_c;                          %Updates ddt in
150                                                             %case of multiple
151                                                             %colliton in one
152                                                             %time step.
153
154                 else                                        %Not collition
155
156                     x_pos(k) = x_pos(k-1) + v(1)*ddt;   %Finds new
157                     y_pos(k) = y_pos(k-1) + v(2)*ddt;   %positions.
158                     z_pos(k) = z_pos(k-1) + v(3)*ddt;   %
159
160                 end
161             end
162             %(III)*************************************************
163             k = k + 1;
164         end
165     end
166     %(II)*********************************************
167
168     %(IV)* Adding the gradient, and computes final phase *****
169     %***** for three different sequenses, and all the     *****
170     %***** different time steps and gradient strengths.   *****
171     n_time = 1;
172     n_G = 1;
173     for Nr = 1:N_time_N_G
174         Y = G_x(n_G)*dt*gamma;
175
176         %***I: +G first step, -G last step***
177         d_phase = [Y*x_pos(1) -Y*x_pos(limit(n_time))];
178         phase(n,Nr) = sum(d_phase)*10^-12;
179
180         n_time = n_time + 1;
181
182         if Nr == N_time*n_G
183             n_time = 1;
184             n_G = n_G + 1;
185         end
186     end
187     %(IV)*****************************************************
188
189 end
190 %(I)*************************************************************************
191
192 string1 = ['Number of collitions: ', num2str(Crash_nr)];
193 disp(string1)
194
195 signal = sum(exp(1i*phase));      %Computes resulting phase
196
197 Times = limit*dt;
198
199 signal_matrix = zeros(N_G+1,N_time+1);   %Creates the output matrixes,
200 signal_matrix(2:end,1) = G_x;            %first column and first row
201 signal_matrix(1,2:end) = Times;          %specifies gradient strength
202
203 %***** Saves all phases into the matrix *****
204 A = 1;
205 B = N_time;
206 for i = 2:N_G+1
207 signal_matrix(i,2:end) = signal(A:B);
208 A = A + N_time;
209 B = B + N_time;
```

```
210  end
211  %****************************************
212
213  signal = signal_matrix;
214
215  %*****time in ms, G in mT/m*****
216  save SignalPulsedGrad signal
217  %************************************************************
```

Bibliography

[1] Fieremans E, Novikov DS, Jensen JH, Helpern JA. Monte Carlo study of a two-compartment exchange model of diffusion. NMR Biomed. 2010;23(7):711-724.

[2] Codling EA, Plank MJ, Benhamou S. Random walk models i biology. J. R. Soc. Interface. 2008;5:813-834

[3] Bundell SJ, Bundell KM. Concepts in Thermal Physics. Second edition. New York: Oxford University Press; 2010

[4] Sparr G, Sparr A. Kontinuerliga system. Lund: Studentlitteratur AB; 1999, 2000

[5] Ursell TS. The Diffusion Equation A Multi-dimensional Tutorial. Pasadena: California Institute of Technology; October 2007. Available from `http://www.rpgroup.caltech.edu/~natsirt/aph162/diffusion_old.pdf`

[6] Sen PN. Time-Dependent Diffusion Coefficient as a Probe of Geometry. Concepts in Magnetic Resonance Part A. 2004;23A(1):1-21.

[7] K.G. Helmer, M.D. Hurlimann, T.M. Deswiet, P.N. Sen, C.H. Sotak. Determination of Ratio of Surface Area to Pore Volume Restricted Diffusion in a Constant Field Gradient. JMR. 1995;115:257-259.

[8] Lilley J. Nuclear Physics, Principles and Applications. Chichester: John Wiley & Sons Ltd; 2001

[9] Basser PJ, Özarslan E. Introduction to Diffusion MR. I:Johansen-Berg H, Behrens TEJ, red. Diffusion MRI From Quantitative Measurements to In vivo Neuroanatomy. Elsecier Inc; 2009. 3-10.

[10] Sukstanskii AL, Yablonskiy DA. Effects of Restricted Diffusion on MR Signal Formation. JMR. 2002;157:92-105.

[11] Lori NF, Conturo TE, Bihan DL. Definition of displacement probability and diffusion time in q-space magnetic resonance measjurements that use finite-duration diffusion-encoding gradients. 2003;165:185-195

[12] Hall MG, Alexander DC. Convergence and Parameter Choice for Monete-Carlo Simulations of Diffsuion MRI. IEEE Trans Med Imaging. 2009;28(9):1354-64.

[13] Marschner S. Simple ray-triangle intersection. Cornell University; October 2003. Available from `http://www.cs.cornell.edu/courses/cs465/2003fa/homeworks/raytri.pdf`

[14] Shirley P. Fundamentals of Computer Graphics. Massachusetts: A K Peters; 2002

[15] Matthews PC. Vector Calculus. Great Britain: Springer-Verlag London Limited; 1998

[16] White NS, DALE AM. Distinct Effects of Nuclear Volume Fraction and Cell Diameter on High b-value Diffusion MRI Contrast in Tumors. Magn Reson Med. 2014;72(5):1435-43.

[17] Yeh CH, Schmitt B, Bihan DL, Li-Schlittgen JR, Lin CP, Poupon C. Diffusion Microscopist Simulator: A General Monte Carlo Simulation System for Diffusion Magnetic Resonance Imaging. PLoS One. 2013;8(10):e76626.

[18] Inferring cellular geometry in the short diffusion time limit in a clinical MRI scanner.
Specialization project, Jacob Prescott, supervisor Pål Erik Goa. [1]

[1] Not published