MONTE CARLO SIMULATIONS OF MAGNETIC FIELD PERTURBERS IN 
ULTRA-HIGH FIELD MRI USING SUPERCOMPUTERS

A THESIS

Presented in Partial Fulfillment of the Requirements for the Degree of Bachelor of 
Science in Engineering with distinction in the Electrical and Computer Engineering 
and the University Honors Program

By

Jonathan A. Kopechek

*******

The Ohio State University

2006

BS Committee

Dr. Bradley D. Clymer

Dr. Petra Schmalbrock

Advisor’s Approval

Advisor, College of Engineering
ABSTRACT

Recent research suggests that abnormal iron concentrations in certain parts of the human brain may be linked to several neurological diseases, in particular Alzheimer’s disease. It is hoped that ultrahigh field MRI (background magnetic fields at or above 7 Tesla) can be used to determine the iron concentrations in various regions of the brain. Monte Carlo simulations provide theoretical data which can be compared with experimental MRI data to determine effective methods of measuring local iron concentrations in the brain.

The goal of the project was to develop and implement a set of computer codes to perform the Monte Carlo simulations. The code was written in C and MPI to allow parallel-processing on supercomputers, which is advantageous due to the large size of the data. In the Monte Carlo simulation thousands of protons move randomly throughout the model and as they move they acquire phase changes which are related to changes in the local magnetic field. The brain stores iron in a roughly spherical protein called ferritin. MRI cannot directly detect ferritin in the brain but it may be possible to determine the properties and concentration of ferritin with ultrahigh field MRI. The presence of ferritin will slightly perturb the magnetic field during an MRI scan and this affects the phases of protons as they move randomly throughout the brain. It is this effect that is modeled by the simulation, which returns the theoretical accumulated phase of each proton.
Another aspect is the construction of the computer model of ferritin in the brain. For our purposes we built a cube filled with spherical perturbers representing ferritin. We developed optimization methods to construct a larger and more accurate model. The code can be modified to model other brain parameters. For example, future versions can include branched cylindrical magnetic field perturbers to model blood vessels in the brain, which contain iron in deoxyhemoglobin. By tweaking the model and repeating the Monte Carlo simulations one can predict the phases acquired by the protons. This information can be compared to experimental ultrahigh field MRI data to determine a more accurate model. This could lead to a better understanding of the cause and viable treatment methods of neurological diseases such as Alzheimer’s disease.
To my mother and father, for every blessing they have given me
ACKNOWLEDGMENTS

Most importantly I want to thank Prof. Bradley Clymer for advising me through the course of the project. Despite his very busy schedule he always responded to my questions in a timely manner and provided input at every stage. Every time I met with him to discuss the research I learned something new. I would also like to thank Dr. Petra Schmalbrock for providing the idea and motivations for the project and serving on my committee.

I want to thank all of my friends and professors who assisted with the research. In particular I would like to thank Brent Woods who was extremely helpful in overcoming programming challenges and providing input on other research-related issues.

I also want to thank my grandparents for providing a quiet and comfortable location for me to work on the project. Finally I must thank my parents. Without their support I would never have had the opportunity to take on a project like this. They have blessed me in so many ways.
VITA

April 5, 1984 ......................... Born - Columbus, OH

June 2002 .......................... Graduated from high school, Gahanna, Ohio

June 11, 2006 (expected) ........ B.S.E.C.E., The Ohio State University, Columbus, Ohio
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>2</td>
</tr>
<tr>
<td>Dedication</td>
<td>4</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>5</td>
</tr>
<tr>
<td>Vita</td>
<td>6</td>
</tr>
<tr>
<td>List of Figures</td>
<td>9</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Statement of problem</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Previous Work</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Design Process</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Organization</td>
<td>5</td>
</tr>
<tr>
<td>2. Methods</td>
<td>6</td>
</tr>
<tr>
<td>2.1 The simulation</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Initialization of simulation</td>
<td>6</td>
</tr>
<tr>
<td>2.3 The Monte Carlo walk</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Calculation of relaxivity</td>
<td>7</td>
</tr>
<tr>
<td>2.5 The computer model</td>
<td>8</td>
</tr>
<tr>
<td>2.6 The spherical magnetic field perturber</td>
<td>9</td>
</tr>
</tbody>
</table>
3. Experimental Results and Limitations ................. 12
   3.1 Computational Limitations ....................... 12
   3.2 Uniform grid approach .......................... 12
   3.3 Average magnetic fields approach ............... 13
   3.4 Fast fourier transform approach ................. 15
   3.5 Variable spacing approach ....................... 17

4. Conclusions and Extensions ......................... 18

Appendices:

A. Monte Carlo walk simulation in C ................... 20
B. Monte Carlo Walk Simulation Program in MPI ......... 40
C. Model development code (MATLAB) .................. 62
D. Microcube development for uniform grid approach .... 70

Bibliography ........................................... 84
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Randomly-placed spherical perturbers within cube model</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Contour of magnetic field perturbation in z direction</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Contour of magnetic field perturbation in x/y direction</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Magnetic field perturbation v. distance from perturber origin</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Magnetic field perturbation pattern for a single perturber at x = 16nm, y = 16nm, z = 16nm</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Histogram of average magnetic fields for microcubes</td>
<td>14</td>
</tr>
<tr>
<td>3.2 Magnetic field perturbation pattern for three spherical perturbers using fast Fourier transform approach</td>
<td>16</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Alzheimer’s disease is a neurological condition that can lead to memory loss, cognitive loss and premature death [6]. According to the National Center for Health Statistics over 50,000 Americans died due to Alzheimer’s in 2002, 99 percent of whom were 65 or older, making it one of the leading causes of death in persons over 65 years of age [9]. It is estimated that currently 4.5 million Americans have the disease. Recent studies suggest that a correlation may exist between neurological diseases such as Alzheimer’s and abnormal concentrations of molecular iron in certain regions of the brain [4] [5]. This correlation is not well understood because no noninvasive in vivo method currently exists to accurately determine local brain iron concentrations. However, magnetic resonance imaging (MRI) at ultrahigh magnetic fields (at or above seven Tesla) may provide a viable noninvasive method for in vivo imaging of local iron concentrations in the brain.

1.1 Background

Magnetic resonance imaging is based upon physical principles associated with nuclear magnetic resonance (NMR). NMR relies on a fundamental atomic property known as spin. Each proton, electron, and neutron in the atom possesses a spin of
1/2. The net nuclear spin of an atom is the sum of spins from unpaired protons and neutrons. Due to the abundance of water in the human body MRI utilizes the spin of hydrogen atoms which consist of a single proton and thus has a net nuclear spin of 1/2. The spin of the proton is thought of as a magnetic moment vector. In the presence of an external magnetic field the magnetic moment vector will align itself in the direction of the field. The orientation of the proton poles are determined by energy states. Protons can absorb or release photons in order to transition between the energy states. In the body there are billions of hydrogen atoms transitioning between the energy states and the difference between the number in each state causes a net magnetization. The net magnetization vector points in the direction of the external magnetic field. This is chosen to be the z-direction in literature and in this thesis. The direction of the magnetization vector can be altered by applying an RF pulse at the Larmor frequency, which is given by \( \gamma \cdot B \) where \( \gamma \) is the gyromagnetic ratio (42.58 MHz/T for hydrogen). When the RF pulse is removed the magnetization vector will return to the original orientation by releasing photons. This produces an RF signal that can be detected by antenna. The time that it takes for the magnetization vector to return to the equilibrium state is known as the relaxation time. The relaxation rate is different for various tissues and it is this property that allows tissues to be identified with magnetic resonance imaging. This is why MRI is an important clinical application. [7]

The externally applied magnetic field is uniform, but the presence of molecular iron can alter or perturb the magnetic field close to the iron particles. These effects are generally insignificant in clinical MRI (1.5 T) but can be important at ultrahigh magnetic fields such as seven Tesla. This can be a problem in some applications
but for our research we attempt to utilize these effects to obtain more information about the physiological environment of the brain. These magnetic field perturbations affect the Larmor frequency of the protons and the protons acquire different phases. By observing these phase changes over a period of time one can estimate the average magnetic field experienced by the protons. Theoretically these data should allow us to determine the concentration of iron in specific regions of the brain by first identifying the magnetic field perturbations.

1.2 Statement of problem

A noninvasive method to identify local brain iron concentrations in vivo may be significant in the detection and treatment of neurological disorders. Research on this topic focuses on two approaches - obtaining experimental data from human subjects and determining expected values from theoretical data for comparison and analysis. The latter is the focus of this thesis. Large-scale simulations can provide useful information about the effects of abnormal brain iron concentrations on MRI signals. This thesis implements the Monte Carlo walk simulation on supercomputing clusters in order to allow for larger and faster simulations. The Message Passing Interface (MPI) at the Ohio Supercomputing Center (OSC) is utilized to provide parallel-processing functionality. One of the most important aspects of the simulation is the model development. Possible model implementations are analyzed. The computer codes developed and analysis performed by this thesis lay the framework for future research on this topic.
1.3 Previous Work

This thesis builds on previous work on this topic. Yablonskiy and Haacke developed an approach to describe the MRI signal in the presence of local magnetic field inhomogeneities. They examined field inhomogeneities and perturbation effects due to spherical contrast agents and cylindrical blood vessels containing deoxyhemoglobin, both of which contain iron [13] [14]. Ordidge and Bakker also examined these effects [1] [10]. This thesis implements the Monte Carlo walk simulation technique described by Boxerman, who analyzed the behavior and impact of contrast agents and produced theoretical results closely matching experimental data [3]. Weisskoff also implemented a similar Monte Carlo simulation procedure [12]. In related studies, Reichenbach described methods to identify physiological systems by analyzing local static magnetic field inhomogeneities [11] and Kennan discussed the effects of local field inhomogeneities as caused by contrast agents in the capillaries [8].

1.4 Design Process

The project consisted of two parts - (1) design and implementation of Monte Carlo simulation programs, and (2) design, analysis, and implementation of the computer model. In 2000 J.D. Wylie designed a Monte Carlo walk simulation for a thesis, however his program code was implemented entirely in MATLAB. The first few weeks were spent reviewing related journal articles and learning Wylie’s Monte Carlo walk code. The C programs for the Monte Carlo simulations were designed and written in the spring of 2005. Beginning in June 2005 the MPI code was written to implement parallel-processing and by the end of July 2005 code development and debugging of
the Monte Carlo walk simulations was completed and implemented on the super-
computing clusters. The next step was to design and build a computer model of a
voxel-sized region of the brain. This proved to be a difficult task and was the focus of
much of the project. It was quickly determined that the computational requirements
for a high-resolution voxel-sized model were beyond the scope of current technology.
Several approaches were discussed and tested. The first approach involved breaking
up the voxel into thousands of microcubes. It was thought that by determining the
average magnetic fields of the microcubes would be normally distributed. Histograms
failed a goodness-of-fit test for a Gaussian distribution. It was determined that the
cause of this was truncation error due to the microcube size. However computational
constraints became an issue. To overcome this a new method was devised which uti-
lized the Fourier transform. This technique showed promise but was also limited by
memory constraints. A third approach was analyzed in which the grid spacing of the
magnetic field perturbation map was varied based on distance from perturbers. It
was shown that this approach is theoretically feasible with current supercomputing
technology. Further work on this topic is needed to implement this technique.

1.5 Organization

The following chapter will present the methods implemented in the thesis, includ-
ing the Monte Carlo simulation program and model development. Chapter III dis-
cusses experimental results, limitations and approaches to overcome the limitations.
Chapter IV discusses opportunities for further research on this topic and summarizes
the thesis.
CHAPTER 2

METHODS

2.1 The simulation

The Monte Carlo walk simulation provides theoretical $T_2$ calculations for the model given. The process is described in the following sections.

2.2 Initialization of simulation

The first step in the simulation is to randomly place several thousand protons throughout the model. For simplicity we chose to use a uniform distribution but any distribution or even a probability map could also be implemented. Any protons which are found to be inside a perturber are repositioned until no protons are within perturbers.

2.3 The Monte Carlo walk

After all of the protons have been placed in the model the Monte Carlo walk can begin. A number of parameters are required and we chose the values used by Boxerman [3]. The program iterates through a sequence of simulated time steps until the echo time, $TE$, is reached. The time step length, $\Delta t$, was 20 milliseconds and the echo time was 100 milliseconds, thus there were a total of 50 time steps. At each time
step the protons move according to diffusion principles in random directions, and the accumulated phase for each proton is calculated. The diffusion step length is given by \( \sqrt{2D\Delta t} \), where \( D \) is the diffusion coefficient. We chose \( D \) to be the diffusion coefficient of water, 1.3 \( \mu m^2/ms \) at body temperature. Brain tissue is mostly water so this is a reasonable approximation. As the protons move throughout the model, the magnetic fields that they experience gradually changes due to perturbations caused by the presence of molecular iron. For simplicity we take the average of the magnetic field experienced by each proton during each step in the random walk. This is performed by summing the magnetic field at the beginning and end of each step and dividing by two. The accumulated phase (in radians) of each proton can then be calculated by Equation 2.1 [3].

\[
\phi = \gamma B_{avg} \Delta t
\]  

(2.1)

where \( \gamma \), the gyromagnetic ratio, is equal to 267510 rad/ms/Tesla for hydrogen protons.

### 2.4 Calculation of relaxivity

At the end of the Monte Carlo walk we process the accumulated phase data. For each proton the phase accumulations are summed. Then the real parts and imaginary parts are summed separately for all protons. The signal is computed by taking the magnitude and dividing by the number of protons. The \( T_2 \) relaxivity is calculated as shown in Equation 2.2 [3].

\[
T_2 = \frac{-TE}{\ln(\text{signal})}
\]  

(2.2)
2.5 The computer model

Developing the computer model may be the most significant aspect of the project. We construct a cube and fill it with randomly-placed perturbers. We use a uniform distribution but any distribution or even a probability map could be used to randomly place the perturbers within the model. After placing all of the perturbers within the model we calculate the magnetic field at uniform intervals within the model. This is done by computing the magnetic field perturbation due to each perturber and summing by superposition. The perturber concentration was also variable, however we kept it below two percent so that the superposition principle would still hold.

Figure 2.1: Randomly-placed spherical perturbers within cube model
2.6 The spherical magnetic field perturber

Perturbers of any geometry may be implemented in the Monte Carlo simulation. We chose spherical perturbers for two reasons: (1) A closed-form solution exists for the magnetic field perturbation pattern of a sphere, and (2) The protein ferritin, which contains much of the molecular iron found in the human brain, has a spherical shape. Equation 2.3 gives the magnetic field perturbations at any point throughout the model [2].

\[
\mathbf{B} \approx B_0 \sum_{k=1}^{n} \left( 1 + \frac{\mu_e - 1}{3} + \frac{(\mu_e - \mu_i)R^3((x - x_k)^2 + (y - y_k)^2 - 2(z - z_k)^2)}{3((x - x_k)^2 + (y - y_k)^2 + (z - z_k)^2)^{5/2}} \right) \quad (2.3)
\]

The perturbations are symmetrical in the x and y directions but doubled and opposite in sign in the z direction, where the z is the direction of the static magnetic field. Figures 2.2 and 2.3 show the magnetic field perturbation patterns due to a sphere as a contour plot. Figure 2.5 shows the 3-D representation of the magnetic field perturbations (at the midpoint on the x-axis between the origin and cube edge).
Figure 2.2: Contour of magnetic field perturbation in z direction

Figure 2.3: Contour of magnetic field perturbation in x/y direction
Figure 2.4: Magnetic field perturbation v. distance from perturber origin

Figure 2.5: Magnetic field perturbation pattern for a single perturber at x = 16nm, y = 16nm, z = 16nm
CHAPTER 3

EXPERIMENTAL RESULTS AND LIMITATIONS

3.1 Computational Limitations

Model development is critical in obtaining simulation data that closely matches experimental results from human subjects. The size of the model is an important factor in simulation accuracy. The goal was to construct a model comparable in size to a voxel in an MRI scan. A typical voxel is 50 μm x 50 μm x 1 mm. The ferritin particles are 10-12 nanometers in diameter so high resolution is needed to detect the magnetic field perturbations. The original approach was to build a cubic grid with a resolution of one nanometer but this was quickly abandoned. With the given voxel size this method would have required 2.5x10^{15} sample points. To obtain the precision required for the magnetic field perturbations a double data format must be used. Consequently this method requires 5 petabytes of temporary memory (1 petabyte = 1,000,000 gigabytes). This is clearly beyond current computational capabilities. In response several approaches were hypothesized and tested.

3.2 Uniform grid approach

The most obvious alternative is to increase the spacing between the samples, effectively reducing the resolution. The minimum resolution was chosen to be the diffusion
step length which was 228 nanometers for our study and to keep the lengths as powers of two the minimum resolution was chosen to be 128 nanometers. For the given voxel size and a resolution of 128 nanometers $1.2 \times 10^9$ points are needed to construct one voxel. Constructing the magnetic field perturbation map of a model of this size on a computer requires 2 gigabytes of temporary memory. This is currently beyond the capabilities of many desktop computers but most supercomputing clusters, including several at the Ohio Supercomputing Center, have temporary memory capabilities of 32, 64, and even 128 gigabytes. In addition to the memory requirements, computational time is also significant. Experimental results showed that at this resolution one voxel could theoretically be constructed in about 82 hours on one processor. Utilizing parallel-computing to divide up the tasks of building a voxel would reduce the required execution time linearly. The low resolution required for this technique poses a problem however. The magnetic field perturbation drops off rapidly away from the perturber surface. Perturbers could be located up to 90 nanometers from the nearest sample point. At this distance perturbation effects are minimal. A simulation of this model would not accurately account for the concentration of magnetic field perturbers which could lead to erroneous results.

### 3.3 Average magnetic fields approach

It was clear that the process of calculating the magnetic field at each proton and summing by superposition was the primary limitation in the model development. A new method was sought to estimate the magnetic field values and reduce the number of computations required for each sample point. This approach involved breaking up the voxel into thousands of microcubes and measuring the average magnetic field
inside each. It was hoped that the distribution of average magnetic field data from the microcubes would be a normal distribution. If this was the case the average magnetic fields could be generated using a Gaussian random number generator. As a test, thousands of microcubes were constructed and the average magnetic fields measured. These data were plotted to histograms. At first glance the histograms appeared to show a Gaussian distribution but further inspection showed a number of unexpected small spikes away from the center. Goodness-of-fit tests for the Gaussian distribution failed. Further testing showed that these spikes were due to insufficient microcube size. Figure 3.1 gives the histogram. Increasing the microcube size reduces resolution leading to inaccurate results so another method was sought.

![Histogram of average magnetic fields for microcubes](image.png)

*Figure 3.1: Histogram of average magnetic fields for microcubes*
3.4 Fast fourier transform approach

A novel approach to the problem was developed and implemented using the Fourier transform. The problem of building a model can be handled in the frequency domain by considering the perturbers as impulses and convolving with the magnetic field pattern of a single perturber. Using the fast Fourier transform (FFT) method, the magnetic field perturbation map is transformed into the frequency domain by equation 3.1 and multiplied with the frequency domain representation of an impulse at a specific randomly-chosen location.

\[ F[n] = \sum_{k=0}^{N-1} f[k] e^{-2j\pi nk/N} \]  
(3.1)

\[ f[k] = \frac{1}{N} \sum_{n=0}^{N-1} F[k] e^{2j\pi nk/N} \]  
(3.2)

The resulting pattern is transformed back into the time domain with an inverse FFT shown in equation 3.2, providing the magnetic field perturbation map due to the perturbers. The process is described mathematically by equation 3.3 (time domain) and equation 3.4 (frequency domain).

\[ \Delta B(x - x_0, y - y_0, z - z_0) * \delta(x - x_0, y - y_0, z - z_0) \]  
(3.3)

\[ \mathcal{F}\mathcal{T}\{\Delta B(x, y, z)\} \cdot \mathcal{F}\mathcal{T}\{\delta(x, y, z)\} \]  
(3.4)

In the frequency domain this is simply multiplication. The Fourier transform of an impulse is given in equation 3.5 for the three-dimensional case.

\[ \mathcal{F}\mathcal{T}\{\delta(x - x_0, y - y_0, z - z_0)\} \Rightarrow e^{-j k_x x_0 - j k_y y_0 - j k_z z_0} \]  
(3.5)
where \( \Delta k_x = \frac{2\pi}{N(\Delta z)} \), \( \Delta k_y = \frac{2\pi}{N(\Delta y)} \), \( \Delta k_z = \frac{2\pi}{N(\Delta z)} \).

This method could potentially reduce the execution time required and allow for much better resolution, although the memory requirements would be increased because additional space must be reserved to store the real and imaginary parts of the data. Figure 3.2 shows the results from a demonstration in which three perturbers were placed within a cube and the magnetic field perturbation pattern was computed using the FFT method. The plot gives the perturbation pattern in y and z directions where x is halfway from the origin to the cube edge.

Figure 3.2: Magnetic field perturbation pattern for three spherical perturbers using fast Fourier transform approach
3.5 Variable spacing approach

Another approach that was given consideration involved varying the spacing between sample points based on the distance from the nearest perturber. Because the magnetic field perturbations are stronger closer to the surface of the perturber a higher resolution is required near the perturber. However, further away from the perturber the changes in field perturbations are much less and resolution can be reduced. Implementing this technique with interpolation methods could significantly reduce the number of computations and amount of memory required thus increasing both computational speed and avoiding temporary memory limitations. In addition the model and simulations would be much more accurate. This approach was not implemented in this thesis but may be an interesting topic for future work.
CHAPTER 4

CONCLUSIONS AND EXTENSIONS

To better understand the significance of brain iron in Alzheimer’s disease, new methods are needed to analyze in vivo ultra-high field MRI data. Monte Carlo simulations of computer models may provide important information on this topic. Previous simulations and models were restricted in size by computational capabilities. In this thesis the simulation was implemented in C and MPI so that it could be broken into pieces and distributed to multiple supercomputing clusters for processing. This reduces simulation time and makes it possible to run larger simulations which can lead to greater accuracy. One possible model was designed and implemented to analyze the effects of ferritin on the magnetic resonance relaxivity rate. In this thesis the model consisted of randomly-placed spheres representing ferritin. However future studies can implement new models with infinite cylinders to represent deoxyhemoglobin in capillaries. Model development was limited by computational limitations and better algorithms for building the model were needed to meet memory and speed requirements. Several methods were proposed and tested. The uniform grid approach is the most straightforward but is only feasible with large sample point spacings which greatly reduces resolution and accuracy. The fast Fourier transform technique showed promise but requires additional development and may be constrained by temporary
memory limitations in some cases. The variable spacing approach may reduce the computational speed and temporary memory requirements by reducing the number of sample points used. If an effective interpolation technique is chosen the accuracy may not be compromised while maintaining the computational savings. The idea is that more sample points would be located near the perturbers where the magnetic field changes more rapidly, while less sample points would be located further from the perturbers where the magnetic field is relatively constant. The Monte Carlo walk simulation has been a successful method for predicting the MRI relaxivity from a computer model of a physiological system. The development and implementation of accurate models may lead to significant theoretical data about the impact of brain iron concentrations in Alzheimer’s disease.
APPENDIX A

MONTE CARLO WALK SIMULATION IN C

/*********************** MONTE CARLO WALK SIMULATION ***********************/
/* Code written by Jonathan Kopechek, The Ohio State University */
/* */
/* This C program performs a Monte Carlo simulation to find the T2' value of */
/* the model. The following parameters may be edited to modify the simulation: */
/* */
/* */
/* 1. STATICFIELD */
/* 2. MEAN PERTURBER RADIUS */
/* 3. MRI SEQUENCE TYPE */
/* 4. MODEL_LENGTH */
/* 5. MAGNETIC SUSCEPTIBILITY OUTSIDE PERTURBERS */
/* 6. MAGNETIC SUSCEPTIBILITY INSIDE PERTURBERS */
/* 7. STEPLENGTH */
/* 8. SEQTYPE */
*/ 9. TE */

/* 10. NUM_PROTONS */

/* */

*******************************************************************************/

****** EDIT THE FOLLOWING PARAMETERS TO MODIFY THE MODEL *******/
#define STATICFIELD 7 // Background magnetic field (Tesla)
#define MEAN_RADIUS 5 // Average radius of magnetic field perturber (nm)
#define MODEL_LENGTH 2048 // Length of model (nanometers)
#define CONCENTRATION 0.00001 // Perturber concentration
#define MU_OUTSIDE 1.000000 // Magnetic susceptibility outside perturber
#define MU_INSIDE 1.000006 // Magnetic susceptibility inside perturber
#define STEPLENGTH 128 // Gridsize (nanometers)

/* Define constants for Monte Carlo simulation */
#define SEQTYPE 0 // Sequence type (1 for spin-echo, 0 for gradient echo)
#define TE 100 // Echo time (milliseconds)
#define NUM_PROTONS 500 // Number of protons to simulate
#define D 1300000 // Diffusion coefficient (nanometers^2/milliseconds)
#define dt 0.02 // Step time (seconds)
#define RESOLUTION 1 // Resolution for diffusion length

*******************************************************************************/

********** DO NOT EDIT BELOW THIS LINE ***********/
/ * Include files */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

/* Declare program arrays */
double fieldarray[MODEL_LENGTH/STEPLength][MODEL_LENGTH/STEPLength][MODEL_LENGTH/STEPLength];
int perturbarray[MODEL_LENGTH/STEPLength][MODEL_LENGTH/STEPLength][MODEL_LENGTH/STEPLength];
double startloc[3][NUM_PROTONS];
double phasearray[NUM_PROTONS][5001];
double cumulativephase[NUM_PROTONS];
double cosarray[NUM_PROTONS];
double sinarray[NUM_PROTONS];

/* Initialize variables */
double standard_dev = 1; // Standard deviation of perturber radii
double pi = 3.14159;
double phase, cos_sum, sin_sum;

/******* THIS FUNCTION PICKS THE STARTING LOCATION FOR EACH PROTON *******/
int pickstart(int offset,double mean_rad) {
    int repick = ceil(NUM_PROTONS/20);
    double xstart[repick];
    double ystart[repick];
    double zstart[repick];

    /* Clear variables */
    double xnew = 0;
    double ynew = 0;
    double znew = 0;
    int k, m, x0, y0, z0;

    /* Randomly choose proton starting locations */
    for (k=0; k<repick; k++) {
        xstart[k] = drand48()*MODEL_LENGTH;
        ystart[k] = drand48()*MODEL_LENGTH;
        zstart[k] = drand48()*MODEL_LENGTH;
    }

    /* Repick starting locations if proton is located within perturber */
    for (m=0; m<repick; m++) {
        x0 = floor(xstart[m]/STELPLENGTH);
        y0 = floor(ystart[m]/STELPLENGTH);
z0 = floor(zstart[m]/STEPLength);

while (perturbarray[x0][y0][z0] == 1) {

x0 = floor(xstart[m]/STEPLength);
y0 = floor(ystart[m]/STEPLength);
z0 = floor(zstart[m]/STEPLength);

    // Randomly select next proton position
    double xg = drand48() - 0.5;
double yg = drand48() - 0.5;
double zg = drand48() - 0.5;

    xnew = xstart[m] + ( xg * ((sqrt(2 * D * dt))) * RESOLUTION);
ynew = ystart[m] + ( yg * ((sqrt(2 * D * dt))) * RESOLUTION);
znew = zstart[m] + ( zg * ((sqrt(2 * D * dt))) * RESOLUTION);
    xstart[m] = xnew; ystart[m] = ynew; zstart[m] = znew;
}

    // Calculate adjusted proton starting locations
    startloc[1][m] = xstart[m] * RESOLUTION + offset;
    startloc[2][m] = ystart[m] * RESOLUTION + offset;
    startloc[3][m] = zstart[m] * RESOLUTION + offset;

24
return 0;
}

/****************** THIS FUNCTION PERFORMS THE MONTE CARLO WALK ******************/
int walksim(int offset) {

    /* Initialize variables */
    int err, indicator, j, k;
    double xnew, ynew, znew, xold, yold, zold;

    /* Step proton through model */
    for (j=1; j<NUM_PROTONS; j++) {
        int index = ceil(j/20);
        xnew = startloc[1][index];
        ynew = startloc[2][index];
        znew = startloc[3][index];

        int accumphase = 0;
        for (k=1; k<ceil(TE / (2 * dt)); k++) {

            err = 1;

            }
indicator = 1;
while (err == 1) {

  xold = xnew;
yold = ynew;
zold = znew;

  // Randomly select next proton position
  double xg = drand48() - 0.5;
  double yg = drand48() - 0.5;
  double zg = drand48() - 0.5;

  xnew = xold + ( xg * ((sqrt(2 * D * dt))) * RESOLUTION);
ynew = yold + ( yg * ((sqrt(2 * D * dt))) * RESOLUTION);
znew = zold + ( zg * ((sqrt(2 * D * dt))) * RESOLUTION);

  // Perform data check
  if (floor(xnew) >= MODEL_LENGTH) { xnew = (MODEL_LENGTH)-1; err = 0; }
  if (floor(ynew) >= MODEL_LENGTH) { ynew = (MODEL_LENGTH)-1; err = 0; }
  if (floor(znew) >= MODEL_LENGTH) { znew = (MODEL_LENGTH)-1; err = 0; }
  if (xnew < 0) { xnew = 0; err = 0; }
  if (ynew < 0) { ynew = 0; err = 0; }
  if (znew < 0) { znew = 0; err = 0; }

}
int xloc = floor(xnew/STEPLength);
int yloc = floor(ynew/STEPLength);
int zloc = floor(znew/STEPLength);

err = perturbarray[xloc][yloc][zloc];

}

// Calculate new phase of proton
evalphase(xnew,ynew,znew,xold,yold,zold,indicator,offset);
phasearray[j][k] = phase;

xold = xnew;
yold = ynew;
zold = znew;

}

// Repeat walk, flip phase for spin-echo
for (k=(ceil(TE / (2 * dt))+1); k<ceil(TE / dt); k++) {

err = 1;
if (SEQTYPE == 1) { indicator=-1; }
else { indicator=1; }
while (err == 1) {

xold = xnew;
yold = ynew;
zold = znew;

// Randomly select next proton position
double xg = drand48() - 0.5;
double yg = drand48() - 0.5;
double zg = drand48() - 0.5;

xnew = xold + ( xg * ((sqrt(2 * D * dt))) * RESOLUTION);
ynew = yold + ( yg * ((sqrt(2 * D * dt))) * RESOLUTION);
znew = zold + ( zg * ((sqrt(2 * D * dt))) * RESOLUTION);

// Perform data check
if (floor(xnew) >= MODEL_LENGTH) { xnew = (MODEL_LENGTH)-1; err = 0; }
if (floor(ynew) >= MODEL_LENGTH) { ynew = (MODEL_LENGTH)-1; err = 0; }
if (floor(znew) >= MODEL_LENGTH) { znew = (MODEL_LENGTH)-1; err = 0; }
if (xnew < 0) { xnew = 0; err = 0; }
if (ynew < 0) { ynew = 0; err = 0; }
if (znew < 0) { znew = 0; err = 0; }

int xloc = floor(xnew/STEPLENGTH);
int yloc = floor(ynew/STEPLENGTH);
int zloc = floor(znew/STEPLENGTH);

err = perturbarray[xloc][yloc][zloc];

}

// Calculate new phase of proton

evalphase(xnew,ynew,znew,xold,yold,zold,indicator,offset);

phasearray[j][k] = phase;

xold = xnew;
yold = ynew;
zold = znew;
}

}

return 0;
}

********** THIS FUNCTION CALCULATES THE PROTON'S ACQUIRED PHASE **********/
int evalphase(double xm,double yn,double zm,double xo,double yo,double zo,int indicator, int offset)
{
    int xnew = abs(floor(xn/STEPLength));
    int ynew = abs(floor(yn/STEPLength));
    int znew = abs(floor(zn/STEPLength));
    int xold = abs(floor(xo/STEPLength));
    int yold = abs(floor(yo/STEPLength));
    int zold = abs(floor(zo/STEPLength));
    double gamma = 267510;
    double averageB = (fieldarray[xnew][ynew][znew] + fieldarray[xold][yold][zold]) / 2;
    phase = gamma * averageB * dt * indicator;
    return 0;
}

int builduniverse(double mean_rad)
{
    */ Declare variables */
    double accumulator;
    double accumulatorold;
    double temporary;
    double diameter;
    double P;
    double denominator;
int sphere;
int x,y,z,n,m;

/* Initialize variables */
double Bfield = STATICFIELD;
double sphere_radius = mean_rad;

/* Initialize settings */
int arraysize = MODEL_LENGTH;
double arraysize_float = MODEL_LENGTH;
int step_length = STEPLENGTH;
int numspHERES = ceil(arraysize_float*arraysize_float*arraysize_float*
CONCENTRATION/(4*pi*pwr(f(sphere_radius,3)/3));

/* Declare arrays */
int xpos[numspHERES];
int ypos[numspHERES];
int zpos[numspHERES];

/* Randomly choose locations of magnetic field perturbers */
srand( (unsigned)time( NULL ));
for (m=0; m<numspHERES; m++) {
    xpos[m] = rand()%arraysize;
    ypos[m] = rand()%arraysize;
    zpos[m] = rand()%arraysize;
}
ypos[m] = rand()%arraysize;
zpos[m] = rand()%arraysize;

/* Create array of uniformly spaced intervals in universe and specify locations of perturbers */
for (sphere=0; sphere<numspheres; sphere++) {
    for (x=xpos[sphere]-ceil(sphere_radius); x < (xpos[sphere]+ceil(sphere_radius)); x++) {
        for (y=ypos[sphere]-ceil(sphere_radius); y < (ypos[sphere]+ceil(sphere_radius)); y++) {
            for (z=zpos[sphere]-ceil(sphere_radius); z < (zpos[sphere]+ceil(sphere_radius)); z++) {
                if (((0 < x) & (arraysize > x) & (0 < y) & (arraysize > y) & (0 < z) & (arraysize > z)) {  
                    if (sqrt(pwr(x-xpos[sphere],2)+pwr(y-ypos[sphere],2)+pwr(z-zpos[sphere],2)) <= sphere_radius) {
                        int xr = floor(x/step_length);
                        int yr = floor(y/step_length);
                        int zr = floor(z/step_length);
Perturbarray[xr][yr][zr] = 1;  
                    }
                }
            }
        }
    }
}
} } }

/* Determine magnetic field values at each point in universe */
for (x=0; x<arraysize; x+=step_length) {
    for (y=0; y<arraysize; y+=step_length) {
        for (z=0; z<arraysize; z+=step_length) {
            int xr = floor(x/step_length);
            int yr = floor(y/step_length);
            int zr = floor(z/step_length);
            if (perturbarray[xr][yr][zr] == 1) {
                fieldarray[xr][yr][zr] = Bfield*(1+(MU_OUTSIDE-1)/3);
            }
            else if (perturbarray[xr][yr][zr] < 1) {
                accumulator = 0;
                for (sphere=0; sphere<numspheres; sphere++) {
                    accumulatorold = accumulator;
                    if (((pwr(x-xpos[sphere],2) + pwr(y-ypos[sphere],2) + pwr(z-zpos[sphere],2)) < 1) {
                        temporary = 0;
                    }
                    else if (((pwr(x-xpos[sphere],2) + pwr(y-ypos[sphere],2) + pwr(z-zpos[sphere],2)) > 0) {  
            

denominator = sqrt(pwr(x-xpos[sphere],2) + pwr(y-ypos[sphere],2) + pwr(z-zpos[sphere],2));

temporary = (((MU_OUTSIDE-1)/3)+(MU_OUTSIDE-MU_INSIDE)*(sphere_radius*
sphere_radius*sphere_radius)*((pwr(x-xpos[sphere],2) + pwr(y-ypos[sphere],2) - 2
* pwr(z-zpos[sphere],2))) / (3 * pwr(denominator,5))

accumulator = temporary + accumulatorold;

fieldarray[xr][yr][zr] = Bfield * (1 + accumulator);

return 0;

/******* THIS FUNCTION COMPUTES THE POWER OF AN INTEGER *******/

int pwr(value,power) {
    int i;
    int result = 1;
    for(i=0;i<power;i++) {
        result = result * value;
    }
    return result;
}
*/
/* THIS FUNCTION COMPUTES THE POWER OF A FLOAT */
int pwrfd(double value, int power) {
    int i;
    double result = 1;
    for (i = 0; i < power; i++) {
        result = result * value;
    }
    return result;
}

*/
/* THIS FUNCTION SUMS THE PHASES FOR EACH PROTON */
int sum_phase(void) {
    int i, j;
    for (i = 0; i < NUM_PROTONS; i++) {
        cumulativephase[i] = 0;
        for (j = 0; j < ceil(TE / dt); j++) {
            cumulativephase[i] = cumulativephase[i] + phasearray[i][j];
        }
    }
}


```c

/**
 * This function calculates the real part of the proton phases
 */

int get_cos_array(void) {
    int i;
    for(i=0;i<NUM_PROTONS;i++) {
        cosarray[i] = cos(cumulativephase[i]);
    }
    return 0;
}

/**
 * This function calculates the imaginary part of the proton phases
 */

int get_sin_array(void) {
    int i;
    for(i=0;i<NUM_PROTONS;i++) {
        ...
```


```c
sinarray[i] = sin(cumulativephase[i]);
}

return 0;
}

/****** THIS FUNCTION SUMS THE REAL COMPONENTS OF THE PROTON PHASES *******/
int sum_cosarray(void) {
    int i;
    cos_sum = 0;
    for (i=0; i<NUM_PROTONS; i++) {
        cos_sum = cos_sum + cosarray[i];
    }

    return 0;
}

/****** THIS FUNCTION SUMS THE IMAGINARY COMPONENTS OF THE PROTON PHASES *******/
int sum_sinarray(void) {
    int i;
```
sin_sum = 0;
for(i=0;i<NUM_PROTONS;i++) {
    sin_sum = sin_sum + sinarray[i];
}

return 0;
}

/******* MAIN PROGRAM *******
int main(void) {

    /* Seed random number generators */
srand( (unsigned)time( NULL ) );
srand48( (unsigned)time( NULL ) );

    /* Initialize variables */
    int offset = ceil(MODEL_LENGTH / 2);
double mean_rad = MEAN_RADIUS;
double numprotons = NUM_PROTONS;

    /* Call subroutines for Monte Carlo simulation */
builduniverse(mean_rad);
printf("Model constructed\n");
pickstart(offset,mean_rad);

38
walksim(offset);

printf("Simulation complete\n");

sum_phase();

gtcos_array();

get_sin_array();

sum_cosarray();

sum_sinaarray();

/* Calculate T2' value */

double signal = (1.0 / numprotons) * sqrt(pwr((cos_sum, 2) + pwr((sin_sum, 2)));

double R2 = -log(signal) * 1000 / TE;

double T2 = 1/R2;

printf("T2: %.9f\n", T2);

}

/* END OF PROGRAM */
APPENDIX B

MONTE CARLO WALK SIMULATION PROGRAM IN MPI

/******************** MONTE CARLO WALK SIMULATION **********************/
/* Code written by Jonathan Kopechek, The Ohio State University */
/* */
/* This C program performs a Monte Carlo simulation to find the T2' value of */
/* the model. The following parameters may be edited to modify the simulation: */
/* */
/* */
/* 1. STATICFIELD */
/* 2. MEAN PERTURBER RADIUS */
/* 3. MRI SEQUENCE TYPE */
/* 4. MODEL_LENGTH */
/* 5. MAGNETIC SUSCEPTIBILITY OUTSIDE PERTURBERS */
/* 6. MAGNETIC SUSCEPTIBILITY INSIDE PERTURBERS */
/* 7. STEPLENGTH */
/* 8. SEQTYPE */
/* 9. TE */

/* 10. NUM_PROTONS */

/* */

****************************************************************************************************

******** EDIT THE FOLLOWING PARAMETERS TO MODIFY THE MODEL ***********/
#define STATICFIELD 7 // Background magnetic field (Tesla)
#define MEAN_RADIUS 5 // Average radius of magnetic field perturber (nm)
#define MODEL_LENGTH 1024 // Length of model (nanometers)
#define CONCENTRATION 0.00001 // Perturber concentration
#define MU_OUTSIDE 1.000000 // Magnetic susceptibility outside perturber
#define MU_INSIDE 1.000006 // Magnetic susceptibility inside perturber
#define STEPLENGTH 64 // Gridsize (nanometers)

/* Define constants for Monte Carlo simulation */
#define SEQTYPE 0 // Sequence type (1 for spin-echo, 0 for gradient echo)
#define TE 100 // Echo time (milliseconds)
#define NUM_PROTONS 2000 // Number of protons to simulate
#define D 1300000 // Diffusion coefficient (nanometers^2/milliseconds)
#define dt 0.02 // Step time (seconds)
#define RESOLUTION 1 // Resolution for diffusion length

*************** DO NOT EDIT BELOW THIS LINE ***************
/ * Include files */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <mpi.h>

/* Declare functions */
int pwr(int value,int power);
int pwrfd(double value,int power);
int builduniverse(double mean_rad);
int pickstart(int offset,double mean_rad,int numprocessors,int rank);
int walksim(int offset,int numprocessors);
int sum_phase(int numprocessors, int rank);
int get_cos_array();
int get_sin_array();
int sum_cosarray();
int sum_sinarray();

int evalphase(double xnew,double ynew,double znew,double xold,
double yold,double zold,int indicator,int offset);

/* Declare program arrays */
double fieldarray[MODEL_LENGTH/STEPLENGTH][MODEL_LENGTH/STEPLENGTH]
[MODEL_LENGTH/STEPLENGTH];
int perturbarray[MODEL_LENGTH/STEPLENGTH][MODEL_LENGTH/STEPLENGTH][MODEL_LENGTH/STEPLENGTH];
double startloc[3][NUM_PROTONS];
double phasearray[NUM_PROTONS][5001];
double cumulativephase[NUM_PROTONS];
double temp_phasebuffer[NUM_PROTONS];
double cosarray[NUM_PROTONS];
double sinarray[NUM_PROTONS];

/* Initialize variables */
double standard_dev = 1; // Standard deviation of perturber radii
double pi = 3.14159;
double phase, cos_sum, sin_sum;

/*********** THIS FUNCTION PICKS THE STARTING LOCATION FOR EACH PROTON ***********/
int pickstart(int offset, double mean_rad, int numprocessors, int rank) {
    int repick = ceil(NUM_PROTONS/(20*numprocessors));
double xstart[repick];
double ystart[repick];
double zstart[repick];

    /* Clear variables */
double xnew = 0;
double ynew = 0;
double znew = 0;
int k, m, x0, y0, z0;

/* Randomly choose proton starting locations */
for (k=0; k<repick; k++) {
    xstart[k] = drand48()*MODEL_LENGTH;
    ystart[k] = drand48()*MODEL_LENGTH;
    zstart[k] = drand48()*MODEL_LENGTH;
}

/* Repick starting locations if proton is located within
perturber */
for (m=0; m<repick; m++) {
    x0 = floor(xstart[m]/STEPLENGTH);
    y0 = floor(ystart[m]/STEPLENGTH);
    z0 = floor(zstart[m]/STEPLENGTH);

    while (perturbarray[x0][y0][z0] == 1) {

        x0 = floor(xstart[m]/STEPLENGTH);
        y0 = floor(ystart[m]/STEPLENGTH);
    }
}
z0 = floor(zstart[m]/STEPLENGTH);

    // Randomly select next proton position
    double xg = drand48() - 0.5;
    double yg = drand48() - 0.5;
    double zg = drand48() - 0.5;

    xnew = xstart[m] + ( xg * ((sqrt(2 * D * dt))) * RESOLUTION);
    ynew = ystart[m] + ( yg * ((sqrt(2 * D * dt))) * RESOLUTION);
    znew = zstart[m] + ( zg * ((sqrt(2 * D * dt))) * RESOLUTION);
    xstart[m] = xnew; ystart[m] = ynew; zstart[m] = znew;

}

    // Calculate adjusted proton starting locations
    startloc[1][m] = xstart[m] * RESOLUTION + offset;
    startloc[2][m] = ystart[m] * RESOLUTION + offset;
    startloc[3][m] = zstart[m] * RESOLUTION + offset;

}

return 0;

}
int walksim(int offset, int numprocessors) {

    /* Initialize variables */
    int err, indicator, j, k;
    double xnew, ynew, znew, xold, yold, zold;

    /* Step proton through model */
    for (j=1; j<NUM_PROTONS/numprocessors; j++) {
        int index = ceil(j/20);
        xnew = startloc[1][index];
        ynew = startloc[2][index];
        znew = startloc[3][index];

        int accumphase = 0;
        for (k=1; k<ceil(TE / (2 * dt)); k++) {

            err = 1;
            indicator = 1;
            while (err == 1) {

                xold = xnew;
                yold = ynew;
                zold = znew;

            } /* End of while (err == 1) */

        } /* End of for (k=1; k<ceil(TE / (2 * dt)); k++) */

    } /* End of for (j=1; j<NUM_PROTONS/numprocessors; j++) */

} /* End of walksim(int offset, int numprocessors) */
// Randomly select next proton position

double xg = drand48() - 0.5;
double yg = drand48() - 0.5;
double zg = drand48() - 0.5;

xnew = xold + ( xg * ((sqrt(2 * D * dt))) * RESOLUTION);
ynew = yold + ( yg * ((sqrt(2 * D * dt))) * RESOLUTION);
znew = zold + ( zg * ((sqrt(2 * D * dt))) * RESOLUTION);

// Perform data check

if (floor(xnew) >= MODEL_LENGTH) { xnew = (MODEL_LENGTH)-1;
err = 0; }
if (floor(ynew) >= MODEL_LENGTH) { ynew = (MODEL_LENGTH)-1;
err = 0; }
if (floor(znew) >= MODEL_LENGTH) { znew = (MODEL_LENGTH)-1;
err = 0; }
if (xnew < 0) { xnew = 0; err = 0; }
if (ynew < 0) { ynew = 0; err = 0; }
if (znew < 0) { znew = 0; err = 0; }

int xloc = floor(xnew/STEPLength);
int yloc = floor(ynew/STEPLength);
int zloc = floor(znew/STEPLength);
err = perturbarray[xloc][yloc][zloc];

}

// Calculate new phase of proton
evalphase(xnew, ynew, znew, xold, yold, zold, indicator, offset);
phasearray[j][k] = phase;

xold = xnew;
yold = ynew;
zold = znew;
}

// Repeat walk, flip phase for spin-echo
for (k=(ceil(TE / (2 * dt))+1); k<ceil(TE / dt); k++) {

    err = 1;
    if (SEQTYPE == 1) { indicator=-1; }
    else { indicator=1; }

    while (err == 1) {

        xold = xnew;
                 
            }
yold = ynew;
zold = znew;

// Randomly select next proton position
double xg = drand48() - 0.5;
double yg = drand48() - 0.5;
double zg = drand48() - 0.5;

xnew = xold + ( xg * ((sqrt(2 * D * dt))) * RESOLUTION);
ynew = yold + ( yg * ((sqrt(2 * D * dt))) * RESOLUTION);
znew = zold + ( zg * ((sqrt(2 * D * dt))) * RESOLUTION);

// Perform data check
if (floor(xnew) >= MODEL_LENGTH) { xnew = (MODEL_LENGTH)-1;
  err = 0; }
if (floor(ynew) >= MODEL_LENGTH) { ynew = (MODEL_LENGTH)-1;
  err = 0; }
if (floor(znew) >= MODEL_LENGTH) { znew = (MODEL_LENGTH)-1;
  err = 0; }
if (xnew < 0) { xnew = 0; err = 0; }
if (ynew < 0) { ynew = 0; err = 0; }
if (znew < 0) { znew = 0; err = 0; }

int xloc = floor(xnew/STEPLENGTH);
int yloc = floor(ynew/STEPLENGTH);
int zloc = floor(znew/STEPLENGTH);

err = perturbarray[xloc][yloc][zloc];

}

// Calculate new phase of proton

evalphase(xnew,ynew,znew,xold,yold,zold,indicator,offset);
phasearray[j][k] = phase;

xold = xnew;
yold = ynew;
zold = znew;
}

}

return 0;
}

/********** THIS FUNCTION CALCULATES THE PROTON'S ACQUIRED PHASE **********/
int evalphase(double xn,double yn,double zn,double xo,double yo,
double zo,int indicator, int offset)
{
    int xnew = abs(floor(xn/STEPLENGTH));
    int ynew = abs(floor(yn/STEPLENGTH));
    int znew = abs(floor(zn/STEPLENGTH));
    int xold = abs(floor(xo/STEPLENGTH));
    int yold = abs(floor(yo/STEPLENGTH));
    int zold = abs(floor(zo/STEPLENGTH));
    double gamma = 267510;
    double averageB = (fieldarray[xnew][ynew][znew] + fieldarray[xold][yold][zold]) / 2;
    phase = gamma * averageB * dt * indicator;
    return 0;
}

int builduniverse(double mean_rad)
{

    double sphere_radius = mean_rad;

    /* Declare variables */
    double accumulator;
    double accumulatorold;
    double temporary;
    double diameter;

    51
double P;
double denominator;
int sphere;
int x,y,z,n,m;

/* Initialize variables */
double Bfield = STATICFIELD;

/* Initialize settings */
int arraysize = MODEL_LENGTH;
double arraysize_float = MODEL_LENGTH;
int step_length = STEPLENGTH;
int numspheres = ceil(arraysize_float*arraysize_float*
arraysize_float*CONCENTRATION/(4*pi*pwr(sphere_radius,3)/3));

/* Declare arrays */
int xpos[numspheres];
int ypos[numspheres];
int zpos[numspheres];

/* Randomly choose locations of magnetic field perturbers */
srand( (unsigned)time( NULL ) );
for (m=0; m<numspheres; m++) {
    xpos[m] = rand()%arraysize;
    ypos[m] = rand()%arraysize;
    zpos[m] = rand()%arraysize;
}

/* Create array of uniformly spaced intervals in universe and
specify locations of perturbers */
for (sphere=0; sphere<numspheres; sphere++) {
    for (x=xpos[sphere]-ceil(sphere_radius); x < (xpos[sphere]+ceil(sphere_radius)); x++) {
        for (y=ypos[sphere]-ceil(sphere_radius); y < (ypos[sphere]+ceil(sphere_radius)); y++) {
            for (z=zpos[sphere]-ceil(sphere_radius); z < (zpos[sphere]+ceil(sphere_radius)); z++) {
                if (((0 < x) & (arraysize > x) & (0 < y) & (arraysize > y) &
                    (0 < z) & (arraysize > z)) {
                    if (sqrt(pwr(x-xpos[sphere],2)+pwr(y-ypos[sphere],2)+
pwr(z-zpos[sphere],2)) <= sphere_radius) {
                        int xr = floor(x/step_length);
                        int yr = floor(y/step_length);
                        int zr = floor(z/step_length);
                        perturbarray[xr][yr][zr] = 1;
                    }
                }
/* Determine magnetic field values at each point in universe */
for (x=0; x<arraysize; x+=step_length) {
    for (y=0; y<arraysize; y+=step_length) {
        for (z=0; z<arraysize; z+=step_length) {
            int xr = floor(x/step_length);
            int yr = floor(y/step_length);
            int zr = floor(z/step_length);
            if (perturbarray[xr][yr][zr] == 1) {
                fieldarray[xr][yr][zr] = Bfield*(1+(MU_OUTSIDE-1)/3);
            }
            else if (perturbarray[xr][yr][zr] < 1) {
                accumulator = 0;
                for (sphere=0; sphere<numspheres; sphere++) {
                    accumulatorold = accumulator;
                    if ((pwr(x-xpos[sphere],2) + pwr(y-ypos[sphere],2) +
pwr(z-zpos[sphere],2)) < 1) {
                        temporary = 0;
                    }
                }
else if ((pwr(x-xpos[sphere],2) + pwr(y-ypos[sphere],2) + pwr(z-zpos[sphere],2)) > 0) {
    denominator = sqrt(pwr(x-xpos[sphere],2) + pwr(y-ypos[sphere],2) + pwr(z-zpos[sphere],2));
    temporary = (((MU_OUTSIDE-1)/3) + (MU_OUTSIDE-MU_INSIDE) * (sphere_radius*sphere_radius*sphere_radius) * (pwr(x-xpos[sphere],2) + pwr(y-ypos[sphere],2) - 2 * pwr(z-zpos[sphere],2))) / (3 * pwr(denominator,5)) ;
}
    accumulator = temporary + accumulatorold;
}
fieldarray[xr][yr][zr] = Bfield * (1 + accumulator);
}
}
}
return 0;


#define

/***** THIS FUNCTION COMPUTES THE POWER OF AN INTEGER ********/

int pwr(int value, int power) {
    int i;
    int result = 1;
    return 0;

}
for(i=0;i<power;i++) {
    result = result * value;
}

return result;

/******* THIS FUNCTION COMPUTES THE POWER OF A FLOAT *******/
int pwrfd(double value, int power) {
    int i;
    double result = 1;
    for(i=0;i<power;i++) {
        result = result * value;
    }
    return result;

}

/******* THIS FUNCTION SUMS THE PHASES FOR EACH PROTON *******/
int sum_phase(int numprocessors, int rank) {

    int i, j;
    for(i=0;i<NUM_PROTONS/numprocessors;i++) {

temp_phasebuffer[i] = 0;
for (j=0; j<ceil(TE / dt); j++) {
    temp_phasebuffer[i] = temp_phasebuffer[i] + phasearray[i][j];
}

return 0;

/
****** THIS FUNCTION CALCULATES THE REAL PART OF THE PROTON PHASES
******/
int get_cos_array(void) {
    int i;
    for (i=0; i<NUM_PROTONS; i++) {
        cosarray[i] = cos(cumulativephase[i]);
    }

    return 0;
}
/********** THIS FUNCTION CALCULATES THE IMAGINARY PART OF THE PROTON PHASES

**********/

int get_sin_array(void) {
    int i;
    for(i=0;i<NUM_PROTONS;i++) {
        sinarray[i] = sin(cumulativephase[i]);
    }

    return 0;
}


/********** THIS FUNCTION SUMS THE REAL COMPONENTS OF THE PROTON PHASES

**********/

int sum_cosarray(void) {
    int i;
    cos_sum = 0;
    for(i=0;i<NUM_PROTONS;i++) {
        cos_sum = cos_sum + cosarray[i];
    }

    return 0;
}
/******* THIS FUNCTION SUMS THE IMAGINARY COMPONENTS OF THE PROTON PHASES ***********/

int sum_sinarray(void) {
    int i;
    sin_sum = 0;
    for(i=0;i<NUM_PROTONS;i++) {
        sin_sum = sin_sum + sinarray[i];
    }

    return 0;
}

/******* MAIN PROGRAM ***********/

int main (int argc, char *argv[]) {

    /* Setting to initialize MPI */
    int err;

    int numprocessors, rank;
    MPI_Status status;
    err = MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocessors);
/* Seed random number generators */
srand( (unsigned)time( NULL ) );
srand48( rank * (unsigned)time( NULL ) );

/* Initialize variables */
int offset = ceil(ModelLength / 2);
double mean_rad = MEAN_RADIUS;
double numprotons = NUM_PROTONS;

if (rank == 0) {

/* Build model */
builduniverse(mean_rad);

} //if rank==0

/* Send model to other processors */
MPI_Bcast(&fieldarray,MODEL_LENGTH*MODEL_LENGTH*MODEL_LENGTH/
(STEPLength*STEPLength*STEPLength),MPI_DOUBLE,0,MPI_COMM_WORLD);

/* Call subroutines for Monte Carlo simulation */
pickstart(offset,mean_rad,numprocessors,rank);
walksim(offset,numprocessors);
sum_phase(numprocessors,rank);
/* Gather Monte Carlo walk data from other processors */
MPI_Gather(temp_phasebuffer,NUM_PROTONS/numprocessors,
MPI_DOUBLE,cumulativephase,NUM_PROTONS/numprocessors,MPI_DOUBLE,0,
MPI_COMM_WORLD);

if (rank == 0) {

get_cos_array();
get_sin_array();
sum_cosarray();
sum_sinarray();

/* Calculate T2' value */
double signal = (1.0 / numprotons) * sqrt(pwrf(cos_sum,2) +
pwrf(sin_sum,2));
double R2 = -log(signal) * 1000 / TE;
double T2 = 1/R2;
printf("T2: %.9f\n",T2);

}//if rank==0

} /* END OF PROGRAM */
APPENDIX C

MODEL DEVELOPMENT CODE (MATLAB)

% Bfield.m
% by Jonathan Kopechek
% This routine multiplies the magnetic field perturbation by the fourier %
% transform of impulses (located at perturers' origins) and takes the %
% inverse Fourier transform to find the magnetic field perturbations %
clear;
format long;

% SET PARAMETERS
radius = 4; % Radius of spherical perturber (nanometers)
arraysize = 1024; % Max distance away from center of perturber (nm)
step_length = 16; % Length of each step (nm)
mu_e=1; % Susceptibility outside perturber
mu_i=1.0000006; % Susceptibility inside perturber
diameter = radius*2;  % Diameter of spherical perturber (nanometers)
del_kc = 1;  % Sampling interval in k-space
numspHERES = 3;  % Number of spherical perturbers

% CREATE DATA ARRAYS
fieldarray=zeros(arraysize/step_length,arraysize/step_length,arraysize/
step_length);
perturbarray=fieldarray;

% SET PERTURBER LOCATION
%xpos=ceil(rand(1,numspheres)*arraysize);  % Location of perturber origin
on x-axis (nm)
xpos=ceil(rand(1,numspheres)*实训(arraysize-diameter)+radius);  % Location of
perturber origin on x-axis (nm)
ypos=ceil(rand(1,numspheres)*实训(arraysize-diameter)+radius);  % Location of
perturber origin on y-axis (nm)
zpos=ceil(rand(1,numspheres)*实训(arraysize-diameter)+radius);  % Location of
perturber origin on z-axis (nm)

% SPECIFY PERTURBER LOCATIONS WITHIN UNIVERSE
for sphere=1:1:numspheres
  for x=(xpos(sphere)-ceil(radius)+1):(xpos(sphere)+(ceil(radius)+1))
    for y=(ypos(sphere)-(ceil(radius)+1)):ypos(sphere)+(ceil(radius)+1))
      for z=(zpos(sphere)-(ceil(radius)+1)):zpos(sphere)+(ceil(radius)+1))

if (0 < x) & (arraysize > x) & (0 < y) & (arraysize > y) & (0 < z) & (arraysize > z)
    if sqrt((x-xpos(sphere)).^2+(y-ypos(sphere)).^2+(z-zpos(sphere)).^2) <= radius
        perturbarray(ceil(x/step_length),ceil(y/step_length),ceil(z/step_length))=1;
    end
end
end
end
end

% SCAN THROUGH UNIVERSE AND CALCULATE MAGNETIC FIELD PERTURBATIONS
for x=1:step_length:arraysize
    for y=1:step_length:arraysize
        for z=1:step_length:arraysize
            if perturbarray(ceil(x/step_length),ceil(y/step_length),ceil(z/step_length))=1
                fieldarray(ceil(x/step_length),ceil(y/step_length),ceil(z/step_length))=(mu_e-1)/3; % Magnetic field perturbation within sphere
                elseif perturbarray(ceil(x/step_length),ceil(y/step_length),ceil(z/step_length)) < 1
                    for sphere=1:1:numspheres
                        % Calculate magnetic field perturbation for sphere
                        % ... (code snippet continues)
fieldarray(ceil(x/step_length), ceil(y/step_length), ceil(z/step_length)) = ((mu_e-1)/3*(mu_e-mu_i)*(radius^3)*((x-xpos(sphere))^2+(y-ypos(sphere))^2-2*(z-zpos(sphere)))/3*(((x-xpos(sphere))^2+(y-ypos(sphere))^2+(z-zpos(sphere))^2)^5/2)/fieldarray(ceil(x/step_length), ceil(y/step_length), ceil(z/step_length)); % Magnetic field perturbation outside sphere
end
end
end
end

% SAMPLE THROUGH SPACE
for i=1:1:arraysize/(del_kc*step_length)
    for j=1:1:arraysize/(del_kc*step_length)
        for k=1:1:arraysize/(del_kc*step_length)
            B_kspace_courseK(i,j,k) = fieldarray(i*del_kc,j*del_kc,k*del_kc);
        end
    end
end

% CALCULATE FOURIER TRANSFORM OF PERTURBATION DATA
FFTfieldarray = my3Dfftshift(fftn(my3Dfftshift(B_kspace_courseK)));
% CALCULATE NEW REDUCED ARRAYSIZE
arraysize = length(FFTfieldarray);  % Length of universe

% CONVOLVE (MULTIPLY IN FREQUENCY DOMAIN)
j=sqrt(-1);
for kx=1:1:arraysize
    for ky=1:1:arraysize
        for kz=1:1:arraysize
            imp=0;
            for sphere=1:1:numspheres
                ddx=xpos(sphere)*2*pi/(arraysize/step_length);
                ddy=ypos(sphere)*2*pi/(arraysize/step_length);
                ddz=zpos(sphere)*2*pi/(arraysize/step_length);
                imp = abs(exp(-j*ddx-j*ddy-j*ddz))+imp;
            end
            combined_Bfield(kx,ky,kz) = (FFTfieldarray(kx,ky,kz) .* imp);
        end
    end
end

% TAKE INVERSE FOURIER TRANSFORMS
Bfield_in = my3Dfftshift(ifftn(my3Dfftshift(combined_Bfield)));

% PLOT BFIELD
arraysize = length(Bfield_in);
A(1:arraysize,1:arraysize) = Bfield_in(arraysize/4:arraysize,1:arraysize);
[X,Y] = meshgrid(1:1:arraysize);
Z(X,Y) = A(X,Y);
figure(2);
meshc(X,Y,Z);
title('bfMagnetic field pattern due to spherical perturber at center of cube');
zlabel('DeltaB - Magnetic field difference from static field');
ylabel('z direction');
xlabel('x-y direction');

arraysize = length(fieldarray);
A2(1:arraysize,1:arraysize) = fieldarray(arraysize/4:arraysize,1:arraysize);
[X2,Y2] = meshgrid(1:1:arraysize);
Z2(X2,Y2) = A2(X2,Y2);
figure(4);
meshc(X2,Y2,Z2);
title('bfMagnetic field perturbation map due to randomly-placed spherical perturbers');
zlabel('itDeltaB - Magnetic field perturbation (T)');
ylabel('itz-axis (nm)');
xlabel('itx/y-axis (nm)');

CUSTOM MATLAB FUNCTION
function y = my3Dfftshift(x)

% GET ARRAYSIZE
m = size(x,3);

% FIND MIDDLE OF ARRAY
p = ceil(m/2);

% CREATE NEW ARRAY
y = zeros(m,m,m);

% SWAP BLOCKS OF DATA
y(1:p,1:p,1:p) = x(p+1:m,1:p,p+1:m);
y(p+1:m,1:p,1:p) = x(p+1:m,p+1:m,p+1:m);
y(1:p,p+1:m,1:p) = x(p+1:m,p+1:m,1:p);

68
\[y(1:p,1:p,p+1:m) = x(1:p,p+1:m,p+1:m);\]
\[y(p+1:m,1:p,p+1:m) = x(1:p,1:p,1:p);\]
\[y(p+1:m,p+1:m,p+1:m) = x(p+1:m,1:p,1:p);\]
\[y(p+1:m,p+1:m,1:p) = x(1:p,p+1:m,1:p);\]
\[y(1:p,p+1:m,p+1:m) = x(1:p,1:p,p+1:m);\]
APPENDIX D

MICROCUBE DEVELOPMENT FOR UNIFORM GRID APPROACH

/* This program builds the array of magnetic field values on the micrometer scale */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

/* Declare & define global variables */
#define CONCENTRATION 0.02 // Concentration of spherical perturbers
#define BLOCK_LENGTH 64 // Length of one side of a block (nanometers)
#define MEAN_RADIUS 5 // Average perturber radius (nanometers)
#define NUM_BLOCKS 8 // Number of blocks in one direction
#define NUM_PTS 8 // Number of points in one direction for one block
#define STATICFIELD 7.0 // Static magnetic field (Tesla)
#define BUFFER_LENGTH 32 // Overlap zone between blocks (nanometers)
/* Function Declarations */
int builduniverse(int xblock, int yblock, int zblock);
int pwr(int value, int power);
double pwrfd(double value, int power);

/* Declare arrays */
double pixelarray[NUM_BLOCKS*NUM_PTS][NUM_BLOCKS*NUM_PTS][NUM_BLOCKS*NUM_PTS];
double temparray[NUM_BLOCKS*NUM_PTS][NUM_BLOCKS*NUM_PTS][NUM_BLOCKS*NUM_PTS];
double perturbarray[BLOCK_LENGTH+BUFFER_LENGTH][BLOCK_LENGTH+BUFFER_LENGTH]
[BLOCK_LENGTH+BUFFER_LENGTH];
double avgB[NUM_BLOCKS][NUM_BLOCKS][NUM_BLOCKS];

// Initialize variables
double standard_dev = 1; // standard deviation of perturber radii in micrometers
double mu_e = 1.0000000;
double mu_i = 1.0000006;
double pi = 3.14159;

void main () {

if (BUFFER_LENGTH < 2*BLOCK_LENGTH/NUM_PTS) {
printf("WARNING! Buffer length must be greater than or equal to
%d nanometers.\n\n", 2*BLOCK_LENGTH/NUM_PTS);
FILE *fp;
fp = fopen("avgB.dat", "a");

srand( (unsigned)time( NULL ) );
srand48( (unsigned)time( NULL ) );

//CALCULATE MAGNETIC FIELDS EXPERIENCED
printf("Program started\n");

int numspheres = ceil(pwr(BLOCK_LENGTH,3)*CONCENTRATION/(4*pi*pwr(MEAN_RADIUS,3)/3));
int numblocks = NUM_BLOCKS;

for (int xblock=0; xblock < numblocks; xblock++) {
  for (int yblock=0; yblock < numblocks; yblock++) {
    for (int zblock=0; zblock < numblocks; zblock++) {
      builduniverse(xblock,yblock,zblock);
    }
  }
}

int xavg,yavg,zavg;
//SAVE PIXELARRAY TO FILE

for (int x_read=0; x_read < NUM_BLOCKS*NUM_PTS; x_read++) {
    for (int y_read=0; y_read < NUM_BLOCKS*NUM_PTS; y_read++) {
        for (int z_read=0; z_read < NUM_BLOCKS*NUM_PTS; z_read++) {
            xavg = ceil(x_read/NUM_PTS);
            yavg = ceil(y_read/NUM_PTS);
            zavg = ceil(z_read/NUM_PTS);
            avgB[xavg][yavg][zavg] = avgB[xavg][yavg][zavg] + pixelarray[x_read][y_read][z_read]/(NUM_PTS*NUM_PTS*NUM_PTS);
        }
    }
}

for (int xblock=0; xblock < NUM_BLOCKS; xblock++) {
    for (int yblock=0; yblock < NUM_BLOCKS; yblock++) {
        for (int zblock=0; zblock < NUM_BLOCKS; zblock++) {
            fprintf(fp,"%.15f\n",avgB[xblock][yblock][zblock]-7);
        }
    }
}
};//end main

// This function creates a nanometer-scale block and calculates the magnetic
fields experienced at regular intervals
int builduniverse(int xblock, int yblock, int zblock)
{

// Declare variables
double accumulator, accumulatorold, temporary, param, g;
double x, y, z, denominator;
int sphere;
double xlimit, ylimit, zlimit;
int xfield_iteration, yfield_iteration, zfield_iteration, field_iteration;
int x_indx, y_indx, z_indx;
int xoffset, yoffset, zoffset;
double x_d, y_d, z_d;

// Initialize variables
double mean_rad = MEAN_RADIUS;
int numbblocks = NUM_BLOCKS;
int numspheres = ceil(pwr(BLOCK_LENGTH,3)*CONCENTRATION/(4*pi*
pwrf(mean_rad,3)/3));

// Declare arrays
int xpos[numspheres];
int ypos[numspheres];
int zpos[numspheres];
double radius[numspheres];

// Randomly generate spherical perturber radii with gaussian distribution
for (int n=0; n<numspheres; n++) {
    param = -2.5 + (5*n/numspheres);
    g = 20 * param * drand48() * (1 / (standard_dev * sqrt(2*pi))) *
        exp(-(pwrf((param),2))) / (2*pwrf(standard_dev,2));
    radius[n] = g + mean_rad;
    radius[n] = 4.0;
}

// Randomly choose locations of magnetic field perturbers
srand((unsigned)time(NULL));
for (int n=0; n<numspheres; n++) {
    xpos[n] = rand()%BLOCK_LENGTH;
    ypos[n] = rand()%BLOCK_LENGTH;
    zpos[n] = rand()%BLOCK_LENGTH;
}

int xa, ya, za;
// Create array of points spaced at 1-nm intervals and specify locations of
spherical perturbers
for (sphere=0; sphere<numspheres; sphere++) {
  for (x=xpos[sphere]-ceil(radius[sphere]); x < (xpos[sphere]+ceil(radius
  [sphere])); x++) {
    for (y=ypos[sphere]-ceil(radius[sphere]); y < (ypos[sphere]+ceil
      (radius[sphere])); y++) {
      for (z=zpos[sphere]-ceil(radius[sphere]); z < (zpos[sphere]+ceil
        (radius[sphere])); z++) {
        if ((0 < x) & (BLOCK_LENGTH > x) & (0 < y) & (BLOCK_LENGTH
            > y) & (0 < z) & (BLOCK_LENGTH > z)) {
          if (sqrt(pwr(x-xpos[sphere],2)+pwr(y-ypos[sphere],2)+
            pwr(z-zpos[sphere],2)) <= radius[sphere]) {
            xa = round(x+BUFFER_LENGTH/2);
            ya = round(y+BUFFER_LENGTH/2);
            za = round(z+BUFFER_LENGTH/2);

            perturbarray[xa][ya][za] = 1;

          }
        }
      }
    }
  }
}
// The following code calculates the magnetic fields experiences at regular
intervals within each block

if (xblock == 0) { xoffset = BUFFER_LENGTH/2; }
else { xoffset = 0; }
if (yblock == 0) { yoffset = BUFFER_LENGTH/2; }
else { yoffset = 0; }
if (zblock == 0) { zoffset = BUFFER_LENGTH/2; }
else { zoffset = 0; }

if (xblock == 0 || (xblock+1) == NUM_BLOCKS) { xlimit = BLOCK_LENGTH +
BUFFER_LENGTH/2; }
else { xlimit = BLOCK_LENGTH + BUFFER_LENGTH; }

else { ylimit = BLOCK_LENGTH + BUFFER_LENGTH; }

if (yblock == 0 || (yblock+1) == NUM_BLOCKS) { ylimit = BLOCK_LENGTH +
BUFFER_LENGTH/2; }
else { ylimit = BLOCK_LENGTH + BUFFER_LENGTH; }

if (zblock == 0 || (zblock+1) == NUM_BLOCKS) { zlimit = BLOCK_LENGTH +
BUFFER_LENGTH/2; }
else { zlimit = BLOCK_LENGTH + BUFFER_LENGTH; }

77
for (x=0; x<xlimit; x+=BLOCK_LENGTH/NUM_PTS) {
    if (xblock == 0) { x_index = xblock*NUM_PTS + x/(BLOCK_LENGTH/NUM_PTS); }
    else { x_index = xblock*NUM_PTS + x/(BLOCK_LENGTH/NUM_PTS) - BUFFER_LENGTH/(2*(BLOCK_LENGTH/NUM_PTS)); }

    if (x_index < (NUM_PTS - BUFFER_LENGTH/(2*BLOCK_LENGTH/NUM_PTS)) || x_index > ((NUM_BLOCKS-1)*NUM_PTS + BUFFER_LENGTH/(2*BLOCK_LENGTH/NUM_PTS) - 1) ) { xfield_iteration = NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*NUM_PTS))/(BLOCK_LENGTH/NUM_PTS); }
    else if (x_index > ((xblock)*NUM_PTS - BUFFER_LENGTH/(2*BLOCK_LENGTH/NUM_PTS) - 1) && x_index < (xblock)*NUM_PTS + BUFFER_LENGTH/(2*BLOCK_LENGTH/NUM_PTS) ) { xfield_iteration = 2*NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*NUM_PTS))/(BLOCK_LENGTH/NUM_PTS); }
    else if (x_index > ((xblock+1)*NUM_PTS - BUFFER_LENGTH/(2*BLOCK_LENGTH/NUM_PTS) - 1) && x_index < (xblock+1)*NUM_PTS + BUFFER_LENGTH/(2*BLOCK_LENGTH/NUM_PTS) ) { xfield_iteration = 2*NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*NUM_PTS))/(BLOCK_LENGTH/NUM_PTS); }
    else { xfield_iteration = NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*NUM_PTS))/(BLOCK_LENGTH/NUM_PTS); }
}

for (y=0; y<ylimit; y+=BLOCK_LENGTH/NUM_PTS) {
    if (yblock == 0) { y_index = yblock*NUM_PTS + y/(BLOCK_LENGTH/NUM_PTS); }
    else { y_index = yblock*NUM_PTS + y/(BLOCK_LENGTH/NUM_PTS) -

78
BUFFER_LENGTH/(2*(BLOCK_LENGTH/NUMPTS));

if (y_idx < (NUMPTS - BUFFER_LENGTH/(2*BLOCK_LENGTH/NUMPTS)) || y_idx
> ((NUM_BLOCKS-1)*NUMPTS + BUFFER_LENGTH/(2*BLOCK_LENGTH/NUMPTS) - 1) )
{ yfield_iteration = NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*NUMPTS))/
(BLOCK_LENGTH/NUMPTS); }
else if (y_idx > ((yblock)*NUMPTS - BUFFER_LENGTH/(2*BLOCK_LENGTH/
NUMPTS)) - 1) && y_idx < (yblock)*NUMPTS + BUFFER_LENGTH/(2*BLOCK_LENGTH/
NUMPTS) ) { yfield_iteration = 2*NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*
NUMPTS))/(BLOCK_LENGTH/NUMPTS); }
else if (y_idx > ((yblock+1)*NUMPTS - BUFFER_LENGTH/(2*BLOCK_LENGTH/
NUMPTS)) - 1) && y_idx < (yblock+1)*NUMPTS + BUFFER_LENGTH/(2*BLOCK_LENGTH/
NUMPTS) ) { yfield_iteration = 2*NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*
NUMPTS))/(BLOCK_LENGTH/NUMPTS); }
else { yfield_iteration = NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*
NUMPTS))/(BLOCK_LENGTH/NUMPTS); }

for (z=0; z<zlimit; z+=BLOCK_LENGTH/NUMPTS) {

if (zblock == 0) { z_idx = zblock*NUMPTS + z/(BLOCK_LENGTH/
NUMPTS); }
else { z_idx = zblock*NUMPTS + z/(BLOCK_LENGTH/NUMPTS) -
BUFFER_LENGTH/(2*(BLOCK_LENGTH/NUMPTS)); }

79
if (z_idx < (NUM_PTS - BUFFER_LENGTH/(2*BLOCK_LENGTH/NUM_PTS))) ||
    z_idx > ((NUM_BLOCKS-1)*NUM_PTS + BUFFER_LENGTH/(2*BLOCK_LENGTH/NUM_PTS)
      - 1)) { zfield_iteration = NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*NUM_PTS))
        / (BLOCK_LENGTH/NUM_PTS); }
else if (z_idx > ((zblock)*NUM_PTS - BUFFER_LENGTH/(2*BLOCK_LENGTH/
      NUM_PTS) - 1) && z_idx < (zblock)*NUM_PTS + BUFFER_LENGTH/(2*BLOCK_LENGTH/
      NUM_PTS)) { zfield_iteration = 2*NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*
      NUM_PTS))/(BLOCK_LENGTH/NUM_PTS); }
else if (z_idx > ((zblock+1)*NUM_PTS - BUFFER_LENGTH/(2*BLOCK_LENGTH/
      NUM_PTS) - 1) && z_idx < (zblock+1)*NUM_PTS + BUFFER_LENGTH/(2*BLOCK_LENGTH/
      NUM_PTS)) { zfield_iteration = 2*NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*
      NUM_PTS))/(BLOCK_LENGTH/NUM_PTS); }
else { zfield_iteration = NUM_BLOCKS*(BLOCK_LENGTH/(NUM_BLOCKS*
      NUM_PTS))/(BLOCK_LENGTH/NUM_PTS); }

field_iteration = xfield_iteration*yfield_iteration*zfield_iteration;

int xloc = x + xoffset;
int yloc = y + yoffset;
int zloc = z + zoffset;

if (perturbarray[xloc][yloc][zloc] == 1) {
pixelarray[x_idx][y_idx][z_idx] = (pixelarray[x_idx]
[y_idx][z_idx] + (STATICFIELD*(1+(mu_e-1)/3))/field_iteration);

80
}  
   else {  
   accumulator = 0;  
   for (sphere=0; sphere<numSpheres; sphere++) {  
   accumulatorold = accumulator;  
         
   if ((x - xoffset) == xpos[sphere]) { x_d = 0; }  
   else { x_d = pwr(x-xoffset-xpos[sphere],2); }  
   if ((y - yoffset) == ypos[sphere]) { y_d = 0; }  
   else { y_d = pwr(y-yoffset-ypos[sphere],2); }  
   if ((z - zoffset) == zpos[sphere]) { z_d = 0; }  
   else { z_d = pwr(z-zoffset-zpos[sphere],2); }  

   denominator = sqrt(x_d + y_d + z_d);  
   temporary = STATICFIELD * (((mu_e-1)/3)+(mu_e-mu_i)*  
   (pwr(radius[sphere],3))*((x_d + y_d - 2 * z_d)) / (3 * pwr(denominator,5)) );  
   if (x_d == 0 && y_d == 0 && z_d == 0)  
   { temporary = 0; };  
   if (temporary < -100) { temporary = 0; }  
   if (temporary > 100) { temporary = 0; }  
   accumulator = temporary + accumulatorold;  

   }  
   pixelarray[x_idx][y_idx][z_idx] = (pixelarray[x_idx]
[y_idx][z_idx] + (STATICFIELD + accumulator)/field_iteration;
}

}

}

}

return 0;

//end builduniverse function

// This function computes the power of an integer
int pwr(int value,int power) {
int result = 1;
for(int i=0;i<power;i++) {
result = result * value;
}
return result;
}

// This function computes the power of a float
double pwrf(double value,int power) {
double result = 1;
for(int i=0;i<power;i++) {
    result = result * value;
}

return result;

// END OF FILE
BIBLIOGRAPHY


