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 Mat-\lab, som egner seg til å simulere diffusjons MRI. Implementeringen består av simulering av diffusjon ved hjelp av Monte Carlo metoder, deteksjon av hindringer og kollisjons-interaksjon, og inkludering av grådienter 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

1 Introduction

2 Theory
   2.1 Diffusion
   2.2 Magnetic Resonance Imaging
   2.3 Reflection and Surface Detection

3 Method
   3.1 Random Walk
   3.2 Detection and Interaction With Obstacles
   3.3 Gradient and Signal
   3.4 Simulations

4 Results
   4.1 Random Walk
   4.2 Reflection Inside a Non-Spherical Obstacle
   4.3 Pulsed Gradient on Free Diffusion
   4.4 Spin Echo for Diffusion in Sphere
   4.5 FID Signal for Diffusion in Sphere

5 Discussion and Conclusions
   5.1 Performance and Execution Time
   5.2 Random Walk and Obstacle Interaction
   5.3 Signal and ADC
   5.4 Other
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]

$$\frac{\partial q}{\partial t} + \nabla \cdot \mathbf{j} = k.$$  \hspace{1cm} (2.1)

Here, $q$ is the concentration of the system in question, $\mathbf{j}$ 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];

$$\mathbf{j} = -D \nabla q.$$  \hspace{1cm} (2.2)

Substituting this into the continuity equation gives the diffusion equa-
tion

$$\frac{\partial q}{\partial t} = D \nabla^2 q.$$  \hspace{1cm} (2.3)
This is a partial differential equation, and solving this for the case of free diffusion results in the Green function given by \[ G(r, r', t) = \exp\left(-\frac{|r-r'|^2}{4Dt}\right)\left(\frac{4\pi Dt}{d/2}\right). \] (2.4)

Here, \( r \) is the final position and \( r' \) 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
\[
\langle r \rangle = r'.
\] (2.5)

The second moment gives the expectation value of the squared position
\[
\langle r^2 \rangle = |r'|^2 + 2dDt.
\] (2.6)

From this one can see that the mean square displacement a particle has moved during time \( t \) becomes
\[
\langle (r - r')^2 \rangle = \langle r^2 \rangle + r'^2 - 2r'\langle r \rangle = 2dDt.
\] (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
2.2. MAGNETIC RESONANCE IMAGING

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 depen-
dent 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 obsta-
cle, the short time behavior of $D(t)$ may be described by [6] [7]

$$D(t) = D_0 \left(1 - k \frac{S}{V} (D_0 t)^{1/2}\right) + O(D_0 t). \quad (2.8)$$

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 $B_0$ has two avail-
able energy states $\pm \mu_p B_0$, where $\mu_p$ 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 a resulting magnetization vector $M$. But, because the spins
are not in phase, $M$ is static aligned with $B_0$ and no signal is present.

The spin’s precession frequency is given by $\omega = -\gamma B_0$ and is know
as the Larmor frequency, $\gamma$ 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 $B_0$ and
now rotates around this, and this rotating magnetization vector is the
MR signal. Keeping the RF-signal on for the time it takes $M$ to reach
down to the plane perpendicular to $B_0$ defines $T_1$, 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 \( B_0 \) back to aligning with \( B_0 \) (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 \( \omega \), the phase at time \( t \) will be given by \( \phi = \omega t + \phi_0 \), where \( \phi_0 \) 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 \( \delta \) 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 at 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 \( dt \), the phase change resulted from being exposed to a gradient \( G_x \) in the \( x \)-direction is given by

\[
\phi(x) = -\omega(x)dt = -\gamma(B_0 + G_xx)dt. \tag{2.9}
\]

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

\[
\phi_2 - \phi_1 = -\gamma\delta\left[B_0 + G_x x_1 - B_0 - G_x x_2\right] = -\gamma\delta G(x_2 - x_1). \tag{2.10}
\]

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 \( \rho(x_1) \), the diffusion propagator \( P(x_1, x_2, \Delta) \) (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

\[
E(q) = \int \int \rho(x_1)P(x_1, x_2, \Delta)e^{-iq(x_2-x_1)}dx_1dx_2. \tag{2.11}
\]

For free diffusion, this results in a Gaussian phase distribution

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

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

\[
E(b) = e^{-bD}, \tag{2.13}
\]

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, \( \Delta \), \( \Delta + \delta \) and \( \Delta - \delta/3 \) is used [11]. Though it seems quite settled to use \( \Delta - \delta \) for the pulsed gradient cases.
CHAPTER 2. THEORY

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 $t_c \equiv \frac{1}{\gamma GR}$. Measuring the signal as a function of the dimensionless parameter $\tau = \frac{t}{t_c}$, the signal will depend on $p$ given by

\[ p = \frac{t_c}{t_d} = \frac{D}{\gamma G x R^3}. \quad (2.14) \]

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

2.3 Reflection and Surface Detection

Given a particle with initial position $r$ and a constant velocity $v$, its trajectory is described by $r' = r + vt$. 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 $r'$ lies on a distance $R$ from the origin, i.e. $R^2 = |r'|^2$. Substituting this into the equation for the particle trajectory and solving for $t$ gives the following equation for the collision time;

\[ t_c = \frac{-r \cdot v \pm \sqrt{(r \cdot v)^2 - v^2(r^2 - R^2)}}{v^2}. \quad (2.15) \]

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

\[ t_c = \frac{(p - r) \cdot \hat{n}}{v \cdot \hat{n}} \quad (2.16) \]

Here $p$ is a point in the plane, and $\hat{n}$ 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]

\[ (p_j - p_i) \times (r_c - p_i) \cdot \hat{n} \geq 0. \quad (2.17) \]
2.3. REFLECTION AND SURFACE DETECTION

\( r_c \) is here the point where the particle hits the plane, and \( p_i \) 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 \( v' = v_x \hat{x} - v_z \hat{z} \), 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 \( v \cdot \hat{n} \) and \( v_x = |v| |\hat{n}| \sin(\theta) = |v \times \hat{n}| \). Noticing \( v \times \hat{n} \) points in the negative \( y \) direction this is forced in the correct direction by crossing it with \(-\hat{n}\). Following, the new velocity can be expressed \( v' = (v \times \hat{n}) \times (-\hat{n}) - (v \cdot \hat{n})\hat{n} \). Using suffix notation [15] and rearrange the double cross product may be simplified to

\[
\begin{align*}
(v \times \hat{n}) \times \hat{n} &= \epsilon_{ijk} (v \times \hat{n})_j n_k \\
&= \epsilon_{ijk} \epsilon_{jlm} v_l n_m n_k = \epsilon_{kij} \epsilon_{jlm} v_l n_m n_k \\
&= (\delta_{kl} \delta_{im} - \delta_{km} \delta_{il}) v_l n_m n_k = v_k n_i n_k - v_i n_k n_k \\
&= (v \cdot \hat{n}) \hat{n} - v.
\end{align*}
\]

(2.18)

Here \( \epsilon \) is the alternating tensor and \( \delta_{ij} \) is the Kronecker delta. Now substituting this into the expression for \( v' \) gives

\[
v' = -((v \cdot \hat{n}) \hat{n} - v) - (v \cdot \hat{n}) \hat{n} = v - 2(v \cdot \hat{n}) \hat{n}.
\]

(2.19)
Figure 2.1: A particle with initial velocity $v = v_x \hat{x} + v_z \hat{z}$ 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 $\gamma$, the number of spins $N$, the gradient strengths $G$ and the total simulation time $T$ or the gradient time specifications $\delta$ and $\Delta$.

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 $\phi$ is determined by a random number between zero and $2\pi$, while the polar angle $\theta$ 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 \texttt{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 \texttt{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 \(X(i, j)\hat{x} + Y(i, j)\hat{y} + Z(i, j)\hat{z}\). 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
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 $v_1$ and $v_2$ 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 $v = dr/dt$, 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 in 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.

Because $\hat{n}$ 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 $v_1$ and $v_2$. 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.
3.3. GRADIENT AND SIGNAL

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 $\hat{n} \cdot \vec{v} > 0$. The new position is found by setting the length of the new velocity equal the old, and let the spin move $\vec{v}t_c + \vec{v}'(dt - t_c)$. 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 $dt$, and exposed to a gradient $G_x$ is given by $d\phi = \gamma dt G_x x$. 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
CHAPTER 3. METHOD

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 $\delta/dt$ 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

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

Several gradients, $\delta$ and $\Delta$ 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 $dt = 0.001$ 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 $10 \text{ s/mm}^2 < b < 800 \text{ s/mm}^2$, 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 function 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 $D = 1 \mu\text{m}^2/\text{ms}$ which is taken from [16], where it is used for modeling extracellular water. The total diffusion time is $T = 100$ ms, and the
3.4. SIMULATIONS

The simulations are 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 $\rho(r) \times r$, where $\rho(r)$ 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 $D_0 = 1.65 \, \mu m^2/\text{ms}$.

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 $\mu$m with impermeable walls. The intrinsic diffusion coefficient is set to $D_0 = 1.65 \, \mu m^2/\text{ms}$. 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 were performed. Four different setups for the spin echo sequences are used, pulsed gradient, constant gradient with no break between the positive and the negative parts, 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 $\delta = 0.001 \text{ms}$ corresponding to adding the gradient over only one time step, this is done to let the gradient approach a Dirac delta function and gives $T \approx \Delta$. 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, $\delta$ and $\Delta$ 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
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 $b \leq 800 \text{ s/mm}^2$, and meaning over these gave the diffusion coefficient $D_0 = (1.656 \pm 0.005) \text{ µm}^2/\text{ms}$.

### 4.4 Spin Echo for Diffusion in Sphere

**Pulsed Gradient**

The calculation for the pulsed gradient includes 35 different simulation times with $T_{\text{min}} = 30$ ms and $T_{\text{max}} = 200$ ms, and 60 different gradient strengths with $G_{\text{min}} = 26434 \text{ mT/m}$ and $G_{\text{max}} = 606434 \text{ mT/m}$. 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
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. $r = 72.1 \mu m$ and $r = 72.5 \mu m$ including and not including $D_0$ respectively. Doing a rerun with the same parameters only changing the sphere radius to $85.0 \mu m$ gave $r = 95.6 \mu m$ and $r = 96.0 \mu m$ 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.
### CHAPTER 4. RESULTS

<table>
<thead>
<tr>
<th>Analytical</th>
<th>Mean</th>
<th>Standard deviation</th>
<th>Kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>14.14</td>
<td>3.00</td>
</tr>
</tbody>
</table>

| $N = 10K$  | $x$  -0.23 | 14.15 | 2.98 |
| $N = 50K$  | $x$  0.01  | 14.21 | 2.94 |
| $N = 100K$ | $x$  0.06  | 14.19 | 3.00 |
| $N = 400K$ | $x$  0.01  | 14.17 | 3.01 |

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

<table>
<thead>
<tr>
<th>$R = 65 \mu m$</th>
<th>Not including $D_0$</th>
<th>Including $D_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>55.0</td>
<td>55.3</td>
</tr>
</tbody>
</table>

| $R = 75 \mu m$ | 72.1 | 72.5 |
| $R = 75 \mu m$ | 73.0 | 73.4 |
| $R = 85 \mu m$ | 95.6 | 96.0 |

Table 4.2: Radii in $\mu m$ for the pulsed gradient simulations, calculated from the ADC regression lines. ADC for simulation 1, 75 $\mu m$, is shown in Fig. 4.6
4.4. SPIN ECHO FOR DIFFUSION IN SPHERE

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 is 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.
CHAPTER 4. RESULTS

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.
4.4. **SPIN ECHO FOR DIFFUSION IN SPHERE**

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/S₀) is plotted for both the simulated and the experimental ([18]) data with the following values: δ = 7.8 ms, 21.8 ms, 26.8 ms and Δ = 16.6 ms, 27.6 ms, 32.6 ms. These values corresponds to
### CHAPTER 4. RESULTS

<table>
<thead>
<tr>
<th>Diffusion time</th>
<th>Without break</th>
<th>With 5 ms break</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta - \delta/3$</td>
<td>65.9, 68.0</td>
<td>63.0</td>
<td>Not including $D_0$</td>
</tr>
<tr>
<td></td>
<td>71.9, 74.1</td>
<td>70.8</td>
<td>Including $D_0$</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>80.7, 83.3</td>
<td>79.4</td>
<td>Not including $D_0$</td>
</tr>
<tr>
<td></td>
<td>88.0, 90.8</td>
<td>87.5</td>
<td>Including $D_0$</td>
</tr>
<tr>
<td>$\Delta + \delta$</td>
<td>114.1, 117.7</td>
<td>115.8</td>
<td>Not including $D_0$</td>
</tr>
<tr>
<td></td>
<td>124.5, 128.4</td>
<td>125.3</td>
<td>Including $D_0$</td>
</tr>
</tbody>
</table>

Table 4.3: Radii in $\mu$m for the constant gradient simulations, calculated from the ADC regression lines, for the 75 $\mu$m 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.

<table>
<thead>
<tr>
<th>Diffusion time</th>
<th>Without break</th>
<th>With 5 ms break</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta - \delta/3$</td>
<td>82.6, 74.1</td>
<td>78.4</td>
<td>Not including $D_0$</td>
</tr>
<tr>
<td></td>
<td>89.9, 81.2</td>
<td>87.9</td>
<td>Including $D_0$</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>101.2, 90.8</td>
<td>98.6</td>
<td>Not including $D_0$</td>
</tr>
<tr>
<td></td>
<td>110.1, 99.5</td>
<td>108.6</td>
<td>Including $D_0$</td>
</tr>
<tr>
<td>$\Delta + \delta$</td>
<td>143.1, 128.4</td>
<td>143.7</td>
<td>Not including $D_0$</td>
</tr>
<tr>
<td></td>
<td>156.7, 140.7</td>
<td>155.5</td>
<td>Including $D_0$</td>
</tr>
</tbody>
</table>

Table 4.4: Radii in $\mu$m for the constant gradient simulations, calculated from the ADC regression lines, for the 85 $\mu$m 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.
4.5. **FID SIGNAL FOR DIFFUSION IN SPHERE**

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.

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 $\tau$, for three different values of $p$. 

the experimental echo times: 38 ms, 60 ms and 70 ms.
CHAPTER 4. RESULTS

(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.
4.5. FID SIGNAL FOR DIFFUSION IN SPHERE

Figure 4.10: Logarithmic plot of simulated and experimental signal. The lines with the same color and scattering symbols correspond to the same $\delta$ and $\Delta$, and the densest lines corresponds to the simulated results.

Figure 4.11: FID signal plotted as function of the dimensionless parameter $\tau$ for high $b$-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.
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.

```matlab
1 %******************************************************************************
2 d*cos_theta = 2*rand(1, K) - 1;
3 d_theta = acos(d*cos_theta);
4 d_phi = 2*pi*rand(1, K);
5 x = cumsum([0 dr*sin(d_theta).*cos(d_phi)]);
6 y = cumsum([0 dr*sin(d_theta).*sin(d_phi)]);
7 z = cumsum([0 dr*d*cos_theta]);
8 %******************************************************************************
```

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.

```matlab
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %Returns the indexes belonging to square/triangle in Obstacle [X,Y,Z] 
3 %with grid points nearest (x,y,z) .
4 %Gives out the points counterclockwise from outside obstacle clockwise from inside obstacle
5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6 function MinDist = Get4ClosestDist(x, y, z, X, Y, Z, ObsSize)
7 8 Shape = 0;  %Shape = 0  =>  square
9  %Shape = 1  =>  triangle
10  %Shape = 2 or 3  =>  first: square or triangle
11  % then : triangle, top or bottom
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

35
%******** Initializing ********
 InitialIndex1 = round(ObsSize/2);
 InitialIndex2 = InitialIndex1 + 1;

 index1_1 = InitialIndex1;
 index1_2 = InitialIndex1;
 index2_1 = InitialIndex2;
 index2_2 = InitialIndex2;

 Dist1 = (X(InitialIndex1, InitialIndex1) - x)**2 + ...
       (Y(InitialIndex1, InitialIndex1) - y)**2 + ....
       (Z(InitialIndex1, InitialIndex1) - z)**2 + 1;

%******** Finds the closest gridpoint ********
 for j = 1 : ObsSize
     for k = 1 : ObsSize - 1
         Dist = (X(j,k) - x)**2 + ...
             (Y(j,k) - y)**2 + ....
             (Z(j,k) - z)**2 ;
         if Dist < Dist1
             index1_1 = j;
             index1_2 = k;
             Dist1 = Dist;
         end
     end
 end

%******** Find the closest of the neighboring points ********
 I1 = index1_1 - 1;
 I2 = index1_1 + 1;
 I3 = index1_2 - 1;
 I4 = index1_2 + 1;
 if index1_2 == 1
     I3 = ObsSize - 1;
 end
 if index1_1 == 1
     Shape = 1;
     index2_1 = 2;
     I1 = index2_1;
 end
 if index1_1 == ObsSize
     Shape = 1;
     index2_1 = ObsSize - 1;
     I1 = index2_1;
 end
 if index1_1 == 2
     Shape = 2;
 end
 if index1_1 == ObsSize - 1
     Shape = 3;
 end
 if Shape == 0
     DistNeighbor1 = (X(I1,index1_2) - x)**2 + ...
                     (Y(I1,index1_2) - y)**2 + ....
                     (Z(I1,index1_2) - z)**2;
     DistNeighbor2 = (X(I2,index1_2) - x)**2 + ...
                     (Y(I2,index1_2) - y)**2 + ....
\[ \text{DistNeighbor3} = (X(\text{index1}_1, I3) - x)^2 + (Y(\text{index1}_1, I3) - y)^2 + \ldots \]
\[ (Z(\text{index1}_1, I3) - z)^2; \]
\[ \text{DistNeighbor4} = (X(\text{index1}_1, I4) - x)^2 + (Y(\text{index1}_1, I4) - y)^2 + \ldots \]
\[ (Z(\text{index1}_1, I4) - z)^2; \]
\[ [M, I] = \min([\text{DistNeighbor1} \ \text{DistNeighbor2} \ \text{DistNeighbor3} \ \text{DistNeighbor4}]); \]

if \( I == 1 \)
\[ \text{index2}_1 = I1; \]
\[ \text{index2}_2 = \text{index1}_2; \]
else if \( I == 2 \)
\[ \text{index2}_1 = I2; \]
\[ \text{index2}_2 = \text{index1}_2; \]
else if \( I == 3 \)
\[ \text{index2}_1 = \text{index1}_1; \]
\[ \text{index2}_2 = I3; \]
else
\[ \text{index2}_1 = \text{index1}_1; \]
\[ \text{index2}_2 = I4; \]
end

else if \( \text{Shape} == 1 \)
\[ \text{Dist} = \text{zeros}(1, \text{ObsSize} - 1); \]
for \( k = 1 : \text{ObsSize} - 1; \)
\[ \text{Dist}(k) = (X(I1, k) - x)^2 + (Y(I1, k) - y)^2 + (Z(I1, k) - z)^2; \]
end
\[ [M, I] = \min(\text{Dist}); \]
\[ \text{index2}_2 = I; \]
else if \( \text{Shape} == 2 \)
\[ \text{DistNeighbor1} = (X(I1, \text{index1}_2) - x)^2 + (Y(I1, \text{index1}_2) - y)^2 + \ldots \]
\[ (Z(I1, \text{index1}_2) - z)^2; \]
\[ \text{DistNeighbor2} = (X(I2, \text{index1}_2) - x)^2 + (Y(I2, \text{index1}_2) - y)^2 + \ldots \]
\[ (Z(I2, \text{index1}_2) - z)^2; \]
\[ \text{DistNeighbor3} = (X(\text{index1}_1, I3) - x)^2 + (Y(\text{index1}_1, I3) - y)^2 + \ldots \]
\[ (Z(\text{index1}_1, I3) - z)^2; \]
\[ \text{DistNeighbor4} = (X(\text{index1}_1, I4) - x)^2 + (Y(\text{index1}_1, I4) - y)^2 + \ldots \]
\[ (Z(\text{index1}_1, I4) - z)^2; \]
\[ [M, I] = \min([\text{DistNeighbor1} \ \text{DistNeighbor2} \ \text{DistNeighbor3} \ \text{DistNeighbor4}]); \]
if \( I == 1 \)
\[ \text{index2}_1 = I1; \]
\[ \text{index2}_2 = \text{index1}_2; \]
else if \( I == 2 \)
\[ \text{index2}_1 = I2; \]
\[ \text{index2}_2 = \text{index1}_2; \]
else if \( I == 3 \)
\[ \text{index2}_1 = \text{index1}_1; \]
\[ \text{index2}_2 = I3; \]
else
\[ \text{index2}_1 = \text{index1}_1; \]
\[ \text{index2}_2 = I4; \]
end
else if \( \text{Shape} == 3 \)
\[ \text{DistNeighbor1} = (X(I1, \text{index1}_2) - x)^2 + (Y(I1, \text{index1}_2) - y)^2 + \ldots \]
\[ (Z(I1, \text{index1}_2) - z)^2; \]
\[ \text{DistNeighbor2} = (X(I2, \text{index1}_2) - x)^2 + (Y(I2, \text{index1}_2) - y)^2 + \ldots \]
\[ (Z(I2, \text{index1}_2) - z)^2; \]
\[ \text{DistNeighbor3} = (X(\text{index1}_1, I3) - x)^2 + (Y(\text{index1}_1, I3) - y)^2 + \ldots \]
\[ (Z(\text{index1}_1, I3) - z)^2; \]
\[ \text{DistNeighbor4} = (X(\text{index1}_1, I4) - x)^2 + (Y(\text{index1}_1, I4) - y)^2 + \ldots \]
(Z(index1_1, I4) - z).^2;

[M, I] = min([DistNeighbor1 DistNeighbor2 DistNeighbor3 DistNeighbor4]);

if I == 1
    index2_1 = I1;
    index2_2 = index1_2;
    Shape = 0;
elseif I == 2
    index2_1 = I2;
    index2_2 = index1_2;
    Shape = 1;
elseif I == 3
    index2_1 = index1_1;
    index2_2 = I3;
    Shape = 2;
else
    index2_1 = index1_1;
    index2_2 = I4;
    Shape = 2;
end

%******************************************************************************
%*****Finding the two (one) other points******************************************************************************
if Shape == 0
    if index1_2 == index2_2
        I1 = index1_1 - 1;
        I2 = index1_1 + 1;
        Dist1 = (X(I1, index1_2) - x).^2 ... + (Y(I1, index1_2) - y).^2 ... + (Z(I1, index1_2) - z).^2;
        Dist2 = (X(I2, index1_2) - x).^2 ... + (Y(I2, index1_2) - y).^2 ... + (Z(I2, index1_2) - z).^2;
        Dist3 = (X(I1, index2_2) - x).^2 ... + (Y(I1, index2_2) - y).^2 ... + (Z(I1, index2_2) - z).^2;
        Dist4 = (X(I2, index2_2) - x).^2 ... + (Y(I2, index2_2) - y).^2 ... + (Z(I2, index2_2) - z).^2;
        [M, I] = min([Dist1 Dist2 Dist3 Dist4]);
    else
        index3_1 = I1;
        index3_2 = index1_2;
        index4_1 = I1;
        index4_2 = index2_2;
        if Shape == 2 || I == 4
            index3_1 = I2;
            index3_2 = index1_2;
            index4_1 = I2;
            index4_2 = index2_2;
            end
        elseif index1_2 == index2_2
            I1 = index1_2 - 1;
            I2 = index1_2 + 1;
            if index1_2 == 1
                I1 = ObsSize - 1;
            elseif index1_2 == ObsSize
                I2 = 2;
            end
            Dist1 = (X(index1_1, I1) - x).^2 ... + (Y(index1_1, I1) - y).^2 ... + (Z(index1_1, I1) - z).^2;
Dist2 = (X(index1, \ldots, I2) - x)^2 \ldots
+ (Y(index1, \ldots, I2) - y)^2 \ldots
+ (Z(index1, \ldots, I2) - z)^2;

Dist3 = (X(index2, \ldots, I1) - x)^2 \ldots
+ (Y(index2, \ldots, I1) - y)^2 \ldots
+ (Z(index2, \ldots, I1) - z)^2;

Dist4 = (X(index2, \ldots, I2) - x)^2 \ldots
+ (Y(index2, \ldots, I2) - y)^2 \ldots
+ (Z(index2, \ldots, I2) - z)^2;

\[ [M, I] = \min([\text{Dist1 Dist2 Dist3 Dist4}]); \]

if I == 1 || I == 3
    index3 = index1;
    index4 = I1;
else
    index3 = index1;
    index4 = I2;
end

end

if Shape == 1
    if index1 == 1
        index3 = 2;
        I1 = index2 - 1;
        I2 = index2 + 1;
        if index2 == 1
            I1 = ObsSize - 1;
        end
        if index2 == ObsSize
            I2 = 2;
        end
    end
    if index2 == ObsSize
        I2 = 2;
    end
end

else if index2 == 1
    index3 = 2;
    I1 = index1 - 1;
    I2 = index1 + 1;
    if index1 == 1
        I1 = ObsSize - 1;
    end
    if index1 == ObsSize
        I2 = 2;
    end
end

end

if index1 == 1
    index3 = 1;
    I1 = index2; %test
    I2 = index2 + 1;
    if index2 == 1
        I1 = ObsSize - 1;
    end
    if index2 == ObsSize
        I2 = 2;
    end
end

else if index2 == 1
    index3 = 2;
    I1 = index1 - 1;
    I2 = index1 + 1;
    if index1 == 1
        I1 = ObsSize - 1;
    end
    if index1 == ObsSize
        I2 = 2;
    end
end

if index2 == 1
    index3 = 2;
    I1 = index1 - 1;
    I2 = index1 + 1;
    if index1 == 1
        I1 = ObsSize - 1;
    end
    if index1 == ObsSize
        I2 = 2;
    end
end

\[ [M, I] = \min([\text{Dist1 Dist2 Dist3 Dist4}]); \]

if I == 1
    index3 = I1;
else
    index3 = I2;
end

\[ \text{Dist} = [(X(2, I1) - x)^2 + (Y(2, I1) - y)^2 + (Z(2, I1) - z)^2 \ldots
+ (X(2, I2) - x)^2 + (Y(2, I2) - y)^2 + (Z(2, I2) - z)^2]; \]

\[ [M, I] = \min(\text{Dist}); \]

if I == 1
    index3 = I1;
else
    index3 = I2;
end
APPENDIX A. CODE

```plaintext
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   end
269   dist = [(X(index3,1,I1)-x)^2 + (Y(index3,1,I1)-y)^2 ...
270       + (Z(index3,1,I1)-z)^2 ...
271       (X(index3,1,I2)-x)^2 + (Y(index3,1,I2)-y)^2 + ...
272       (Z(index3,1,I2)-z)^2 ];
273   [M,I] = min(dist);
274   if I == 1
275     index3,2 = I1;
276   else
277     index3,2 = I2;
278   end
279   elseif index2,1 == ObsSize
280     index3,1 = ObsSize - 1;
281     I1 = index1,2 - 1;
282     I2 = index1,2 + 1;
283     if index1,2 == 1
284       I1 = ObsSize - 1;
285   end
286   if index1,2 == ObsSize
287     I2 = 2;
288   end
289   end
290   dist = [(X(index3,1,I1)-x)^2 + (Y(index3,1,I1)-y)^2 + ...
291       (Z(index3,1,I1)-z)^2 ...
292       (X(index3,1,I2)-x)^2 + (Y(index3,1,I2)-y)^2 + ...
293       (Z(index3,1,I2)-z)^2 ];
294   [M,I] = min(dist);
295   if I == 1
296     index3,2 = I1;
297   else
298     index3,2 = I2;
299   end
300   end
301   MinDist = [index1,1, index1,2, index2,1, index2,2, ...
302       index3,1, index3,2];
303   end
304   if Shape == 2 %square or triangle
305     if index1,1 == 2
306       dist = [(X(1,1)-x)^2 (Y(1,1)-y)^2 (Z(1,1)-z)^2 ...
307         (X(3,index1,2)-x)^2 (Y(3,index1,2)-y)^2 (Z(3,index1,2)-z)^2 ...
308         (X(3,index2,2)-x)^2 (Y(3,index2,2)-y)^2 (Z(3,index2,2)-z)^2 ];
309       [M,I] = min(dist);
310       if I == 1
311         index3,1 = 1;
312         index3,2 = 1;
313         Shape = 1;
314       else
315         index3,1 = 3;
316         index3,2 = index1,2;
317         index4,1 = 3;
318         index4,2 = index2,2;
319         Shape = 0;
```
end  
else  %dvs  index_1 = ObsSize-1
  a = ObsSize;
  b = ObsSize - 2;
  dist = [(X(a,1)-x).^2 (Y(a,1)-y).^2 (Z(a,1)-z).^2 ... 
         (X(b,index1_2)-x).^2 (Y(b,index1_2)-y).^2 (Z(b,index1_2)-z).^2 ... 
         (X(b,index2_2)-x).^2 (Y(b,index2_2)-y).^2 (Z(b,index2_2)-z).^2 ];
  [M, I] = min(dist);
  if I == 1
    index3_1 = a;
    index3_2 = 1;
  else
    index3_1 = b;
    index3_2 = index1_2;
    index4_1 = b;
    index4_2 = index2_2;
    Shape = 0;
  end
end
if Shape == 1
  MinDist = [index1_1, index1_2, index2_1, index2_2, ...
             index3_1, index3_2];
  if index1_1 == 1 || index1_1 == 2
    Shape = 3;
    else
      Shape = 2;
  end
else
  MinDist = [index1_1, index1_2, index2_1, index2_2, ...
             index3_1, index3_2, index4_1, index4_2];
end
%******************************************************************************************************************
%***** Sorts the points to be counterclockwise*****
if Shape == 0
  %square
  I1 = index1_1 - 1;
  I2 = index1_1 + 1;
  I3 = index1_2 - 1;
  I4 = index1_2 + 1;
  if I3 == 0
    I3 = ObsSize-1;
  end
  if I4 == ObsSize + 1
    I4 = 2;
end
k = 1:3
if MinDist(2+k+1) == I1 && MinDist(2+k+2) == I3
  MinDist(3:8) = [index1_1, I3, I1, I3, I1, index1_2];
elseif MinDist(2+k+1) == I1 && MinDist(2+k+2) == I4
  MinDist(3:8) = [I1, index1_2, I1, I4, index1_1, I4];
elseif MinDist(2+k+1) == I2 && MinDist(2+k+2) == I3
  MinDist(3:8) = [I2, index1_2, I2, I3, index1_1, I3];
elseif MinDist(2+k+1) == I2 && MinDist(2+k+2) == I4
  MinDist(3:8) = [index1_2, I2, I3, index1_1, I3];
APPENDIX A. CODE

MinDist(3:8) = [index1_1, I4, I2, I4, I2, index1_2];
end

elseif Shape == 2 %triangle, top
  if index1_1 == ObsSize
    if index2_2 == 2 && index3_2 == ObsSize
      MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
    elseif index3_2 < index3_2
      MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
    else
      MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
    end
  elseif index2_2 == ObsSize-1 && index3_2 == 1
    MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
  elseif index2_2 == 2 && index3_2 == 2
    MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
  elseif index2_2 < index3_2
    MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
  else
    MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
  end
end

elseif index2_1 == ObsSize
  if index3_2 == ObsSize-1 && index1_2 == 1
    MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
  elseif index1_2 == 2 && index3_2 == 2
    MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
  elseif index2_2 < index3_2
    MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
  else
    MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
  end
end

else
  %index3_1 == ObsSize
  if index2_2 == ObsSize-1 && index1_2 == 1
    MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
  elseif index1_2 == 2 && index3_2 == 2
    MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
  elseif index2_2 < index3_2
    MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
  else
    MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
  end
end

elseif Shape == 3 %triangle, bottom
  if index1_1 == 1
    if index3_2 == ObsSize-1 && index2_2 == 1
      MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
    elseif index3_2 == ObsSize && index2_2 == 2
      MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
    elseif index2_2 < index3_2
      MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
    else
      MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
    end
  elseif index3_2 == ObsSize && index2_2 == 2
    MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
  elseif index2_2 == ObsSize-1 && index3_2 == 1
    MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
  elseif index2_2 == 2 && index3_2 == 2
    MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
  elseif index2_2 < index3_2
    MinDist(3:6) = [index3_1, index3_2, index2_1, index2_2];
  else
    MinDist(3:6) = [index2_1, index2_2, index3_1, index3_2];
  end
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.

```matlab
function Neighbours = GetNeighbours(index1_1, index1_2, ...
index2_1, index2_2, ...
index3_1, index3_2, ...
ObsSize, varargin)

N = size(varargin);
N = N(2);

if N == 2
    index4_1 = varargin{1};
    index4_2 = varargin{2};
end
Shape = 4;
else
    Shape = 3;
end

if Shape == 4
    CN = zeros(8,8);
    % sorts the grid points
    if index4_1 > index1_1
        Index1_1 = index4_1;
    end
end
```
Index1_2 = index4_2;
Index2_1 = index1_1;
Index2_2 = index1_2;
Index3_1 = index2_1;
Index3_2 = index2_2;
Index4_1 = index3_1;
Index4_2 = index3_2;

elsif index3_1 > index4_1
Index1_1 = index3_1;
Index1_2 = index3_2;
Index2_1 = index4_1;
Index2_2 = index4_2;
Index3_1 = index1_1;
Index3_2 = index1_2;
Index4_1 = index2_1;
Index4_2 = index2_2;

elsif index2_1 > index3_1
Index1_1 = index2_1;
Index1_2 = index2_2;
Index2_1 = index3_1;
Index2_2 = index3_2;
Index3_1 = index4_1;
Index3_2 = index4_2;
Index4_1 = index1_1;
Index4_2 = index1_2;
else
Index1_1 = index1_1;
Index1_2 = index1_2;
Index2_1 = index2_1;
Index2_2 = index2_2;
Index3_1 = index3_1;
Index3_2 = index3_2;
Index4_1 = index4_1;
Index4_2 = index4_2;
end

I1 = Index1_2 - 1;
if I1 == 0
I1 = ObsSize - 1;
end

CN(:, :) = [Index1_1 Index1_2 Index1_1 I1 Index2_1 I1 Index2_1];
I2 = Index2_1 - 1;
I3 = Index3_2 + 1;
if I3 == ObsSize + 1
I3 = 2;
end

if I2 == 1
CN(2, :) = [Index2_1 Index2_2 Index2_1 I1 1 0 0];
CN(3, :) = [Index2_1 Index2_2 1 1 Index3_1 Index3_2 0 0];
CN(4, :) = [Index3_1 Index3_2 1 1 Index3_1 I3 0 0];
else
CN(2, :) = [Index2_1 Index2_2 Index2_1 I1 I2 I2 Index2_2];
CN(3, :) = [Index2_1 Index2_2 I2 Index2_2 I2 Index3_2 Index3_1 Index3_2];
CN(4, :) = [Index3_1 Index3_2 I2 Index3_2 I2 I3 Index3_1 I3];
end

CN(5, :) = [Index4_1 Index4_2 Index3_1 Index3_2 Index3_1 I3 Index4_1 I3];
I4 = Index4_1 + 1;
if I4 == ObsSize
CN(6, :) = [Index4_1 Index4_2 Index4_1 I3 ObsSize 1 0 0];
CN(7, :) = [Index4_1 Index4_2 ObsSize 1 Index1_1 Index1_2 0 0];
CN(8, :) = [Index1_1 Index1_2 ObsSize 1 Index1_1 I1 0 0];
else
    CN(6,:) = [Index4_1 Index4_2 Index4_1 I3 I4 I3 I4 Index4_2];
    CN(7,:) = [Index4_1 Index4_2 I4 Index4_2 I4 Index1_2 Index1_1 Index1_2];
    CN(8,:) = [Index1_1 Index1_2 I4 Index1_2 I4 I1 Index1_1 I1];
end
else % triangle
    R = 0;
    if index1_1 == 1
        R = 1;
        Index1_1 = index1_1;
        Index1_2 = index1_2;
        Index2_1 = index2_1;
        Index2_2 = index2_2;
        Index3_1 = index3_1;
        Index3_2 = index3_2;
    elseif index2_1 == 1
        R = 1;
        Index1_1 = index2_1;
        Index1_2 = index2_2;
        Index2_1 = index3_1;
        Index2_2 = index3_2;
        Index3_1 = index1_1;
        Index3_2 = index1_2;
    elseif index3_1 == 1
        R = 1;
        Index1_1 = index3_1;
        Index1_2 = index3_2;
        Index2_1 = index1_1;
        Index2_2 = index1_2;
        Index3_1 = index2_1;
        Index3_2 = index2_2;
    elseif index2_1 == ObsSize
        Index1_1 = index2_1;
        Index1_2 = index2_2;
        Index2_1 = index3_1;
        Index2_2 = index3_2;
        Index3_1 = index1_1;
        Index3_2 = index1_2;
    elseif index3_1 == ObsSize
        Index1_1 = index3_1;
        Index1_2 = index3_2;
        Index2_1 = index1_1;
        Index2_2 = index1_2;
        Index3_1 = index2_1;
        Index3_2 = index2_2;
    else
        Index1_1 = index1_1;
        Index1_2 = index1_2;
        Index2_1 = index2_1;
        Index2_2 = index2_2;
        Index3_1 = index3_1;
        Index3_2 = index3_2;
    end
if R == 1
    if bottom
        for k = 1:ObsSize-1
            CN(k,:) = [1 1 2 k+1 2 k 0 0];
        end
        I1 = Index2_2 + 1;
        I2 = Index3_2 - 1;
        if I1 == ObsSize + 1
            I1 = 2;
APPENDIX A. CODE

```matlab
end
if I2 == 0;
I2 = ObsSize - 1;
end
CN(ObsSize,:) = [2 Index2 2 I1 3 I1 3 Index2 2];
CN(ObsSize+1,:) = [2 Index2 3 Index2 2 3 Index3 2 2 Index3 2];
else
for k = 1:ObsSize-1
CN(k,:) = [ObsSize 1 ObsSize-1 k ObsSize-1 k+1 0 0];
end
I1 = Index2 - 1;
I2 = Index3 + 1;
if I1 == 0
I1 = ObsSize - 1;
end
if I2 == ObsSize + 1;
I2 = 2;
end
P1 = ObsSize - 1;
P2 = ObsSize - 2;
CN(ObsSize,:) = [P1 Index2 2 P1 I1 P1 I2 Index2 2];
CN(ObsSize+1,:) = [P1 Index2 2 P2 Index2 2 P2 Index3 2 P1 Index3 2];
CN(ObsSize+2,:) = [P1 Index3 2 P2 Index3 2 P2 I2 P1 I2];
end
Neighbours = CN;
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.

```
normal = − normal;
end
v_new = v − 2 * dot(v, normal) * normal;

\[
v_new = \frac{v_new \cdot \text{norm}(v)}{\text{norm}(v_new)}
\]

\[
\text{NewPos} = [\text{pos_int} + (dt-t) \cdot v_new \ldots
\]
\[
vnew dt-t];
\]
else
    NewPos = 0;
end

%triangle
if \text{dot}\left(\text{cross}(GP2-GP1, \text{pos_int}-GP1) \cdot \text{normal}\right) >= 0 \&\& \ldots \%do collide
\text{dot}\left(\text{cross}(GP3-GP2, \text{pos_int}-GP2) \cdot \text{normal}\right) >= 0 \&\& \ldots \%
\text{dot}\left(\text{cross}(GP1-GP3, \text{pos_int}-GP3) \cdot \text{normal}\right) >= 0
\text{v_new} = v - 2 \cdot \text{dot}(v, \text{normal}) \cdot \text{normal};

\[
\text{NewPos} = [\text{pos_int} + (dt-t) \cdot v_new \ldots
\]
\[
\text{v_new dt-t}];
\]
else
    NewPos = 0;
end

end

 else
    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 scripts 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 \times 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]

%************************************************
11 dr = \sqrt{6 \times D \times dt}; %Stepsize [\mu m]
12 N_G = \text{length}(G_x); %Number of different gradients.
13 N_time = \text{length}(\text{diff_times}); %Number of different diffusion times.
14 N_time_N_G = N_time \times N_G; %Total number of resulting phases/file.
15
16 limit = \text{round}(\text{diff_times} / dt); %Finding the step numbers
17 for i = 1:N_time %associated with the different
18 if limit(i) / 2 \neq \text{round}(limit(i) / 2) \%diffusion times.
19 limit(i) = limit(i) + 1; %And forces it to be even.
20 end
21 end
22
23 K = \text{max}(limit); %Total number of steps.
24 phase_1 = \text{zeros}(N, N_{time}, N_G); %Initialize arrays
25
Finding initial positions for all particles

```matlab
% Initial positions for all particles

cos_theta = 2*rand(1,N)-1;
theta = acos(cos_theta);
phi = 2*pi*rand(1,N);
r0 = rand(1,N)*R;
x0 = r0.*sin(theta).*cos(phi);
y0 = r0.*sin(theta).*sin(phi);
z0 = r0.*cos_theta;

string1 = ['Number of steps : ', num2str(K)];
disp(string1)
Crash_nr = 0; % Keeps track of number of collisions.

% Loop over all the particles
parfor n = 1:N
    if n/1000 == round(n/1000)
        string1 = ['Particle Nr : ', num2str(n), ' of ', num2str(N)];
        disp(string1)
    end

    x_pos = zeros(1,K); % Initializing arrays to save all positions.
    y_pos = x_pos;
    z_pos = x_pos;

    x_pos(1) = x0(n); % Adds the initial position.
    y_pos(1) = y0(n);
    z_pos(1) = z0(n);

% Finds position 2-K
k = 2; while k <= K
    if dK > 0 % Enters to find dK new positions.
        if dK > K-k+1 % Hinders the program to run
            dK = K-k+1;
        end
        if dK == 0 % Past total diffusion time.
            k = K + 1;
        end
    end

    d_cos_theta = 2*rand(1, dK) - 1; % Computes random
    d_theta = acos(d_cos_theta); % Directions.
    d_phi = 2*pi*rand(1, dK);
    dx_vec = d*x*sin(d_theta).*cos(d_phi); % Finds the
    dy_vec = d*x*sin(d_theta).*sin(d_phi);
    dz_vec = d*z*cos_theta;
```
\[
\begin{align*}
    x_{\text{pos}}(k:k+dK-1) &= \text{cumsum}(dx_{\text{vec}}) + x_{\text{pos}}(k-1); \quad \% \text{Finds and} \\
    y_{\text{pos}}(k:k+dK-1) &= \text{cumsum}(dy_{\text{vec}}) + y_{\text{pos}}(k-1); \quad \% \text{Saves the new} \\
    z_{\text{pos}}(k:k+dK-1) &= \text{cumsum}(dz_{\text{vec}}) + z_{\text{pos}}(k-1); \quad \% \text{Positions.}
\end{align*}
\]

\[k = k + dK;\]

\textbf{else} \quad \% \text{Enters when next step may give collision.}

\[
\begin{align*}
    pos &= [x_{\text{pos}}(k-1) \ y_{\text{pos}}(k-1) \ z_{\text{pos}}(k-1)]; \quad \% \text{Current position.} \\
    d_{\cos\theta} &= 2*\text{rand} - 1; \quad \% \text{Finds direction.} \\
    d_{\phi} &= 2*\pi*\text{rand}; \quad \% \\
    dx &= dr*sin(d_{\theta}).cos(d_{\phi}); \quad \% \text{Computes} \\
    dy &= dr*sin(d_{\theta}).sin(d_{\phi}); \quad \% \text{The step.} \\
    dz &= dr*d_{\cos\theta}; \quad \% \\
    v &= [dx \ dy \ dz]/dt; \quad \% \text{Finds velocity.} \\
    
\end{align*}
\]

\[ddt = dt; \quad \% \text{Variable to get correct total stepsizes during collision.}
\]

\%(III)****** Loop in case of multiple collision **********
\%********** in same time step. **********
\%collide = 1;
\%while collide == 1
\%
\%***** Computes time until collision from current *****
\%***** position given the velocity v.  *****
\%\[\begin{align*}
    A &= \text{dot}(\text{pos},v); \\
    B &= \text{dot}(v,v); \\
    C &= \sqrt{A^2 - B*(\text{dot}(\text{pos},\text{pos}) - R^2)}; \\
    t_c &= (-A + C)/B; \\
    \text{if } t_c < 0 \\
    t_c &= (-A - C)/B; \\
\end{align*}\]
\%
\%
\%if t_c >= ddt || t_c < 0 \%The step do not
\%collide = 0; \%lead to collision. \\
\%end
\%
\%if isreal(t_c) == 0 \%Hope it never
\%disp('something is wrong: ') \%Enters this one. \\
\%k = K + 1; \\
\%end
\%
\%if collide == 1 \%Collision
\%Crash_nr = Crash_nr + 1;
\%
\%Surface normal in collision point.
\%
\%Computes new velocity.
\%
\%New position.
\%
\%Updates v.
APPENDIX A. CODE

ddt = ddt−t*c;  %Updates ddt in
%case of multiple
%collision in one
time step.

else  %Not collision
end

x_pos(k) = x_pos(k−1) + v(1)*ddt;  %Finds new
y_pos(k) = y_pos(k−1) + v(2)*ddt;  %positions.
z_pos(k) = z_pos(k−1) + v(3)*ddt;  

end

k = k + 1;

%(III)*****************************************************************************

%(II)*****************************************************************************

%(IV) Adding the gradient, and computes final phase *****
%**** for three different sequences, and all the
%**** different time steps and gradient strengths. *****

n_time = 1;
n_G = 1;

for Nr = 1:N_time*N_G
    Y = G.*x(n_G)*dt*gamma;

    if Nr == N_time*n_G
        n_time = 1;
        n_G = n_G + 1;
    end

end

%(IV)*****************************************************************************

%(I)*****************************************************************************

string1 = ['Number of collisions: ', num2str(Crash_nr)];
disp(string1)
signal = sum(exp(1i*phase));  %Computes resulting phase
Times = limit*dt;

signal_matrix = zeros(N_G+1,N_time+1);  %Creates the output matrixes,
signal_matrix(2:end,1) = G.*x;  %first column and first row
signal_matrix(1,2:end) = Times;  %specifies gradient strength

%**** Saves all phases into the matrix *****

A = 1;
B = N_time;
for i = 2:N_G+1
    signal_matrix(i,2:end) = signal(A:B);
end
A = A + N_time;
B = B + N_time;
end

signal = signal_matrix;

%**** time in ms, G in mT/ms****
save SignalPulsedGrad signal
Bibliography


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


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

---

¹Not published