COMPUTATIONAL MAGNETIC THIN FILM DYNAMICS

JASON I. MERCER

Your file Votre référence
ISBN: 978-0-494-33436-2
Our file Notre référence
ISBN: 978-0-494-33436-2

Computational Magnetic Thin Film Dynamics

by

© Jason I. Mercer
BScCS (University of New Brunswick)

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Masters of Computational Science.

Department of Computational Sciences
Memorial University of Newfoundland

October 3, 2007

ST. JOHN'S NEWFOUNDLAND

Contents

Abstract

A numerical solution to the Landau-Lifshitz-Glibert (LLG) equation describing the dynamics of a set of interacting classical spins on multilayered geometries has been developed. This implementation has been parallelized using a set of custom developed shared memory utilities. Algebraic and computational optimizations have been devised and implemented which allow both high precision batch mode simulations and lower precision real-time interaction across networks.

Acknowledgements

I'm happy to be able to thank the people who have made this thesis possible.

Firstly, I thank my Supervisor, Dr. John Whitehead. His gentle assurance that physics was OK, there's a bug in the code and his seemingly limitless patience allowed me to explore wide avenues of physics, mathematics and computer science along the path to this work.

Dr. Keith De'Bell deserves recognition for introducing me to this field and providing a similar research environment and I thank him for that.

My fellow graduate students and friends provided a constant source of welcomed distractions and constructive conversations. It would have been a dry graduate program without them.

Finally, my family has an awful lot to do with we me being here today. Many thanks to my Mom and Dad, Alberta and Paul Mercer and my sister, Danielle Goguen.

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis we describe the development of methods to simulate the dynamics of lattice-based thin film magnetic systems using the Landau-Lifshitz-Gilbert (LLG) formulation which can take advantage of multiprocessor shared memory architectures. The simulated system consists of magnetic moments interacting via both exchange and dipolar fields and subject to surface anisotropy, external applied fields and perturbations derived from finite temperature effects.

While the LLG method has been extensively studied and widely implemented, this thesis presents a number of mathematical and computational improvements which will allow us to simulate larger systems over longer times than was previously feasible. This is important as it allows us to look at systems with mesoscopic magnetic structures such as the striped systems studied by Booth et. al.[2]. To study such systems we require very large lattices to minimize the influence of commensurate effects imposed by the boundary conditions on finite lattices and to study the behaviour of the system close to phase boundaries.

1.1 Outline

In Chapter 2, the physical model which we simulate is described. This model includes the structure of the lattice, boundary conditions, energies and interactions of the spins. The differential equation used to describe the evolution of the magnetic moments is also analyzed.

In Chapter 3, the details of the numerical integration technique is presented and discussed. The two main topics are the evaluation of the long range dipolar interaction and selecting a numerical integration method which best suits the dynamics.

Chapters 4 and 5 investigate problems and present solutions to implementing the numerical simulation in a parallel environment. Methods to parallelize the calculation of the dipolar interaction, optimally execute a group of parallel tasks and a discussion regarding the synchronization routines are included in these chapters.

Techniques for managing and analyzing the data produced by our simulation are presented in Chapter 6. Included in that chapter are methods for reading various datafiles in a convenient manner and visualizing the data.

Chapter 7 presents several demonstrations of the suite of simulation, analysis and visualization tools.

Chapter 2

Physics

2.1 Magnetic Model

In this thesis we model two dimensional magnetic systems as an array of classical spins on a lattice. The spins are represented by vectors of fixed length which, without loss of generality, we assume is unity. We restrict our considerations to square lattices, however these methods have been generalized to other lattice structures. The dynamics of the spins are determined by four interactions: exchange, dipolar, surface anisotropy and external applied fields. The energy of the spin systems is therefore written as

$$E = E^J + E^d + E^k + E^z, \tag{2.1}$$

where E^J, E^d, E^k, E^z represent the energies of the exchange interaction, dipolar interaction, surface anisotropy and applied field respectively.

Denoting a spin configuration of a lattice of N spins as $\{\vec{\sigma}_i\}$, where $\vec{\sigma}_i$ is the vector describing the direction of the i^{th} spin ($|\vec{\sigma}| = 1$), we may write each of these terms as

follows

$$E^J = -J \sum_{\langle i,j \rangle} \vec{\sigma}_i \cdot \vec{\sigma}_j \tag{2.2}$$

$$E^d = g \sum_{i,j}{}' \left(\frac{\vec{\sigma}_i \cdot \vec{\sigma}_j}{|\vec{r}_{ij}|^3} - 3 \frac{(\vec{\sigma}_i \cdot \vec{r}_{ij})(\vec{\sigma}_j \cdot \vec{r}_{ij})}{|\vec{r}_{ij}|^5} \right) \tag{2.3}$$

$$E^k = -\kappa \sum_i (\vec{\sigma}_i \cdot \hat{n}_i)^2 \tag{2.4}$$

$$E^z = -\sum_i \vec{\sigma}_i \cdot \vec{H}_i^z, \tag{2.5}$$

where the angled brackets denote nearest neighbours, the prime on the summation excludes $i = j$, $\hat{n}_i$ defines the easy axis at site i and $\vec{r}_{ij}$ is the vector describing the displacement between sites i and j. $\vec{H}_i^z$ is the applied field or Zeeman field at lattice site i. The origin of each of these terms is discussed in a number of standard texts on the subject [19][20][24].

2.2 Effective Field

The effective field, $\vec{H}_i^{\text{eff}}$, is the negative functional derivative of the sum of the above energies. Simply stated, the effective field at site i is the sum of all fields from interacting spins and the external field. Each energy listed above has an associated field such that $\vec{H}_i^\Delta = -\partial E^\Delta / \partial \vec{\sigma}_i$. Explicitly these are

$$\vec{H}_i^J = J \sum_{\langle i,j \rangle} \vec{\sigma}_j \tag{2.6}$$

$$\vec{H}_i^d = -g \sum_j{}' \frac{\vec{\sigma}_j}{r_{ij}^3} - \frac{3\vec{r}_{ij}(\vec{\sigma}_j \cdot \vec{r}_{ij})}{r_{ij}^5} \tag{2.7}$$

$$\vec{H}_i^\kappa = 2\kappa \hat{n} \, (\vec{\sigma} \cdot \hat{n}), \tag{2.8}$$

which are the exchange interaction, dipolar interaction and surface anisotropy at lattice site i. The energy terms may then be rewritten as

$$E^J = \sum_i \vec{\sigma}_i \cdot \vec{H}_i^J \tag{2.9}$$

$$E^d = \sum_i \vec{\sigma}_i \cdot \vec{H}_i^d \tag{2.10}$$

$$E^\kappa = \sum_i \vec{\sigma}_i \cdot \vec{H}_i^\kappa \tag{2.11}$$

$$E^z = \sum_i \vec{\sigma}_i \cdot \vec{H}_i^z. \tag{2.12}$$

The total system energy may therefore be written as

$$E = \sum_i \vec{\sigma}_i \cdot \vec{H}_i^{\text{eff}} \tag{2.13}$$

with

$$\vec{H}_i^{\text{eff}} = \vec{H}_i^J + \vec{H}_i^d + \vec{H}_i^\kappa + \vec{H}_i^z. \tag{2.14}$$

These effective fields play a role in the evaluation of the spin dynamics and allow us to define a "local" magnetic energy density

$$E_i = \vec{\sigma}_i \cdot \vec{H}_i^{\text{eff}}. \tag{2.15}$$

This quantity will prove useful in the analysis of non-equilibrium, particularly metastable, spin configurations.

2.3 Boundary Conditions

Ideally we would like to consider infinite lattices, however in simulations we are restricted to configurations of finite size. In order to minimize the effects of boundaries we consider

finite lattices but impose periodic boundary conditions. While the introduction of periodic boundary conditions is relatively straightforward for the exchange and surface anisotropy, the long range nature of the dipolar interaction requires more care. This care is required since the dipolar interaction is a sum over all spins which, under these boundary conditions, is now defined as a sum over all displaced replicas of spins.

To properly account for the dipolar interaction we consider an infinite lattice made up of T equivalent $L \times L$ tiles as shown in Figure 2.1. This figure depicts a finite sized spin

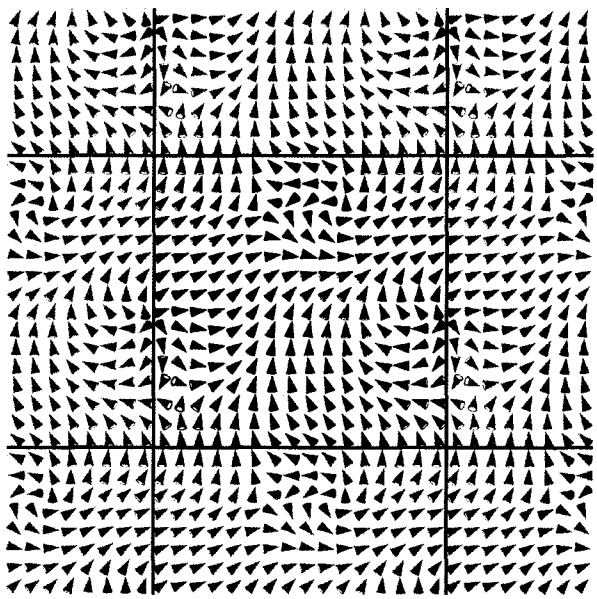


Figure 2.1: Replicated Spin Systems

system in black with tiled replicas on all sides, these replicas conceptually extend infinitely. Thus we consider the subset of spin configurations on an infinite lattice that satisfies the following periodicity

$$\vec{\sigma}(x,y) = \vec{\sigma}(x+mL, y+nL), \tag{2.16}$$

where m and n are integers. This allows us to write the effective field due to the dipolar interaction defined by Equation 2.7 as the sum over a cell of LxL spins as

$$H_i^d = \sum_j \Gamma_{ij}^{\alpha\beta} \vec{\sigma}_j, \tag{2.17}$$

with

$$\Gamma_{xy}^{\alpha\beta} = \sum_i \sum_j \frac{\delta^{\alpha\beta}}{r^3} - \frac{3r^\alpha r^\beta}{r^5} \tag{2.18}$$

$$\vec{r} = (x+iL, y+jL). \tag{2.19}$$

Where the Greek variables α and β are meant to represent combinations of Cartesian coordinates. These sums may be calculated exactly using Ewald Summation Techniques[15] or by direct summation. While the Ewald Summation method allows us to extract important features of the interaction analytically, direct summation is sufficiently precise for numerical calculations as long as the periodicity is in no more than two dimensions.

2.4 Static Spin Configuration Energies

In Tables 2.1 and 2.2, list at the end of this chapter, we present the energies for various spin configurations calculated using the methods outlined above. These analytical calculations agree with similar results presented in [8] and will serve as an important check for work presented later in this thesis. In this table, the subscripts x, y and z are the Cartesian coordinates of the spin with z being perpendicular to the plane. Lattice indexes i and j count in the $\hat{x}$ and $\hat{y}$ directions. The results obtained in Tables 2.1 and 2.2 illustrate a number of interesting features regarding the nature of the interaction and of the particular spin configurations.

1. The anisotropic nature of the dipolar interaction gives rise to a difference in energy between the planar and uniaxial spin configurations for both the antiferromagnetic and ferromagnetic cases. In the case of the ferromagnetic configuration, it favours the in plane orientation while the out of plane (uniaxial) orientation is preferred in the antiferromagnetic configuration.

2. The planar ferromagnetic configuration and the planar antiferromagnetic configuration are degenerate under both exchange and dipolar interactions. Indeed, calculations show that these spin configurations belong to a continuously degenerate manifold of states.

Both of these properties have an important bearing on the equilibrium and non-equilibrium properties of these systems.

2.5 Dynamics

The main thrust of this thesis is how we can evaluate the evolution of a lattice of interacting spins. The principle approach to the study of spin dynamics employs the so called Landau-Lifshitz-Gilbert (LLG) equation. The effects of temperature are usually included through a stochastic applied field. In order to understand the origins of the LLG equation we consider first the case of a single spin in an applied, possibly time dependent, field and include the effects of damping through a linear dissipative term. This may be readily generalized to include a lattice of interacting spins by replacing the applied field with the effective field defined in equation 2.14. The effects of temperature are realized with the inclusion of a Langevin forcing function.

2.5.1 LLG Equation

The system dynamics may be described by considering a single magnetic moment, $\vec{\sigma}$ in an external magnetic field $\vec{H}$. The angular momentum of the magnetic moment is given by

$$\vec{L} = \frac{\vec{\sigma}}{\gamma} \tag{2.20}$$

where $\gamma < 0$ is the gyromagnetic ratio. The torque on the spin due to the magnetic field is given as

$$\vec{\tau} = \vec{\sigma} \times \vec{H}. \tag{2.21}$$

The rate of change in angular momentum is therefore given by

$$\frac{d\vec{\sigma}}{dt} = -\gamma \vec{\sigma} \times \vec{H}, \tag{2.22}$$

which is the undamped Landau-Lifshitz equation.

In the case of constant field ($\dot{\vec{H}} = 0$) this equation describes the precession of a spin around an axis parallel to the direction of the applied field. It is straightforward to show that in a constant field the energy of the spin is also constant.

A more realistic model of a spin interaction is obtained by adding damping. Landau and Lifshitz achieved this by adding a torque perpendicular to the direction of precession which was scaled by the phenomenological constant $\lambda > 0$ which results in

$$\frac{d\vec{\sigma}}{dt} = -\gamma \vec{\sigma} \times \vec{H} - \lambda \vec{\sigma} \times (\vec{\sigma} \times \vec{H}). \tag{2.23}$$

Gilbert proposed a different approach[13] which modeled the damping after a viscous force of the form

$$\alpha \vec{\sigma} \times \frac{d\vec{\sigma}}{dt}, \tag{2.24}$$

were α is the Gilbert damping constant. Replacing Landau and Lifshitz's damping term with Gilbert's dissipative expression yields

$$\frac{d\vec{\sigma}}{dt} = -\gamma\vec{\sigma}\times\vec{H} + \alpha\vec{\sigma}\times\frac{d\vec{\sigma}}{dt}. \tag{2.25}$$

Solving for $\frac{d\vec{\sigma}}{dt}$, we arrive at the explicit form

$$\frac{d\vec{\sigma}}{dt} = -\frac{\gamma}{1+\alpha^2}\vec{\sigma}\times\vec{H} - \frac{\gamma\alpha}{(1+\alpha^2)}\vec{\sigma}\times(\vec{\sigma}\times\vec{H}) \tag{2.26}$$

which is the Landau-Lifshitz-Gilbert equation. The addition of the dissipative term to the Landau-Lifshitz equation both slows the precession and gradually aligns the spin with the field.

There are several important features to note in Equation 2.26. The first is that the dot product of $\vec{\sigma}$ and the equation is zero[33].

$$\vec{\sigma}\cdot\frac{d\vec{\sigma}}{dt} = \frac{\gamma}{(1+\alpha^2)}\vec{\sigma}\cdot(\vec{\sigma}\times\vec{H}) + \frac{\gamma\alpha}{(1+\alpha^2)}\vec{\sigma}\cdot(\vec{\sigma}\times(\vec{\sigma}\times\vec{H})) = 0 \tag{2.27}$$

expanding the left hand side we get

$$\frac{d\vec{\sigma}}{dt}\cdot\vec{\sigma} = \frac{1}{2}\frac{d(\vec{\sigma}\cdot\vec{\sigma})}{dt} = 0, \tag{2.28}$$

or

$$\frac{1}{2}\frac{d|\vec{\sigma}|^2}{dt} = 0, \tag{2.29}$$

which shows that the length of the spin does not change over time. The dynamics described by the LLG Equation preserve spin length. This is a key consequence which we will revisit in a later section.

Secondly, we can show that $d(\vec{\sigma}\cdot\vec{H})/dt$ is zero when α is zero

$$\left.\frac{d(\vec{\sigma}\cdot\vec{H})}{dt}\right|_{\alpha=0} = \vec{\sigma}\cdot\frac{d\vec{H}}{dt} + \vec{H}\cdot\frac{d\vec{\sigma}}{dt}, \tag{2.30}$$

since we have defined energy as $\vec{\sigma} \cdot \vec{H}$ and we are considering a constant field ($\frac{d\vec{H}}{dt} = 0$) we write

$$\left.\frac{dE}{dt}\right|_{\alpha=0} = \vec{H} \cdot \frac{d\vec{\sigma}}{dt}. \tag{2.31}$$

Using the definition of $\frac{d\vec{\sigma}}{dt}$ from Equation 2.26 with zero damping

$$\left.\frac{dE}{dt}\right|_{\alpha=0} = \gamma\vec{H} \cdot (\vec{\sigma} \times \vec{H}) = 0 \tag{2.32}$$

which states that, with damping set to zero, the energy will not change over time. As with the previous result, we will explore this in a later section.

2.6 Langevin Field

We have alluded to the presence of a thermal field but have yet to define its source or properties. This field is derived from environmental microscopic degrees of freedom such as phonons, conducting electrons or nuclear spin[12] and drives each magnetic moment along a Brownian-type rotation. The inclusion of the thermal field allows the system to escape from local energy minima and can aide in the relaxation process. Néel[30] first proposed this process in 1949 and latter Brown[4] further developed the concept by looking at it as a stochastic process.

We will implement the stochastic system as described by the Brown-Kubo-Hashitsume model. This model augments the effective field, $\vec{H}^{\text{eff}}$, with a thermal field, $\vec{H}^{th}$. We re-express the Landau-Lifshitz-Gilbert equation as the stochastic Landau-Lifshitz-Gilbert equation

$$\frac{d\vec{\sigma}}{dt} = \frac{\gamma}{(1+\alpha^2)}\vec{\sigma} \times (\vec{H} + \vec{H}^{th}) + \frac{\gamma\alpha}{(1+\alpha^2)}\vec{\sigma} \times (\vec{\sigma} \times (\vec{H} + \vec{H}^{th})). \tag{2.33}$$

The shape of the random thermal field is described by[12]

$$\langle \vec{H}_i^{th}(t) \rangle = 0 \tag{2.34}$$

$$\langle \vec{H}_i^{th}(t) \vec{H}_j^{th}(t+\Delta t) \rangle = 2D_{LLG}\delta_{ij}\delta(\Delta t), \tag{2.35}$$

for lattice sites i and j at times t and $t+\Delta t$. This definition states that there is no correlation in the field across sites or in time and that the time averaged fluctuations at a particular site is zero. The amplitude of the fluctuating field for use in the LLG equation follows from the fluctuation-dissipation theorem [32]

$$D_{LLG} = \frac{\alpha}{1+\alpha^2}\frac{k_B T}{\gamma}. \tag{2.36}$$

for temperature T, since the Néel time is

$$\tau_N = \frac{1}{\alpha}|\vec{\sigma}|2\gamma k_B T, \tag{2.37}$$

and

$$\frac{1}{\tau_N} = 2D\gamma^2(1+\alpha^2). \tag{2.38}$$

Therefore, we define the Langevin field as

$$\vec{H}_i^{th} = \sqrt{\frac{\alpha T}{\Delta t|\vec{\sigma}_i|}}\eta, \tag{2.39}$$

with η representing the normal distribution with mean zero and standard deviation 1 and Δt as the discrete time step.

2.7 Summary

This chapter describes the physical model that we wish to analyze. The energies and their associated effective fields have been given, we have proposed a method to minimize

edge effects by imposing periodic boundary conditions and we have derived the differential equation which describes the time dependant dynamics of the system. A comparison with energies that have been analytically calculated for some important spin configurations provides both a reference and a check for computations in later sections.

In the next chapter we will discuss implementation considerations and present methods to manage the complications which arise from the interactions and the differential equation.

System	Configuration	E^J	E^d	E^κ
Planar Ferromagnet	$\sigma_x = 1$ $\sigma_y = 0$ $\sigma_z = 0$	$-2.0J$	$-4.5168g$	0κ
Planar Ferromagnet	$\sigma_x = \sqrt{\frac{1}{2}}$ $\sigma_y = \sqrt{\frac{1}{2}}$ $\sigma_z = 0$	$-2.0J$	$-4.5168g$	0κ
Canted Ferromagnet	$\sigma_x = \sqrt{\frac{1}{2}}$ $\sigma_y = 0$ $\sigma_z = \sqrt{\frac{1}{2}}$	$-2.0J$	$2.2584g$	-0.5κ
Uniaxial Ferromagnet	$\sigma_x = 0$ $\sigma_y = 0$ $\sigma_z = 1$	$-2.0J$	$9.0336g$	-1κ
Planar Antiferromagnet (AA) phase	$\sigma_x = -1^{i+j}$ $\sigma_y = 0$ $\sigma_z = 0$	$2.0J$	$1.3229g$	0κ

Table 2.1: Energies of static configurations

System	Configuration	E^J	E^d	E^κ
Planar Antiferromagnet (AF) collinear phase	$\sigma_x = -1^j$ $\sigma_y = 0$ $\sigma_z = 0$	$0J$	$-5.0989g$	0κ
Planar Antiferromagnet Microvortex phase	$\sigma_x = \sqrt{\frac{1}{2}}(-1)^j$ $\sigma_y = \sqrt{\frac{1}{2}}(-1)^i$ $\sigma_z = 0$	$0J$	$-5.0989g$	0κ
Uniaxial Antiferromagnet (AA) phase	$\sigma_x = 0$ $\sigma_y = 0$ $\sigma_z = -1^{i+j}$	$2.0J$	$-2.6459g$	-1κ
Uniaxial Antiferromagnet (AF) phase	$\sigma_x = 0$ $\sigma_y = 0$ $\sigma_z = -1^j$	$0J$	$-0.9355g$	-1κ

Table 2.2: Energies of static configurations (continued)

Chapter 3

Algorithm

The LLG Equation, given in Equation 2.26, is very difficult to solve numerically. There are essentially three distinct aspects to the problem.:

- How do we accurately and efficiently compute the effective fields from a given spin configuration?

- How do we compute the evolution of a spin given the effective field at a particular site?

- How do we combine the above processes into an algorithm that is accurate, efficient and stable to determine the evolution of the entire lattice from a given set of initial conditions?

We will discuss each of these aspects of the numerical solution separately.

3.1 Calculating Effective Fields

As described in Chapter 2, the effective field $\vec{H}^{\text{eff}}$ is comprised of the sum of six separate terms. Three of these arise as a consequence of the interactions contained in the Hamiltonian for the model: the exchange interaction, the dipolar interaction and the surface anisotropy. The other three terms consist of the applied external field, the damping field and, in the finite temperature case, the Langevin Field.

The expressions for five of these fields are given by equations 2.6, 2.7, 2.8, 2.39 and the Zeeman field. The sixth, the damping field, is defined by the LLG Equation and expressed in terms of the effective field. The applied field, the field due to the surface anisotropy and the exchange can all be readily calculated at each site for an arbitrary spin configuration. The computation of the dipolar and Langevin fields require somewhat more thought.

The key to calculating the dipolar part of the effective field is to recall that we are dealing with a finite lattice with periodic boundary conditions. There are three methods available to us to calculate the dipolar field: explicit summation over replicated lattices, precomputed summations and precomputed summations in reciprocal space. The precomputed summations can be performed using one of two different methods.

3.1.1 Calculation of the Dipolar Fields

The dipolar field is given by Equation 2.7 as

$$\vec{H}_i^d = g \sum_{j \neq i} \frac{\vec{\sigma}_j}{r_{ij}^3} - \frac{3\vec{r}_{ij}(\vec{\sigma}_j \cdot \vec{r}_{ij})}{r_{ij}^5}. \tag{3.1}$$

This equation states that the dipolar field, $\vec{H}^{\text{d}}$ at site i has contributions from every other site in the system. Imposing periodic boundary conditions greatly simplifies the evaluation

of $\vec{H}_i^d$. Since we are dealing with replicated lattices, the spin at site $\{i, j\}$ is the same as the spin at site $\{i + aL, j + bL\}$ and so we can factor this spin out of the summation. We can define the dipolar field at site i as the contribution from every other spin on the lattice and each of their replicas plus the contribution from the replicas of spin i itself. The dipolar field at site i due to site j is

$$\vec{H}_{ij}^d = g \sum_{\vec{R}}{}' \frac{\vec{\sigma}_j}{|\vec{R}_{ij}|^3} - \frac{3\vec{R}_{ij}(\vec{\sigma}_j \cdot \vec{R}_{ij})}{|\vec{R}_{ij}|^5}, \tag{3.2}$$

where $\vec{R}$ are the original and replicated lattices, $\vec{R}_{ij}$ is the distance between site i on the original lattice and site j on the replicated lattice $\vec{R}$ and the prime on the summation imposes the condition that $\vec{R} \neq 0$ if $i = j$. Factoring out $\vec{\sigma}_j$ makes this expression independent of the spin configuration

$$(\vec{H}_{ij}^d)^\alpha = g \left(\sum_{\beta=\{x,y,z\}} \sigma_j^\beta \sum_{\vec{R}}{}' \frac{\delta^{\alpha\beta}}{|\vec{R}_{ij}|^3} - \frac{3R^\alpha R^\beta}{|\vec{R}_{ij}|^5} \right), \tag{3.3}$$

with Greek letters denoting Cartesian components. Using Einstein notation and replacing the inner summation and it's expression with $\Gamma_{ij}^{\alpha\beta}$ such that

$$\Gamma_{ij}^{\alpha\beta} = \sum_{\vec{R}}{}' \frac{\delta^{\alpha\beta}}{|\vec{R}_{ij}|^3} - \frac{3R^\alpha R^\beta}{|\vec{R}_{ij}|^5}, \tag{3.4}$$

we can write the effective dipolar field at site i due to the spin at site j and its replicas as

$$(\vec{H}_{ij}^d)^\alpha = g\Gamma_{ij}^{\alpha\beta}\sigma_j^\beta. \tag{3.5}$$

The dipolar field at site i from all other sites is therefore given by

$$(\vec{H}_i^d)^\alpha = g \sum_j \Gamma_{ij}^{\alpha\beta}\sigma_j^\beta. \tag{3.6}$$

3.1.2 Summation in Reciprocal Space

While the interaction matrix Γ_{ij} defined by Equation 3.4 may be computed and stored as a look-up table, the evaluation of the dipolar field by Equation 3.6 is of $O(N^2)$ and as such becomes computationally expensive when scaled to larger systems. A more sensible approach is to define the Discrete Fourier Transform (DFT) for both the spin variables, effective field and interaction matrix as follows

$$\vec{\sigma}(q_x, q_y) = \sum_{r_x=0}^{N-1} \sum_{r_y=0}^{N-1} \vec{\sigma}(r_x, r_y) e^{-i2\pi(q_x r_x + q_y r_y)\frac{1}{N}} \tag{3.7}$$

$$\vec{H}(q_x, q_y) = \sum_{r_x=0}^{N-1} \sum_{r_y=0}^{N-1} \vec{H}(r_x, r_y) e^{-i2\pi(q_x r_x + q_y r_y)\frac{1}{N}} \tag{3.8}$$

$$\Gamma^{\alpha\beta}(q_x, q_y) = \sum_{r_x=0}^{N-1} \sum_{r_y=0}^{N-1} \Gamma^{\alpha\beta}(r_x, r_y) e^{-i2\pi(q_x r_x + q_y r_y)\frac{1}{N}} \tag{3.9}$$

and the inverses as

$$\vec{\sigma}(r_x, r_y) = \frac{1}{N^2} \sum_{q_x=0}^{N-1} \sum_{q_y=0}^{N-1} \vec{\sigma}(q_x, q_y) e^{i2\pi(q_x r_x + q_y r_y)\frac{1}{N}} \tag{3.10}$$

$$\vec{H}(r_x, r_y) = \frac{1}{N^2} \sum_{q_x=0}^{N-1} \sum_{q_y=0}^{N-1} \vec{H}(q_x, q_y) e^{i2\pi(q_x r_x + q_y r_y)\frac{1}{N}} \tag{3.11}$$

$$\Gamma^{\alpha\beta}(r_x, r_y) = \frac{1}{N^2} \sum_{q_x=0}^{N-1} \sum_{q_y=0}^{N-1} \Gamma^{\alpha\beta}(q_x, q_y) e^{i2\pi(q_x r_x + q_y r_y)\frac{1}{N}} \tag{3.12}$$

It is straightforward to show that the $O(N^2)$ operation in real space is now an order $O(N)$ convolution in reciprocal space

$$H^{\alpha}(\vec{q}) = \Gamma^{\alpha\beta}(\vec{q})\sigma^{\beta}(\vec{q}). \tag{3.13}$$

Realizing that the evaluation of $\sigma(\vec{q})$ from $\sigma(\vec{r})$ and the evaluation of $H(\vec{r})$ from $H(\vec{q})$ using the DFT method both scale as $O(N\log(N))$ suggests that it would be easier to calculate the

dipolar fields by DFTs. The actual savings over the previous method depends on system size and DFT method but using standard FFT packages results calculations completing in $1/50^{th}$ the time for a $32x32$ system. Moving to larger systems results in a greater savings since this method scales more efficiently than the order $O(N^2)$ method. While in principle smaller systems minimize these savings, in practise we find that it is only for a 2x2 that the real space calculation is faster than the reciprocal space method.

3.2 Computing the Interaction Matrix

The precomputed interaction matrices are system size dependent. A 32×32 Γ^{xx} matrix can not be used in a 1024×1024 system but since each matrix is configuration independent, a single matrix can be reused for all calculations at a defined system dimension. Since a matrix is reusable, it has a one time cost of construction and so an optimal computation method is not essential. We will first look at the method which parallels the Explicit Summation of Dipolar Fields and then explore a faster method using the Ewald Summation Technique.

3.2.1 Explicit Interaction Matrix Summation

The easiest way to compute Equation 3.4 is a direct summation. We may truncate the summation to a radial distance of $10^{\frac{16}{3}}$ sites[1]. Furthermore, some of the matrices will be zero when we consider single layer problems, these matrices are the Γ^{xz} and Γ^{yz} matrices.

[1]This truncation is permissible as we are dealing with 64 bit floating point values, which have approximately 16 decimal digits in the mantissa, and the longest range component of the dipolar field rolls off as $\frac{1}{r^3}$

Counting FLOPs required to compute the cases when $\alpha = \beta$ yields 11 operations and 7 operations when $\alpha \neq \beta$. Summing out to a radial distance of $10^{\frac{16}{3}}$ results in 1.7×10^9 total FLOPs per matrix element for a 32×32 matrix at $\alpha = \beta$ and 1.1×10^9 FLOPs per matrix element when $\alpha \neq \beta$. Calculating the full interaction matrices for Γ^{xx}, Γ^{xy}, Γ^{yy} and Γ^{zz} totals 6.3×10^{12} FLOPs. Using computing resources running at 10 GFLOPS, this operation will take on the order of 10 minutes.

3.2.2 Interaction Matrix via Ewald Summation

The key to an improvement in runtime lies again in reciprocal space. Using the Ewald Summation method, we can deal with each term in 3.4 separately and divide each into two summations, one in real space, the other in reciprocal space. Appendix A describes this method in detail. For more information, Lo and Yu [25] present the summation in a concise manner.

Again, a precise estimate of the FLOPs required to evaluate the Ewald summation is dependent on the FFT method. As for the extent of the summation, MacIsaac [26] states that a summation over 5 replicated systems seems to give adequate convergence. If we double this to 10 replicated systems we still have a four order of magnitude savings over the explicit summation. More work has to be done at each summation step, but a runtime totalling $1/50^{\text{th}}$ that of the above is a very conservative estimate. Generating the interaction matrices using Ewald Summation can be accomplished in seconds.

While this method is clearly faster than the previous, it is more complex both conceptually and in its implementation. There is a definite saving in time, but since this is a one time cost for all simulations at a given system size, this saving is negligible. Also, this method

is mathematically challenging and presents obstacles when attempting to simulate lattice structures with unit cells which are not square. Triangular lattices, for example, can be simulated using explicit precomputed interaction matrices with only a trivial modification to the method. For ease of use and flexibility, we have decided to use the Explicit Summation method in our calculation of the interaction matrices.

3.3 Langevin Field

The calculation of the Langevin Field involves one of the more important aspects of modern computer science, namely, pseudo-random number generation. Poor quality pseudo-random number generators (PRNG) can lead to systematic errors in simulations[11]. These errors can manifest themselves as inappropriate specific heat for a given noise value or dynamics with discernible cycles induced by the random values.

The field of cryptology has provided a host of new tools to use as sources of pseudo-random number generators. These tools supply sequences of random numbers of a quality much higher than is traditionally needed for physical simulations. The well known RANLUX PRNG provides various luxury levels as tradeoffs between throughput and quality. While RANLUX is a standard in physical simulations, we opted to expore alternatives. The ISAAC PRNG by Jenkins[21] has been optimized for 64 bit architectures and allows us to select two 32 bit double words for each generated 64 bit quad word. The ISAAC generator was designed to be a cryptographically secure pseudo-random number generator (CSPRNG). CSPRNGs generate streams of bits with the feature that given the first k bits of the stream, there is no polynomial-time method which can guess the $k+1^{\text{th}}$ bit in the

stream with greater than 50% accuracy. This feature is not a requirement of a physical simulation but since ISAAC outperforms RANLUX, in terms of speed, randomness and cycle length[23], we decided to use the ISAAC PRNG.

We generated the Langevin field by picking a random point on a sphere for a direction as defined by the second method presented by Marsaglia in [28]. This requires 3 independent standard normal variates which are each used as Cartesian coordinates to select direction and base magnitude for the Langevin field at each site. As ISAAC and most other standard RNGs generate uniform random values, an efficient method to transform the data into Gaussian random values needed to be implemented. We decided to follow the algorithm presented by Box and Muller in [3] as defined in Algorithm 1 which requires two random uniform values which ISAAC provides.

Algorithm 1 Box-Muller Random Normal Deviates

Require: rand() generates a uniform random variable on the range [-1, 1]

1: **repeat**
2: $x_1 \Leftarrow$ rand()
3: $x_2 \Leftarrow$ rand()
4: $w \Leftarrow x_1^2 + x_2^2$
5: **until** w less than 1
6: $t \Leftarrow \sqrt{-2\ln(w)}/w$
7: $n_1 \Leftarrow x_1 * t$
8: $n_2 \Leftarrow x_2 * t$

Line 5 of the algorithm imposes the restriction that the point $\{x_1, x_2\}$ must lie inside the unit circle, if this is not the case a new pair of uniform random values are selected

and retested. This condition does not ensure a constant mapping of input to output since approximately $\frac{4-\pi}{4}$ of the input data is discarded. This is a feature we wish to avoid. Our solution is to, rather than selecting a new pair of random values, transform the existing values. We apply Marsaglia's Xorshift RNG[29] to any pair that fall outside the unit circle and retest. This RNG is a simple generator involving only bit shifts and xors and so is very fast. The quality of this is not an issue as we are shuffling bits in a manner that has a period of the entire space with a high quality random seed provided by ISAAC. This shuffle is listed in Algorithm 2.

Algorithm 2 Marsaglia Xorshift RNG

Require: y is a bitstring of length 64

1: $y \Leftarrow (y \text{ left shift } a) \text{ xor } (y \text{ right shift } b) \text{ xor } (y \text{ right shift } c)$

The methods presented in this section allow us to generate and transform a set of random values of a uniform distribution into a set from a Gaussian distribution of a predictable size. Furthermore, we do not waste the calculations of a significant fraction of the uniform random data which would traditionally be discarded. The resulting values are used to calculate the Langevin field given by Equation 2.39. With the calculation of the thermal field, we now have the last component required to build $\vec{H}^{\text{eff}}$.

3.4 Calculating Spin Evolution

Having described how we calculate $\vec{H}^{\text{eff}}$ for a given spin configuration we examine how we can incorporate it into an efficient integration scheme that will allow us to compute solutions to the LLG equation. We begin by describing how we may accurately calculate

```
tcoef = StepSize*(1-0.5*relstep);
t1coef= StepSize*0.5*relstep;
for(i=0;i<Nx;i++) for(k=0;k<Nz;k++) {
  vtemp.FastAdd(tcoef,torque[i][k],t1coef,torque1[i][k]);
  m1[i][k].FastAdd(vtemp,1.-vtemp.MagSq()/2.,m[i][k]);
  // Includes correction due to restriction that |m1[i][k]| \in S^2
  m1[i][k].Scale(1);
}
```

Figure 3.1: OOMMF Spin update section (grid.cc, lines 1917-1924)

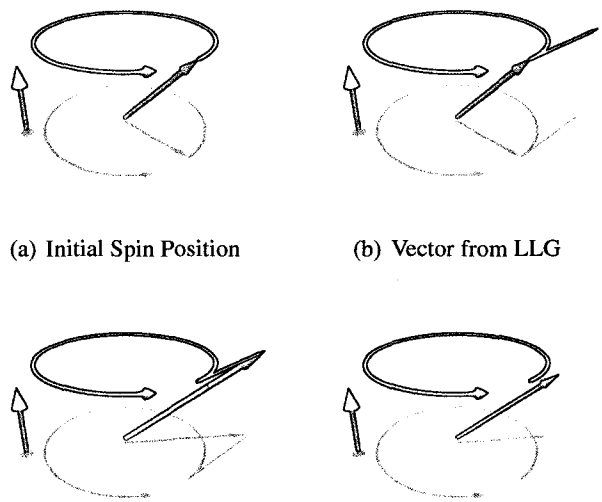
the rotation of the spins for a given set of fields of a finite but small time interval Δt. We then discuss various integration schemes and describe the adaptive Euler method that seems well suited to the integration of the LLG equation.

A technique that is widely used to calculate the rotation in the LLG equation is to express the spin vectors in terms of their Cartesian components and to calculate $\Delta\sigma_x$, $\Delta\sigma_y$ and $\Delta\sigma_z$ separately. This does not however preserve the length of the spin vector and requires that at each step we renormalize the spin vector to it's original length.

The software package *Object Oriented MicroMagnetic Framework* (OOMMF) [18] uses this method in their `grid.cc` as shown in figure 3.1. In this code, the variable `m1` is a 2 dimensional array of type `ThreeVector` which describes the spin's orientation in Cartesian coordinates, `torque` and `torque1` are 2 dimensional arrays of type `ThreeVector` which describes the calculated torque as defined by the LLG equation.

Even with the renormalization of the spin vector after the rotation, this method of calculating an infinitesimal rotation can introduce systematic errors. To illustrate this we consider the trivial case of the vector of a single spin aligned out of plane subject to a constant applied field aligned perpendicular to the plane. In the case of zero damping the spin should trace a circle as it precesses about the field.

Figure 3.2 shows the result of using this method under the conditions described. In these



(a) Initial Spin Position (b) Vector from LLG

(c) New Spin Position (d) Scaled Spin Position

Figure 3.2: This illustrates how the conventional update fails for a spin precessing in a field. The final position no longer traces the expected circular path.

figures the small vertical arrow represents the direction of the applied field, the circular arrow is the expected precessional path and the canted arrow is the magnetic spin. The length of the vector computed via LLG is exaggerated to better illustrate the problem. It is clear to see that the new spin position after rescaling no longer traces the precessional path

but has rotated toward the plane. Using this method in any simulation with zero damping will result in spins orienting themselves orthogonally to the applied field. Rotating toward or away from an applied field violates the conservation of energy demonstrated in Equation 2.32 for a spin precessing in a constant applied field.

OOMMF does not allow zero damping simulations but we can examine systems with very low damping and demonstrate the flaw inherent in this method. For small damping ($\alpha = 10^{-5}$) energy is drained from the system as it relaxes which is clearly presented in Figure 3.3a. If we decrease them damping even further ($\alpha = 10^{-13}$), we see that the energy of the system, plotted in Figure 3.3b, increases with increasing time, despite the fact that the simulation is run at zero temperature and at finite, small damping.

The approach that we have therefore chosen to calculate $\vec{\sigma}_{t+\Delta t}$ from $\vec{\sigma}_t$ for a given effective field $\vec{H}_t^{\text{eff}}$ is based on the result that, provided Δt is sufficiently small, we treat $\vec{H}_t^{\text{eff}}$ as constant, then the vector $\vec{\sigma}_t$ will simply precess about an axis $\hat{n}$ with a constant angular frequency ω, where $\hat{n}$ and ω are given by

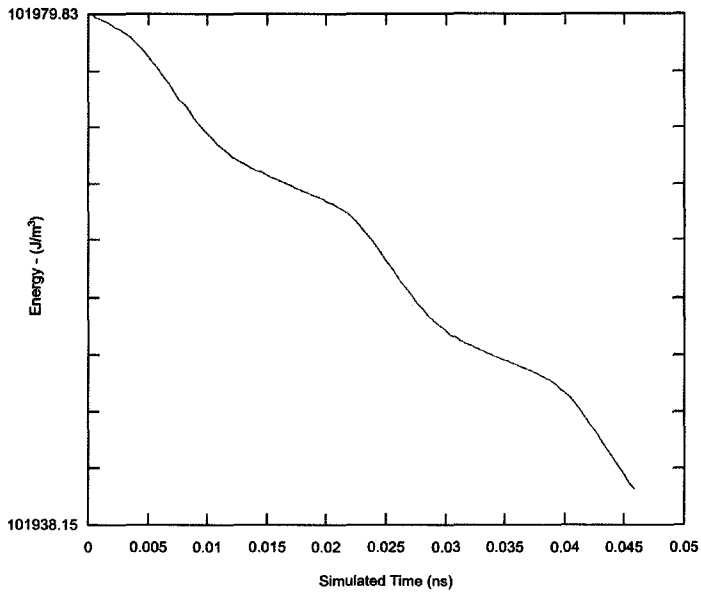
$$\hat{n} = \frac{\vec{H}_t^{\text{eff}} + \alpha \vec{H}_t^{\text{eff}} \times \vec{\sigma}_t}{|\vec{H}_t^{\text{eff}} + \alpha \vec{H}_t^{\text{eff}} \times \vec{\sigma}_t|} \tag{3.14}$$

and

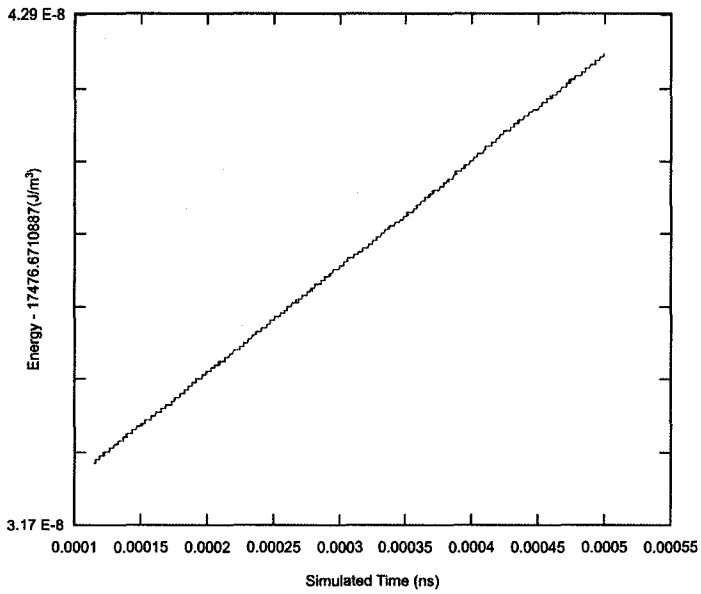
$$\omega = |\vec{H}_t^{\text{eff}} \times \vec{\sigma}_t|. \tag{3.15}$$

We can therefore write

$$\vec{\sigma}_{t+\Delta t} = \bar{R}(\hat{n}_t, \omega_t \Delta t) \cdot \vec{\sigma}_t, \tag{3.16}$$



(a) Damping = 10^{-5}



(b) Damping = 10^{-13}

Figure 3.3: Energy results from OOMMF for two values of damping

where the rotation matrix $\bar{R}(\hat{n},\theta)$ is given by

$$\begin{pmatrix} (1-\cos(\theta))n_x n_x+\cos(\theta) & (1-\cos(\theta))n_x n_y-\sin(\theta)n_z & (1-\cos(\theta))n_x n_z+\sin(\theta)n_y \\ (1-\cos(\theta))n_x n_y+\sin(\theta)n_z & (1-\cos(\theta))n_y n_y+\cos(\theta) & (1-\cos(\theta))n_y n_z-\sin(\theta)n_x \\ (1-\cos(\theta))n_x n_z-\sin(\theta)n_y & (1-\cos(\theta))n_y n_z+\sin(\theta)n_x & (1-\cos(\theta))n_z n_z+\cos(\theta) \end{pmatrix}. \tag{3.17}$$

Spin rotations using the method described by Equation 3.16 are easy to conceptualize but a more efficient scheme may be implemented via Quaternions. Quaternions are 4 dimensional complex numbers which are capable of describing rotations in 3 dimensional space. A quaternion $\mathbf{Q}$ can be described by the tuple

$$\mathbf{Q} = \{w, \vec{s}\}, \tag{3.18}$$

where $\vec{s}$ is the 3 component vector $\{i, j, k\}$. The operations we require to implement the 3 space rotation are the multiply and conjugate. Under quaternion algebra, multiplication is non-co mutative and is defined as

$$\mathbf{Q}_1 = \{w_1, \vec{s}_1\} \qquad \mathbf{Q}_2 = \{w_2, \vec{s}_2\}$$

$$\mathbf{Q}_1\mathbf{Q}_2 = \{w_1 w_2 - \vec{s}_1 \cdot \vec{s}_2, w_1\vec{s}_2 + w_2\vec{s}_1 + \vec{s}_1 \times \vec{s}_2\}, \tag{3.19}$$

or, more explicitly

$$\begin{aligned} \mathbf{Q}_1\mathbf{Q}_2 = \{&w_1 w_2 - i_1 i_2 - j_1 j_2 - k_1 k_2, \;\; w_1 i_2 + i_1 w_2 + j_1 k_2 - k_1 j_2, \\ &w_1 j_2 + j_1 w_2 + k_1 i_2 - i_1 k_2, \;\; w_1 k_2 + k_1 w_2 + i_1 j_2 - j_1 i_2\}. \end{aligned} \tag{3.20}$$

The conjugate of a quaternion is defined as

$$\mathbf{Q}^* = \{w, -\vec{s}\}. \tag{3.21}$$

These operations allow us to write the rotation of a vector $\vec{v}$ about a unit vector $\vec{z}$ by θ radians as

$$\left\{w, \vec{v}'\right\} = \left\{\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right)\vec{z}\right\}\left\{0, \vec{v}\right\}\left\{\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right)\vec{z}\right\}^* \tag{3.22}$$

which is clearer when written in the form where vectors are implicitly cast to quaternions

$$\vec{v}' = \mathbf{Z}\vec{v}\mathbf{Z}^*. \tag{3.23}$$

These forms of the rotation show that not all of the computed right hand side of Equation 3.22 is required for the rotated $\vec{v}$. Namely, the w component of the rotated vector is not used. Making use of this information allows us to truncate the final quaternion multiplication on the right hand side and save four floating point multiplies and three subtractions.

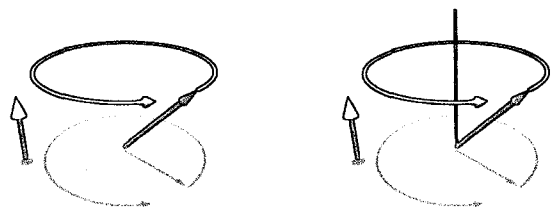
We can now rewrite Equation 3.16 in terms of quaternions as

$$\mathbf{Z}_t = \left\{\cos\left(\frac{\omega_t \hat{n}_t \Delta t}{2}\right), \sin\left(\frac{\omega_t \hat{n}_t \Delta t}{2}\right)\hat{n}_t\right\} \tag{3.24}$$

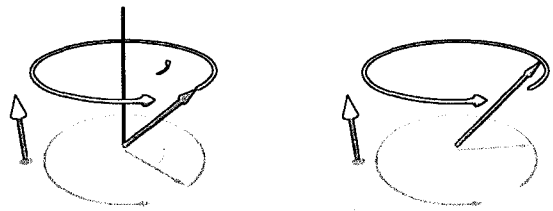
$$\vec{\sigma}_{t+\Delta t} = \mathbf{Z}_t\, \vec{\sigma}_t\, \mathbf{Z}_t^*. \tag{3.25}$$

Using quaternion rotations as a substitute for matrix based rotations saves 2 floating point multiplications per rotation and leads to more readable code as shown at the end of this chapter in Figure 3.6.

A depiction of the proper undamped precession using rotations, matrix or quaternion based, about an axis is shown in Figure 3.4. Unlike before, our resulting vector continues to trace along the expected precessional path.



(a) Initial Spin Position (b) Determine Axis of Rotation



(c) Calculate Rotation (d) Apply Rotation

Figure 3.4: This illustrates how a quaternion based update results in a spin correctly precessing in a field. The final position continues to trace the expected circular path.

3.5 Integration Through Time

In the previous section we described a method that calculates the rotation of a spin for a given effective field over some small but finite time interval. The method preserves the norm of the spin vector, to within machine precision, and is exact in the case of a constant field. In reality, of course, the effective field at each site is changing with time. The question then arises as to how we compute the change in the spin vector over a finite interval given that the fields change with time and are calculated self consistently for the spin configuration.

We have considered a number of integration procedures: the Euler Method, second

order Runge Kutta and the Burlisch-Stoer method. The straightforward Euler method has the advantage that it is readily generalized to finite temperature but the stepsize required to ensure the stability of the approach is typically very small[6]. The higher order Runge Kutta schemes (first order Runge Kutta is the Euler Method) are more robust and allow for larger time steps but are problematic at finite temperature. While Honeycutt has presented a method to implement a stochastic Runge Kutta scheme for certain types of noise[17], generalizing this method to our case was not obvious. The Burlisch-Stoer method is an adaptive order numerical integration method which was recommended to us by Olle. G. Heinonen of Seagate [16]. This method adaptively alters the order of the integrator to fulfil a user defined tolerance which, while ingenious for non-stochastic processes, merely compounds the difficulties presented by Honeycutt.

While the difficulties in generalizing to higher order schemes limited their usefulness, experience in applying these different integration techniques demonstrated there were clear advantages to using an adaptive step method. We therefore implemented a first order Euler method with an adaptive step. While this method contains many of the strengths of the Burlisch-Stoer method, it avoids the complications of higher order stochastic integration schemes.

Our adaptive step Euler method begins with the spin configuration $\{\vec{\sigma}(\vec{r},t,i)\}$ which represents all spins at lattice locations $\vec{r}$ on the lattice $\mathbf{L}$ ($\vec{r} \in \mathbf{L}$), at time t and simulation iteration i. From this set we have calculated the effective fields $\{\vec{H}^{\text{eff}}(\vec{r},t,i)\}$ and selected

a time step Δt_i. Next we calculate $\vec{\sigma}(\vec{r}, t+\Delta t_i, i)$ and $\vec{\sigma}(\vec{r}, t+\Delta t_i/2, i)$ as

$$\vec{\sigma}(\vec{r}, t+\Delta t_i, i) = \mathbf{Z}(\hat{n}, \omega\Delta t)\ \vec{\sigma}\ \mathbf{Z}^*(\hat{n}, \omega\Delta t)\Big|_{\vec{r},t,i} \tag{3.26}$$

$$\vec{\sigma}\left(\vec{r}, t+\frac{\Delta t_i}{2}, i\right) = \mathbf{Z}\left(\hat{n}, \omega\frac{\Delta t}{2}\right)\ \vec{\sigma}\ \mathbf{Z}^*\left(\hat{n}, \omega\frac{\Delta t}{2}\right)\Big|_{\vec{r},t,i}. \tag{3.27}$$

From $\{\vec{\sigma}(\vec{r}, t+\frac{\Delta t_i}{2}, i)\}$ we calculate the corresponding effective fields $\{\vec{H}^{\text{eff}}(\vec{r}, t+\frac{\Delta t_i}{2}, i)\}$. We then compute

$$\vec{\sigma}'(\vec{r}, t+\Delta t_i, i) = \mathbf{Z}\left(\hat{n}, \omega\frac{\Delta t}{2}\right)\ \vec{\sigma}\ \mathbf{Z}^*\left(\hat{n}, \omega\frac{\Delta t}{2}\right)\Big|_{\vec{r},t+\frac{\Delta t}{2},i} \tag{3.28}$$

which is a better approximation to the true value of $\vec{\sigma}(\vec{r}, t+\Delta t_i, i)$ than the value given in Equation 3.26. These two values are compared for equality to a tolerance *tol* and declared equal if

$$tol < \underset{\vec{r}}{\text{Max}}\left(|\sigma'^{\alpha}(\vec{r}, t, i) - \sigma^{\alpha}(\vec{r}, t, i)|\right) \tag{3.29}$$

for each Cartesian component α. If it is found that $\vec{\sigma}$ and $\vec{\sigma}'$ are equal by Equation 3.29 then we define the following simulation time, step size and spin configuration for the next iteration as

$$\begin{aligned} t_{i+1} &= t_i + \Delta t_i \\ \Delta t_{i+1} &= 2\Delta t_i \\ \vec{\sigma}(\vec{r}, t_{i+1}, i+1) &= \vec{\sigma}(\vec{r}, t_i + \Delta t_i, i), \end{aligned} \tag{3.30}$$

otherwise we do not advance the simulation through time and retry the step at a finer reso-

lution

$$t_{i+1} = t_i$$
$$\Delta t_{i+1} = \frac{1}{2}\Delta t_i$$
$$\vec{\sigma}(\vec{r}, t_{i+1}, i+1) = \vec{\sigma}(\vec{r}, t_i, i). \tag{3.31}$$

Schematically, this method is depicted in Figure 3.5 at the end of this chapter.

3.6 Conclusion

This chapter described a computationally efficient solution to the LLG equation that describes a magnetic thin film. Treating the dipolar interaction in an intelligent manner dramatically improves the time taken to compute the dipolar field. Our spin update method preserves spin length and energy in undamped, zero temperature systems. Finally, the adaptive step integration scheme appears to be well suited to our system dynamics.

In addition to the efficiencies described in this chapter, parallel computing gives us a way to further reduce runtime by spanning our problem across multiple computational nodes. In the next chapter we will review current techniques for parallelism, how they can be applied to our problem and determine what level of savings we can achieve in simulation runtimes.

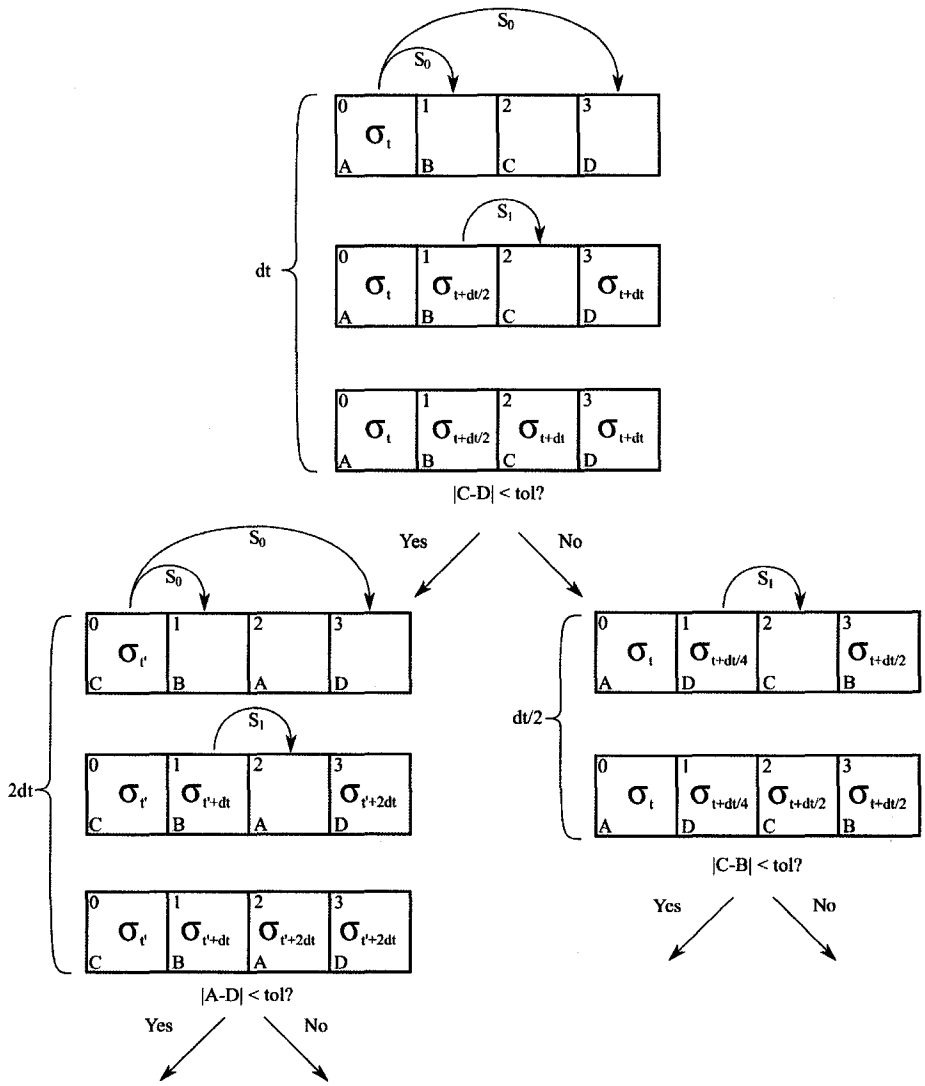


Figure 3.5: Adaptive Timestep Schematic

```
qVec.w = 0;
qVec.x = r_sx[i]; //rotate this vector
qVec.y = r_sy[i];
qVec.z = r_sz[i];

cost = cos(0.5 * rotateAmount[i] * s->dt);
sint = sin(0.5 * rotateAmount[i] * s->dt);

qRot.w = cost;
qRot.x = sint * damped_h[i*3+0]; //rotate about damped h
qRot.y = sint * damped_h[i*3+1];
qRot.z = sint * damped_h[i*3+2];

//this is the rotation: qRes = qRot qVec qRot*
qRes = qmultXYZ(qmult(qRot, qVec), qconjugate(qRot));

s->r_sxr[i] = qRes.x;
s->r_syr[i] = qRes.y;
s->r_szr[i] = qRes.z;
```

Figure 3.6: Quaternion Based Rotations (magsim.c, lines 793-811)

Chapter 4

Parallelism

In this chapter we discuss how the methods described in the previous chapter can be adapted to take advantage of shared memory multi-processor architectures through parallelism. This involves dividing the work required to evaluate the algorithm over multiple computational units. Several factors must be considered when writing parallel code including how to divide the problem among the processors based on the available hardware.

4.1 Parallel Design Patterns

There are several methods for partitioning a problem over multiple processors. We will discuss two common methods, functional and domain decomposition, and review the difficulties and benefits of using each to implement parallel code for the algorithm we used to solve the spin dynamics. We then go on to describe a hybrid parallelization scheme which dynamically combines elements of these two approaches.

4.1.1 Functional Decomposition

The first design pattern we consider is functional decomposition. As the name implies, parallelism under this scheme is based on partitioning or decomposing the problem based on function or task. This is a very natural method and is analogous to a group of students working on an assignment, in which each student completes one question and then the solutions are pooled just before the deadline. When considering this under the context of parallel programming, we use a CPU or group of CPUs each operating on an individual task while other groups are operating on separate tasks.

4.1.2 Domain Decomposition

The second design pattern is Domain Decomposition in which tasks are performed in sequence using a single group of processors. Rather than performing all tasks in parallel, as is the case of Functional Decomposition, we parallelize the evaluation of each operation. Using this scheme, each processor is performing the same set of instructions on separate portions of the data. The partitioning of the data depends on the initial format. For example, a matrix of data could be divided by rows, columns, blocks or a random pattern. The partition method is selected by minimizing the access, evaluation and write times. If a matrix is stored by rows then, for sufficiently large datasets, accessing by columns will result in numerous cache misses and eventually page faults which would generate unneeded overhead. Dividing the data into blocks, while more expensive in terms of cache misses, minimizes the perimeter around each subset of data which, depending on the situation, may result in lower total runtime than accessing by rows.

4.2 Applying Parallel Concepts

The choice of design pattern and how it is implemented requires careful consideration of the tasks, and in particular, the sequence in which tasks are to be performed as well as their interdependence.

Our algorithm conveniently divides into 6 tasks, the calculation of the 5 contributions to the effective field field defined by Equations 2.6 to 2.8 and 2.39, namely the exchange interaction, dipolar interaction, surface anisotropy, applied external field and thermal perturbations. The sixth task is applying the rotation based on the effective field defined by Equation 2.33.

Obviously the various contributions to the effective field may be evaluated independently. However, we need to complete the calculation of all the fields before we can determine the total effective field and evaluate the rotation of the spins. Thus a simple parallelization strategy would be to evaluate the effective field using functional decomposition, provided we can ensure that we do not begin evaluating the rotation until all the individual tasks are completed and the total effective field is calculated.

In order to assess how effective this parallelization strategy might be we need to determine the time taken for a single processor to complete each of these tasks. Table 4.1 shows the distribution of task timings per iteration for system sizes between 64x64 and 512x512 on a single CPU execution of the simulation, in this case one of the SGI Altix 350 processors. Table 4.2 breaks the execution time down by lattice site and Table 4.3 presents the ratio of each task as a percentage.

Task	32x32	64x64	128x128	256x256
Dipolar Interaction	3.455	20.002	105.965	507.506
Exchange Interaction	1.150	4.782	19.449	80.660
Surface Anisotropy	0.217	1.325	5.539	29.750
External Field	0.102	0.428	1.779	14.034
Apply Rotation	2.502	11.131	47.827	214.052
Total	7.426	37.668	180.558	846.004

Table 4.1: Average execution time in milliseconds (10^{-3}s) per iteration

Task	32x32	64x64	128x128	256x256
Dipolar Interaction	3.3740	4.8833	6.4676	7.7439
Exchange Interaction	1.1230	1.1675	1.1871	1.2308
Surface Anisotropy	0.2119	0.3235	0.3381	0.4539
External Field	0.0996	0.1045	0.1086	0.2141
Apply Rotation	2.4434	2.7175	2.9191	3.2662
Total	7.2520	9.1963	11.0204	12.9090

Table 4.2: Average execution time in microseconds (10^{-6}s) per iteration per lattice site

Task	32x32	64x64	128x128	256x256
Dipolar Interaction	46.53	53.10	58.69	59.99

Task	32x32	64x64	128x128	256x256
Exchange Interaction	15.49	12.70	10.77	9.53
Surface Anisotropy	2.92	3.52	3.07	3.52
External Field	0.06	0.08	0.10	0.22
Apply Rotation	33.69	29.55	26.49	25.30

Table 4.3: Ratios (percent) of task times

We can immediately see from the data that the ratio of time spent on each task is not a constant but depends on the size of the system. In the case of the dipolar interaction this increase is perhaps not too surprising since the FFT algorithm is known to scale as $O(N \log(N))$, we would expect the execution time per site to scale as $O(\log(N))$. However, since the evaluation time of the contribution to the effective field from the other 4 interactions should scale as $O(N)$, we would expect the execution time per site to be roughly constant (ie independent of N). The fact that the data in Table 4.2 manifest a more complex dependence on N is due to cache misses and page faults. The number of potential floating point operations per second remain the same, but we spend more and more time delivering data to the processor when we cannot hold it all in cache. This demonstrates that significant portion of the runtime is dependent on the detailed nature of the hardware used.

The data included in Tables 4.1-4.3 also show that the evaluation of the contribution to the effective field due to the dipolar interaction takes significantly longer than the evaluation of all the other contributions combined. This implies that the best we could do with a purely functional decomposition of the calculation would be to improve the runtime by a factor of 15% to 20%. In addition, while this could be accomplished by assigning one processor to evaluate the dipolar contribution to the effective field and another to evaluating the

remaining contributions, tasks finishing asynchronously would degrade the performance as it would result in processors sitting idle while there is work to be done. Therefore while functional decomposition of the calculation of the effective field does offer potential gains, we would need to divide the problem differently depending on number of processors, cache and memory access times and lattice sizes. Also considering that this division of processors is discrete, we would certainly have processors sitting idle for a significant time.

This line of reasoning, based on the data presented in Table 4.2 suggests that an appropriate parallelization strategy would be based on domain decomposition. While this approach requires more synchronization, each synchronization event is far less expensive than synchronizations in a functional decomposition scheme. The reason for this cost savings is that we can achieve a much more balanced division of labour. For example, considering the surface anisotropy, we now divide the total number of lattice sites by the number of processors. Even with small systems such as 32x32, the maximum work difference between any two processors in a 7 CPU group is less than 1%. Computing an effective division is also trivial for most situations, simply dividing the task by the number of CPUs will maximize efficiency. While we can implement efficient parallel code almost trivially for many aspects of computing the contributions to the effective field, the dipolar interaction poses a problem due to the nature of the FFT calculation as described in Section 3.1.2. In order to implement a parallel solution that can efficiently evaluate the dipolar effective field requires a more careful analysis. In the next sections we examine how the FFT algorithm may be efficiently parallelized.

4.3 Fourier Transform Revisited

With the exception of the dipolar interaction the various contributions to the effective field (exchange, anisotropy, etc.) are trivial to parallelize using the domain decomposition paradigm which requires the evaluation of N independent quantities. It is far less obvious, however, to parallelize the evaluation of the dipolar contribution. For this reason and the fact, as shown in Table 4.3, that the evaluation of the dipolar contribution to the effective field represents the most computationally intensive part of the calculation of the effective field, considerable effort and thought has gone into evaluating the dipolar field and how it can be efficiently parallelized.

In this section we therefore revisit the FFT algorithm to identify how we may best exploit the advantages of parallelism. We begin by stating the Discrete Fourier Transform as

$$F_k = \sum_{j=0}^{N-1} W^{jk} f_j, \tag{4.1}$$

with

$$W = e^{2\pi i/N}. \tag{4.2}$$

This states that each element in frequency space F_k is a linear combination of all elements in real space f_j. This formulation of the Fourier Transform is clearly $O(N^2)$, the key to reducing the order to $O(N \log(N))$ lies in rewriting Equation 4.1 as Danielson and Lanczos presented in [7]

$$F_k = F_k^e + W^k F_k^o \tag{4.3}$$

which they elegantly prove with

$$
\begin{aligned}
F_k &= \sum_{j=0}^{N-1} e^{2\pi i jk/N} f_j \\
&= \sum_{j=0}^{N/2-1} e^{2\pi i(2j)k/(N/2)} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi i(2j+1)k/(N/2)} f_{2j+1} \\
&= \sum_{j=0}^{N/2-1} e^{2\pi i(2j)k/(N/2)} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi i(2j)k/(N/2)} e^{2\pi i(1)k/(N/2)} f_{2j+1} \\
&= \sum_{j=0}^{N/2-1} e^{2\pi i(2j)k/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi i(2j)k/(N/2)} f_{2j+1} \\
&= F_k^e + W^k F_k^o. \qquad (4.4)
\end{aligned}
$$

This states that the k^{th} element of the Fourier Transform can be computed as a linear combination of the k^{th} components of the Fourier Transform of the even elements of F and the Fourier Transform of the odd elements. While F^e and F^o are half the length of F, they represent a periodic data set and so k, which indexes over the full length of F, can still be used to index into F^e and F^o. Equation 4.3 is used recursively to create a decimation in time algorithm breaking the problem into a pair of smaller and smaller problems. The recursion stops when the data to be operated on is of length one, in which case the Fourier Transform is simply the identity operator. The calculation of each F_k in isolation is an order N operation, but when we consider the entire set of F, we can reuse much of the computations, for instance, the 5^{th} element of F^e is the same as the $N/2+5^{th}$. This is the key to reducing the time complexity to $O(N \log(N))$.

Let us consider an FFT of 8 datapoints, we can explicitly write the decimation as

$$\begin{aligned} F_k &= F_k^e + W_{\frac{N}{2}}^k F_k^o \\ &= F_k^{ee} + W_{\frac{N}{4}}^k F_k^{eo} + W_{\frac{N}{2}}^k (F_k^{oe} + W_{\frac{N}{4}}^k F_k^{oo}) \\ &= F_k^{eee} + W_{\frac{N}{8}}^k F_k^{eeo} + W_{\frac{N}{4}}^k (F_k^{eoe} + W_{\frac{N}{8}}^k F^{eoo}) + W_{\frac{N}{2}}^k (F_k^{oee} + W_{\frac{N}{8}}^k F_k^{oeo} + W_{\frac{N}{4}}^k (F_k^{ooe} + W_{\frac{N}{8}}^k F^{ooo})) \end{aligned} \tag{4.5}$$

where

$$W_a^k = e^{2\pi ik/a}. \tag{4.6}$$

Which shows that we can build the final FFT starting from FFTs of individual elements. Schematically, this operation is depicted in Figure 4.3 where we can see the F^{ddd} data is constructed using identity operators on the Bit Reversal Permutation of the input data. A linear combination of this data is used to calculate the F^{dd} data which is in turn linearly combined to form the F^d and finally the complete Fourier Transformed data F_k.

Data must be processed as depicted in Figure 4.3 such that all elements of the second column are computed before they can be combined to evaluate the elements of the third column. However, there is no restriction on what order we calculate each datapoint in a given column. This implies that the task of evaluating each column may be easily divided up between several processors. This realization is central to constructing a parallel implementation of the FFT.

Our parallel implementation of the algorithm uses a domain decomposition of each column in Figure 4.3. However, since the sequence in which the successive columns is evaluated is important, it is essential that some synchronization must occur. This synchronization is realized by use of a parallel barrier which is a programing construct that will

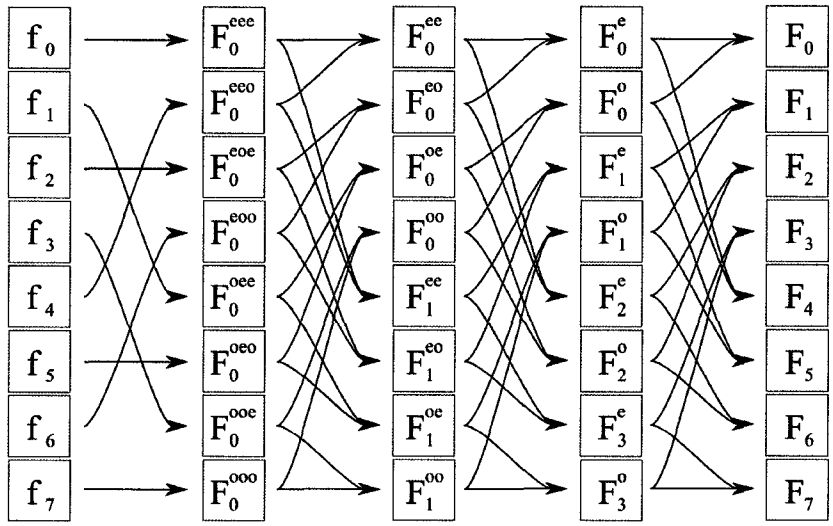


Figure 4.1: Schematic of an FFT of 8 data points

not allow a process of a group to proceed until the entire group has reached the barrier. In this case, the barrier will not allow any of the processes that evaluate the elements in column 3 to begin until all processes used to evaluate column 2 have completed. In the next chapter we will discuss how a barrier may be implemented in detail, but for now it is sufficient to know that it ensures the correct sequence of operations when working in a parallel environment.

We can generalize this implementation of the FFT as a tool to perform the two dimensional FFT. A 2D FFT is the transformation of the rows followed by the transformation of the resulting columns. Figure 4.3 demonstrates this operation. There are some subtleties in how we select the optimal procedure when we parallelize the 2D FFT that depend on how we decompose the matrix. It can be shown[1] that the optimal procedure is to perform a

[1]The coarse grained approach for each 2D FFT employs a total of 3 barriers to properly synchronize

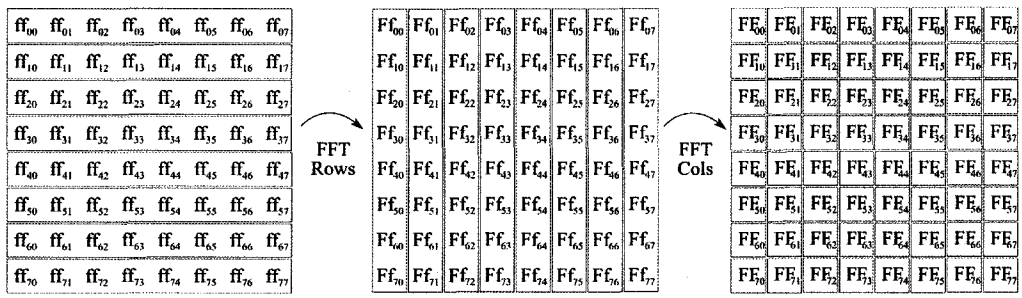


Figure 4.2: Schematic of an FFT of 8x8 data points

domain decomposition on the set of rows and then on the set of columns.

4.4 Evaluating Groups of FFTs in Parallel

Up to this point we have considered how we may parallelize the FFT of a two dimensional matrix. However, the problem we seek to solve requires that we calculate 3L FFTs, corresponding to the three Cartesian components for each of the L layers. How to distribute the evaluation of the 3L FFTs over a fixed number of processors is a complex problem and requires that we consider the nature of the hardware we are using to perform the simulation.

There are two obvious strategies that we may use to evaluate the FFTs. The first is referred to as fine grained parallelism and evaluates each of the 3L FFTs in sequence, using all of the processors for each transform. The second strategy is referred to as coarse grained parallelism and simply distributes the evaluation of the 3L FFTs between 3L processors, with each processor performing an FFT. In this section, we examine the performance of

the process. A properly synchronized fine grained approach must use $3n(\log(n)+1)$ barriers. Balance in workload and access times are the same in each case but the extra time used for synchronization in the fine grain implementation will result in consistently lower performance than the coarse grain.

Number of CPUs	Execution Time (milliseconds)
1	7.930
2	4.269
3	4.651
4	3.704
5	3.726
6	3.356
7	3.092
8	2.971

Table 4.4: Parallel Execution times for 256x256 FFTs on the SGI Altix 350

each of these strategies and demonstrate that there exists an hybrid parallelization scheme that is superior to both.

4.4.1 Fine Grained Parallelism

In order to examine fine grain parallelism we consider the particular case of a single layer and we assume that we have 7 CPUs to work with. The parallel execution time for each FFT is given in Table 4.4. This data was generated on the SGI Altix 350 for a system size of 256x256 using up to 8 CPUs, for this specific case we will only consider the data up to 7 CPUs. Evaluating the 3 FFTs sequentially, using 7 CPUs per evaluation, as depicted schematically in Figure 4.3 results in runtimes of 9.276 ms, which is the execution time for the 7 CPU case times 3. This is clearly a savings over the single processor method which would have taken 23.79ms (3 x 7.930ms) to evaluate the same set of FFTs.

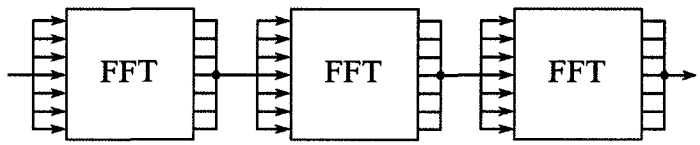


Figure 4.3: Fine Grain Parallel FFT Execution

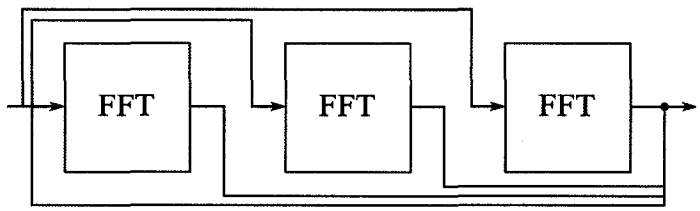


Figure 4.4: Coarse Grain Parallel FFT Execution

4.4.2 Coarse Grained Parallelism

In the case of coarse grained parallelism, we distribute each of the 3 FFTs to a separate CPU as depicted in Figure 4.4. Using the execution times as shown in Table 4.4, the total execution time from the 3 FFTs using this method is 7.93 ms. Thus we see that in this case, coarse grained implementation is 15% faster than the fine grained implementation. However, it is clear that this method is not optimal as over half of the processors are idle during the calculation.

4.4.3 Hybrid Parallelism

We have looked at fine grained parallelism which calculates results faster than a single processor solution and a coarse grained parallel approach which gives us additional savings in time but there seems to be better method. This method is an hybrid approach. In this section we will describe a way to optimally execute a set of FFTs. Each FFT may be indi-

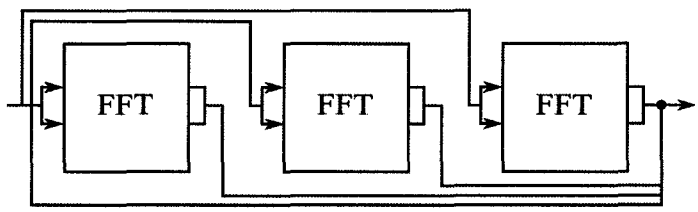


Figure 4.5: Hybrid Parallel FFT Execution

vidually calculated in a parallel fashion and likewise, the set of all FFTs has the opportunity to be executed in parallel environment. We will implement parallelism at both the fine and coarse scale of the problem. We will first present an optimal hybrid solution to the example we have been using in this section and then discuss how we generalize this method to an arbitrary number of FFTs, CPUs and execution times.

The optimal hybrid solution to the execution of 3 FFTs using 7 CPUs with execution times defined in Table 4.4 is represented in Figure 4.5. This solution uses two processors per FFT and one idle processor which forms 4 computational groups. The total execution time for this method is 4.269ms[2], nearly half the time of the coarse grained method and approximately a fifth of the single processor time. It should be noted that this is also the exact execution time for a 6 CPU solution.

The idle processor is the result of the hardware topology of the system, this becomes more clear when Table 4.4 is plotted as Figure 4.6. We can see that there is an increase in execution time when moving from 2 to 3 CPUs. This is due to the hardware of the SGI Altix 350, which is divided into computational nodes, each of which have two processors. The jump from 4 to 5 CPUs also shows the underlying architecture as there is again an increase

[2]4.269ms = max(4.269ms, 4.269ms, 4.269ms, 0ms). 4.269ms is the execution time for a single 256x256 FFT using 2 CPUs.

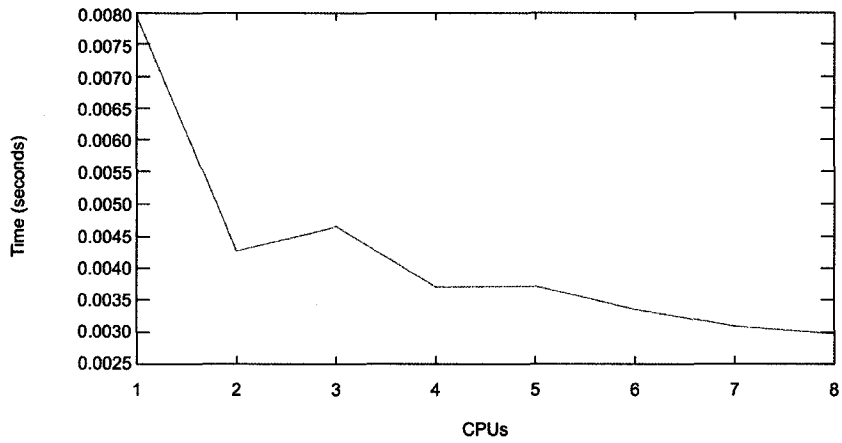


Figure 4.6: Heterogeneous topology visible in parallel FFT execution times from data in Table 4.4

in runtime. Communication time between CPUs on a single node has been experimentally determined to be at least three times faster than communication between CPUs on separate nodes.

The hybrid scheme is an attempt to solve the problem of how to optimize the distribution of $3L$ FFTs across N processors. This is a difficult undertaking for $N > 3L$ as the solution depends on the size of each layer, the number of processors available, their topology and the current computational load on the machine. Any of these variables are dynamic and a given optimization at one particular moment may not be optimal at a later time. We approach the problem in the following way.

1. Before starting the calculation, we determine the time taken for 1 to n CPUs to perform a single FFT in parallel. This we refer to each FFT as a "task" and we denote it's time as $t(n)$. This is equivalent to generating the data given in Table 4.4.

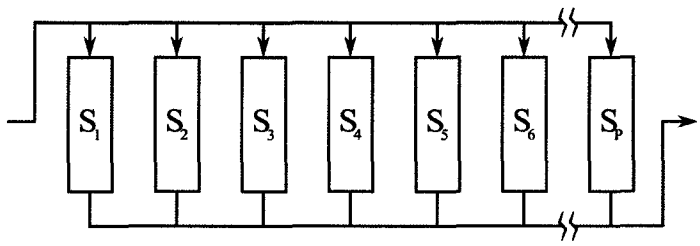


Figure 4.7: Parallel Pipelines

2. We consider dividing the available N processors into P pipes (as in Figure 4.7) where n_i denotes the number of processors assigned to evaluate the set of tasks in the i^{th} pipe, $i \in \{1 \ldots P\}$.

3. For each distribution $\{n_i\}$, we consider the different ways we can allocate the $3L$ tasks across the P pipes. Denoting T_i the number of tasks assigned to the i^{th} pipe, each pipe can be described as $S_i = \{n_i, T_i\}$ and the set of pipes $\mathbf{S} = \{S_i\}$.

4. We then define the time taken to complete pipe i as

$$t(S_i) = t(n_i)T_i, \tag{4.7}$$

and

$$t(\mathbf{S}) = \max(t(S_i)), \tag{4.8}$$

which represents the total time for all the pipes to complete all the tasks assigned to them using the allocated processors, with tasks in each pipe being performed sequentially while the pipes are executed in parallel.

5. The optimal number of pipes and distribution of tasks and processors is then deter-

mined by the set $\{S_i\}$ that minimizes $t(\mathbf{S})$ subject to the constraint

$$\sum_{i=1}^{P} n_i \leq N$$
$$\sum_{i=1}^{P} T_i = 3L. \tag{4.9}$$

The fine grain and coarse grained schemes, discussed earlier, are obviously included in the set of possible solutions. The fine grained scheme coresponds to the set

$$\mathbf{S} = \{\{n\},\{T\}\} = \{\{N,3L\}\}, \tag{4.10}$$

while the coarse grained scheme coresponds to the solution

$$\mathbf{S} = \{\{1,1\}\{1,1\},\ldots,\{1,1\}\}$$
$$|\mathbf{S}| = 3L. \tag{4.11}$$

In this way, the hybrid scheme dynamically distributes the $3L$ tasks across the N processors at the start of each simulation to minimize the time taken to evaluate the dipolar contribution to the effective field for a given number of sites, layers and computational environment.

The example considered earlier with a 256x256 system with $L = 1$ and seven available processors, using the data given in Table 4.4 yields the solution

$$P = 3$$
$$T_i = 2$$
$$n_i = 2, \tag{4.12}$$

which both minimizes the total time as defined by Equation 4.8 and satisfies the restrictions imposed by Equation 4.9.

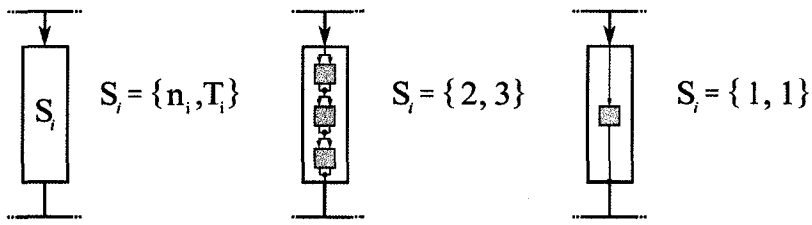


Figure 4.8: Pipeline Detail

4.4.4 Remarks

In this section we have shown progressively better methods for evaluating the set of FFTs in a parallel environment ending with an optimal solution. This should provide a glimpse of the complications present when implementing parallel code. We have been able to reduce our FFT execution time to nearly $1/5^{th}$ when using 6 CPUs. This statement immediately begs the question, why not $1/6^{th}$ the runtime? The answer is that there is overhead when working in a parallel environment. We have stated that finite communication time exists between CPUs which is a source of overhead but there is another element that we have simply called the barrier function. In the next chapter we will return to the barrier and examine it in more detail.

4.5 Speedup

We now examine the success of the hybrid parallelization scheme from a practical perspective. Specifically, how does the method perform in the type of computing environments that we have access to. In our discussion the basic metric is speedup, which is the ratio between the time taken for a job to run on a single CPU against the time taken to run on

multiple CPUs. Thus if, for example, a job takes 1 hour to run on a single CPU and only 1/2 an hour on a 3 CPU system, then we would have a speedup of 2. We refer to a linear speedup when the speedup is equal to the number of processors, and superlinear when it exceeds the number of processors.

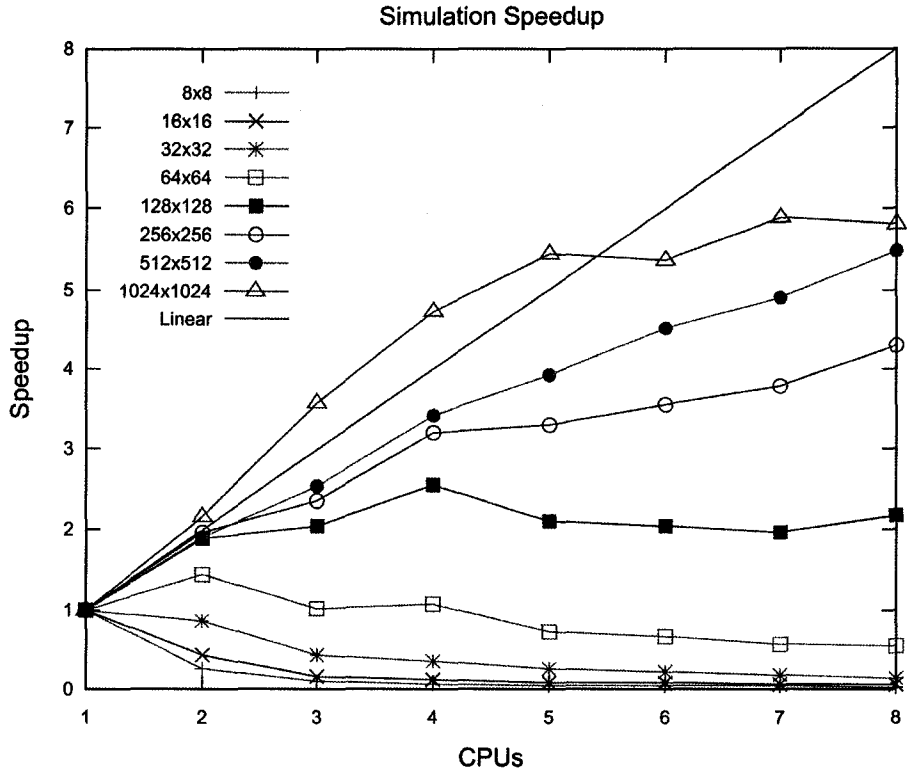


Figure 4.9: Simulation Speedup, Fine Grain FFT method

Figure 4.5 shows the speedup we achieve on the 16 processor SGI Altix 350 using the fine grain parallelism of the set of FFTs. The 64x64 simulation generates marginal speedup with 2 CPUs and after that the parallel overhead dominates and makes each additional processor more and more inefficient. At system sizes of 128x128 and above we begin

to see reasonable speedup. In the case of the 1024x1024 systems we achieve superlinear speedup between 2 and 5 CPUs. Again the limitations of the computational environment can be seen in this graph. For medium to large systems (128x128 to 256x256), we get performance boosts when moving from 1 to 2 CPUs and again from 3 to 4 but the slope decreases when moving from 2 to 3 and 4 to 5. This is due to added communication time between nodes on the Altix.

The gains from the hybrid method of group FFT execution is clear when we look at Figure 4.5, especially when considering the smaller sized systems. The 64x64 simulation, for example, has a speedup of approximately 0.5 when running on 8 processors under the fine grained parallel FFT execution and slightly more than 3 when considering a hybrid approach.

Our code and libraries are designed to migrate our simulation onto processors with the highest possibly locality, given the state of the computer. A method to improve this situation would be to have queueing and scheduling software be aware of the topology of the computer but a rewrite of those subsystems is well outside the scope of this thesis.

In order to identify the bottlenecks that limit the speedup, Figures 4.11 to 4.14 depict speedup by task for systems ranging from 64x64 to 512x512. These speedup graphs provide unique insight into the inner workings of our simulation. These data clearly show that the principle factor limiting the speedup is the dipolar interaction. The dipolar interactions, in these cases, were calculated using the hybrid FFT execution which yield best-case parallelism.

Another item to note is the superlinear speedup of some of our $O(N)$ tasks, this is because of the domain decomposition which reduces cache misses per process. Larger

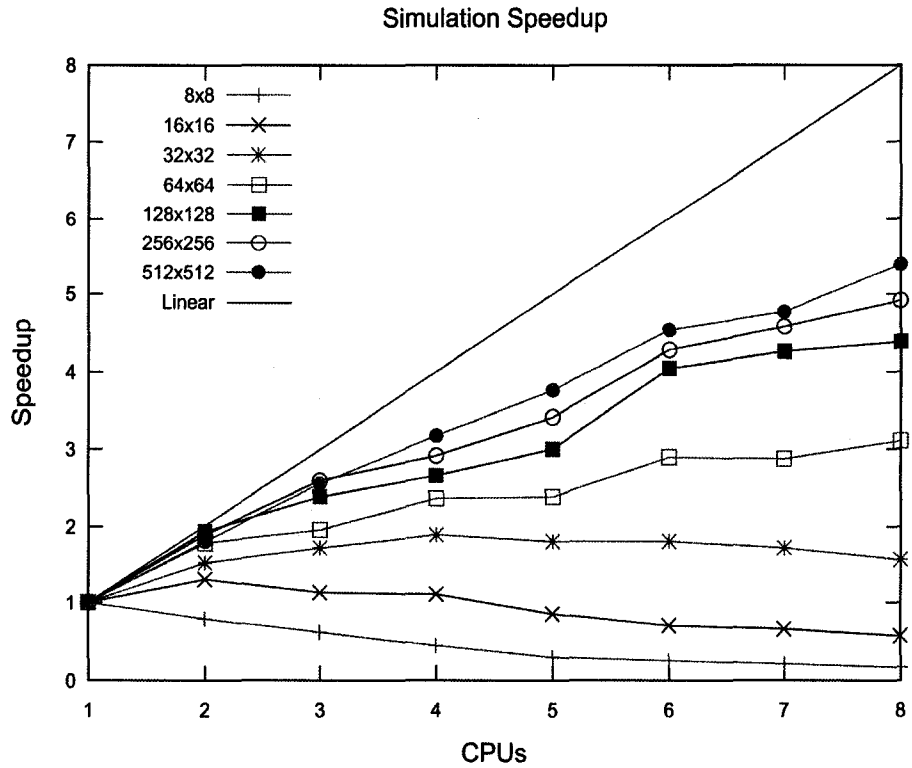


Figure 4.10: Simulation Speedup, Hybrid FFT method

systems cannot be completely fit into cache and so a single CPU execution of that suffers misses and page faults. When divided into smaller segments, fewer pauses are required as the simulation waits for memory to move to the CPU.

4.6 Conclusion

This chapter described the techniques we have devised to first achieve parallelism and then maximize that parallel performance. Our primary focus was the group execution of

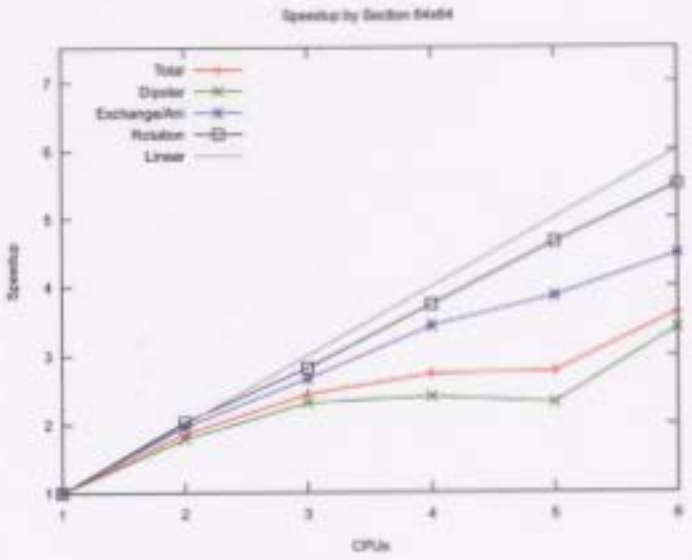


Figure 4.11: Speedup by Section, 64x64

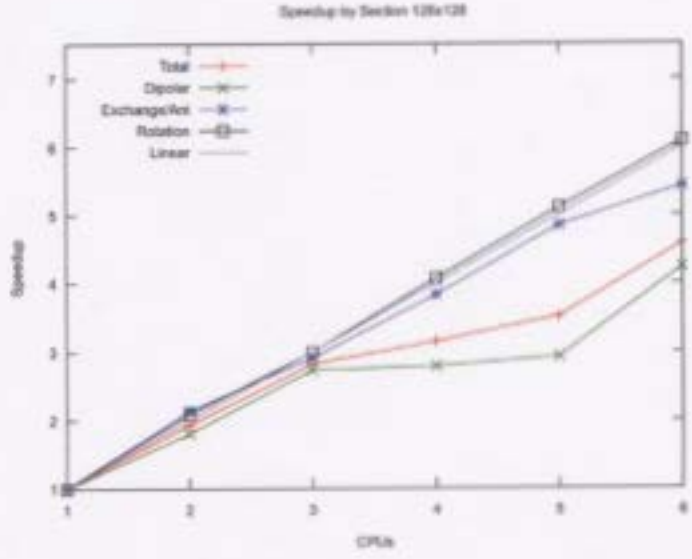


Figure 4.12: Speedup by Section, 128x128

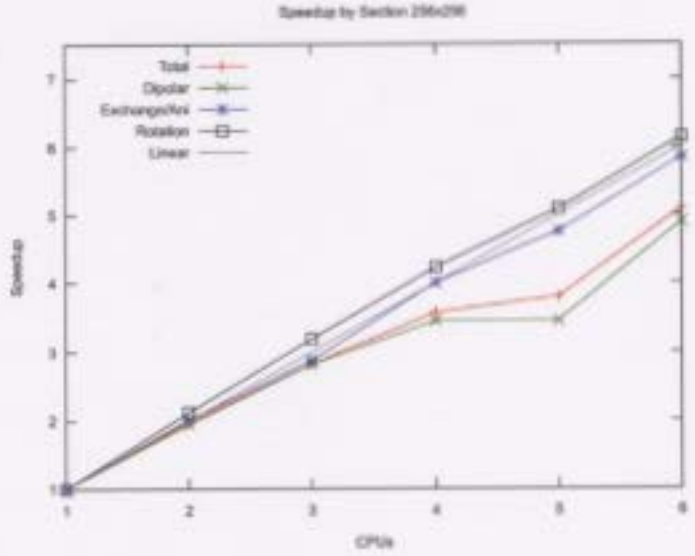


Figure 4.13: Speedup by Section, 256x256

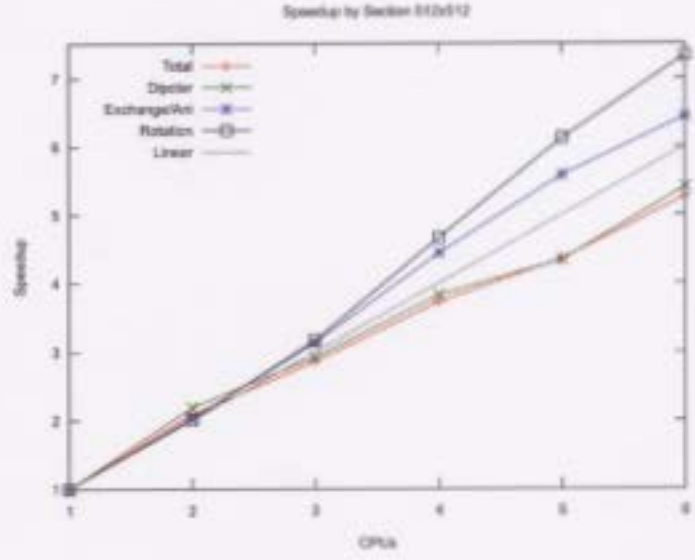


Figure 4.14: Speedup by Section, 512x512

the Fourier Transforms for the dipolar interaction using our hybrid approach but we also parallelized the remaining interactions using domain decomposition and examined their speedup. While we have implemented and analyzed these solutions, we have thus far only described them at the conceptual level. The next chapter will examine the low level design choices required to build a high performance parallel library.

Chapter 5

Implementing a Parallel Solution

We have discussed the algorithmic level choices for creating a parallel simulation and the results obtained from them, but we have not defined the code and libraries used to implement these algorithms. Before coding with a given library, we needed to select the one which best suits our needs. Traditionally this involved either selecting MPI for a cluster environment or OpenMP for an SMP environment but we decided to engage in a more rigorous selection method. We begin this section by examining what we require from a parallel library and then use these requirements to determine which would best suit our needs. Following this, we will examine the inner workings of our choice.

5.1 Requirements of a Parallel Library

A parallel library consists of a set of routines that are responsible for performing all of the low level initialization and communication between individual processors. In this section we consider the features we have identified as essential for the parallelization methods

discussed in the previous section.

5.1.1 Low Latency

Communication time between processors and memory, and general overhead incurred from the parallel programming paradigm must be kept to a minimum. The time associated with a communication action is derived from the path that the information takes through both hardware and software and the amount of information being transferred. We can control the amount of the data transferred but the path that it takes between processes is the responsibility of the parallel library. Our algorithm, especially in the dipolar section, requires all processors to interact with large, overlapping sections of the data. A parallel library which incurs lengthy communication times would be unsuitable for our purposes.

5.1.2 Topologically Aware

Modern computers are increasingly becoming hybrid architectures. Many high-end desktop machines are of a multi-core, multi-cpu design, while high performance Symmetric Multi-Processor (SMP) machines can be multi-core, multi-cpu, multi-node configurations. Traditionally these machines have been treated as homogeneous devices and differences in communication times between CPUs has been ignored. For a wide class of problems the amount of interprocess communication is such that it can be ignored but for some classes of problems, such as the one we consider, the variation in communication times between different elements of a hybrid architecture gives rise to a rich computational topology which must be confronted and dealt with.

In a multi-core, multi-cpu, multi-node SMP computer, a hierarchy of communication

times is established. Communication between two cores on the same chip is the fastest. After this, communication times between CPUs on the same node or board is the next fastest followed by inter-node communication. Inside each of these hierarchical levels there may be subdivisions, as an example, under SGI's Hypercube configuration nodes exhibit locality. One node has 4 neighbouring nodes, 6 next nearest neighbours, 4 neighbours at a distance of 3 and a single node that is 4 edges away. Other configurations may be simpler with a flat bus connecting all the nodes. Each layout presents both challenges and opportunities that a parallel library should properly and efficiently exploit.

5.2 Parallel Options

There exist two widely used parallel libraries which are implemented on all major platforms, MPI and OpenMP. While these libraries provide a high degree of functionality, they unfortunately do not satisfy the criteria stated in the previous section. In order to implement the hybrid parallelization scheme described in the previous chapter we therefore had to develop a number of utilities of our own. These utilities are optimized to take advantage of the shared memory architectures available to us and are collected to construct a library which we refer to as shm_utils which can be linked to the simulation code.

In this section we discuss the shortcoming of the MPI and OpenMP utilities in the context of the hybrid parallelization scheme that we have implemented. We then briefly discuss the shm_util library developed as part of this project.

5.2.1 MPI

The Message Passing Interface (MPI) has become an industry standard in parallel computing. It was initially proposed in November of 1992 [1] and has since been revised and remodelled to it's current 2.0 specification. MPI's main use and strength lies in distributed memory systems such as clusters. Evaluating it's effectiveness against our requirements is complicated by the fact that there are a myriad of implementations of the specifications. We will choose to examine the MPICH 1.2.8 implementation optimized for shared memory environments.

MPI's first problem area arises when we consider low latency. Since its inception, many improvements have been made to implementations but at it's core MPI is still a parallel library meant for cluster environments. As such, basic communication costs carry unnecessary overhead, even in shared memory systems. With applications such as ours which have large amounts of communications between CPUs, fast transfer of data and synchronization is key.

The MPI versions available to us as we developed our simulation assumes a homogeneous computing environment in terms of latency between compute nodes. This ignores the topology and is another source of inefficiency.

While a good parallel tool in many respects, MPI does not adequately suit our needs.

5.2.2 OpenMP

OpenMP is a newer parallel library designed to be used in shared memory environments. It first appeared in 1997 [31] as a FORTRAN package and has since been extended to C/C++. Rather than using explicit functions, as MPI, OpenMP is coded by inserting compiler di-

rectives and comments which give hints to the compiler as to what parallelization strategies to employ. This implementation of a parallel library requires the compiler to understand a larger set of instructions than the original language specification requires which initially lead to a slower adaption to the method as is seen with the GNU set of compilers which only now plan to include OpenMP directives in version 4.2.

Since OpenMP was designed to be used in shared memory systems, unlike MPI, it does not have any unneeded legacy latency overhead. Unfortunately, OpenMP, like MPI, does not consider the varied hardware topology we are seeing in present and next generation high performance computers. Thus, while better suited to our particular problem than MPI, OpenMP's inability to take account of hardware topology severely limits its usefulness to us.

5.2.3 shm_utils

Given the shortcomings of the two industry standard parallel libraries, new parallel routines were developed as part of this research to create a parallel library designed to take advantage of the shared memory architectures and that is sufficiently aware of the topology that it distributes tasks to ensure high locality to minimize communication times.

The resulting library, shm_utils, is a small set of utilities which has function calls reminiscent of MPI but is designed to run on a shared memory architecture and has the ability to understand the topology of the underlying hardware. Appendix B details the library but we will present a few of the features here.

Communication between processes can occur in two ways. First we have a primitive socket based method with send, receive and broadcast commands. These are slow and

intended to be used as helper or debugging functions for the programmer. The second method is based on shared memory. Similar to the malloc routine in standard C, we can allocate shared memory which is accessible to all processes. This is an interesting ability which allows us to have a single array that all processes in a group can read and write to. An unplanned, but immensely beneficial, side effect of this is that we can export information from our simulation to allow external programs access to shared simulation memory. Section 5.4 will discuss the advantages of this feature.

While low latency is important, without exploiting the topology of the system, our simulation may not be running at peak performance. This was overcome in the initialization section of our parallel library which polls the current state of the SMP machine which it is running on and determines the best group of processors to run on based on availability and locality. This speeds up inherent hardare based communication overhead by a factor of three when we intelligently divide the domains.

A main feature of the library is it's process synchronization. Since this is a fundamental aspect to parallel programming, we needed to make it as light weight as possible. The following sections are devoted to that discussion.

5.3 Process Synchronization

Earlier, we made reference to a barrier function as a standard mechanism in parallel computing to enforce synchronization. It is clear from our previous discussion that synchronization routines are an essential component of any parallel implementation of the FFT algorithm since the row transforms must be completed before we begin the column trans-

forms. Without synchronization one process may complete it's assigned set of rows and begin the column transforms on incoherent data.

In this section we will take a close look at how a barrier function is implemented at the lowest level. Since this function is a significant source of overhead, we must have it execute as fast as possible so that our simulation can better scale effectively.

5.3.1 Synchronization Concepts

Process synchronization can be implemented using numerous methods: blocking communication, semaphores and spinlocks are some examples. These methods ensure two or more processes do not enter deadlock situations, operate on incoherent data or attempt to simultaneously access shared resources. The basic synchronization tools include semaphores and their derivatives which we will now briefly explain.

Semaphores

Modern operating systems provide a synchronization mechanism called a semaphore. Dijkstra [9] originally defined a semaphore as a variable shared between processes which can be manipulated by only two operators, P and V, and can take on only nonnegative integer values. The P operator checks the the semaphore s and if it is greater than zero, the operator decrements the semaphore. If the value is zero, then the operator waits until the state has changed. The V operator increments the value of s. These two operators are atomic, that is, they are guaranteed to start and complete their task before any other operaton is performed. Implementation of this atomicity is both hardware and software dependent. Using this primitive tool we can construct more advanced tools which will lead us toward

a semaphore based barrier.

Mutex

The most common use of a semaphore in parallel programming is to create a block of code whose execution is mutually exclusive. This structure is called a mutex and the block of code is referred to as the critical section. Mutexes are constructed with semaphores by first executing the P operator on a semaphore whose initial value was set to 1, performing the tasks in the critical section and then executing a V on the semaphore. Any process which arrives at the mutex when another is inside will find the semaphore's value set to zero and so the arriving process must wait until the value has increased before it can apply the P operation and enter the critical section.

Common reasons to use a mutex is when a shared resource must be accessed, be it memory or hardware.

Turnstyle

A mutex may also enclose an empty critical section, in this case we refer to the structure as a turnstyle. The construction is the exact same as a mutex except the V operation occurs immediately after the P and generally the semaphore is initialized to the zero or locked state. Turnstyles are useful constructs when a primitive, low level, one time barrier is needed. All processes arrive at the turnstyle and must wait until it is unlocked and then they can pass through it one at a time.

Barriers

While the turnstyle offers us a way of stopping all processes, it cannot function as a reusable barrier on it's own. A reusable barrier must reset it's state so that it will halt processes as effectively the second time it has been reached as it did the first time. This ability is essential to parallel programming to ensure synchronicity. In the next sections we will use the tools we have described above to implement a set of barriers, show that each functions properly and then perform benchmarks on them against the MPI barrier.

5.3.2 Semaphore Based Barriers

The first reusable barrier we will consider is the 2 stage semaphore barrier as described by Allen B. Downey[10]. This is implemented using mutexes, turnstyles and shared memory. We will present the routine and then discuss the details.

```
#define Mutex1 comm->semid[0]
#define Mutex2 comm->semid[1]
#define Turnstyle1 comm->semid[3]
#define Turnstyle2 comm->semid[4]
#define UNLOCK &comm->unlock
#define LOCK &comm->lock
void SHM_Barrier(struct SHM_Comm* comm)
{
    if(comm->np == 1)
        return;                                                    10
    int id = comm->id;
    int np = comm->np;
```

```
volatile int *volatile b = (volatile int*)comm->barrier->memory;

    semop(Mutex1, LOCK, 1);
        __mf();
        b[0]++;
        if(b[0] == np) //last one through
        {
            __mf();                                          20
            semop(Turnstyle2, LOCK, 1);
            semop(Turnstyle1, UNLOCK, 1);
        }
        else
            __mf();
    semop(Mutex1, UNLOCK, 1);

    semop(Turnstyle1, LOCK, 1); //all wait until unlock
    semop(Turnstyle1, UNLOCK, 1);
                                                             30
    semop(Mutex2, LOCK, 1);
        __mf();
        b[0]--;
        if(b[0] == 0) //last one through
        {
            __mf();
            semop(Turnstyle1, LOCK, 1);
            semop(Turnstyle2, UNLOCK, 1);
        }
        else                                                 40
```

```
                __mf();
        semop(Mutex2, UNLOCK, 1);

        semop(Turnstyle2, LOCK, 1); //all wait until unlock
        semop(Turnstyle2, UNLOCK, 1);
}
```

Semaphore Based Barrier

Though initially daunting, this routine is not overly complex. The set of defines are used to relabel the code for readability and rename a set of semaphore identifiers in the SHM_Comm structure (see Appendix B for details) as the two mutexes and two turnstyles. It also relabels the locking *P* and unlocking *V* operaions. All semaphores are initialized to the unlocked or one state except for turnstyle number 1 which begins in the locked or zero state, the shared integer *b* is initialized to zero.

The logic of the barrier is straightforward, the opportunity is first provided for a single process work group to immediately exit as no synchronization needs to be done. Next, as each process passes through the first mutex they each increment the value of *b* and then wait at the first turnstyle. The last process to pass through the first mutex will increment *b* so that it's value is equal to the process count and that process will lock the second turnstyle and open the first. This allows all processes to move on to the second stage of the barrier. As each process passes through the second mutex the value of *b* is decremented. The last process though the mutex will lock the first turnstyle, which is essential in making this a reusable barrier, and then unlocks the second turnstyle. At this point all processes are free to exit the second stage and the barrier. The states of the semaphore and shared memory is the exact same as it was before the barrier was encountered.

One item of note in the above code is the __mf() command. This is an Intel Compiler intrinsic function which executes a memory fence. This ensures that the increment of the shared variable *b* is known to all processes. Without an explicit memory fence, it is possible for a process to modify a variable but not flush that modification from it's on-chip write-back cache to the shared system memory even if it is defined as volatile memory.

5.3.3 Shared Memory Barrier I

While the two stage semaphore based barrier functions perfectly, we wanted to explore other methods of implementing reusable barriers. The following is code we designed to implement a barrier without explicit semaphores.

```
void SHM_Barrier(struct SHM_Comm* comm)
{
    if(comm->np == 1)
        return;
    int id = comm->id;
    int np = comm->np;
    volatile int *volatile b = (volatile int*)comm->barrier->memory;

    if(id == 0)
        b[1]=0; //close gate on b1                                    10

    _InterlockedIncrement((void*)&b[0]);
    __fwb();
    while(b[0] != np); //wait on b0
```

```
        if(id == 0)
                b[2]=0; //close gate on b2

        _InterlockedIncrement((void*)&b[1]);
        __fwb();                                                    20
        while(b[1] != np); //wait on b1

        if(id == 0)
                b[0]=0; //close b0

        _InterlockedIncrement((void*)&b[2]);
        __fwb();
        while(b[2] != np); //wait on b2
}
```

Shared Memory Based Barrier I

This three stage shared memory barrier uses three shared integers. The first is initialized to zero which effectively closes the first gate, which is the empty while loop on line 14. As each process approaches the first while loop it increments $b[0]$ with the intrinsic atomic increment operator supplied by the Intel Compiler. Following this another intrinsic is called to flush the write back cache from each CPU. When the primary process, process number 0, approaches the first while loop it closes the second gate by setting $b[1] = 0$ before incrementing $b[0]$.

At each of the three stages processes must wait for all of the others to reach the gate, or while loop. Before the gate is opened, the next gate is locked. These three stages are required to properly implement a reusable barrier. If only two of these stages were used, it

is possible that back to back execution of the barrier could lead to an inconsistent state.

5.3.4 Shared Memory Barrier II

The previous barrier did not have any explicit semaphores or mutexes, but inside the construction of the intrinsic atomic increment operator is a hidden critical section. A slightly different approach would need to be implemented to create a truly semaphore-less reusable barrier. The following is what we designed.

```
void SHM_Barrier(struct SHM_Comm* comm)
{
        if(comm->np == 1)
                return;
        int id = comm->id;
        int np = comm->np;
        int np2 = np*2;
        int np3 = np*3;
        int i=0;
        volatile int *volatile b = (volatile int*)comm->barrier->memory;

        b[id+np] = 0; //close b1
        b[id] = 1;
BAR0:
        if(!b[i])
                goto BAR0;
        i++;
        if(i^np)
                goto BAR0;
```

```
                                                                    20
        b[id+np2]= 0; //close b2
        b[id+np] = 1;
BAR1:
        if(!b[i])
                goto BAR1;
        i++;
        if(i^np2)
                goto BAR1;

        b[id]= 0; //close b0                                        30
        b[id+np2] = 1;
BAR2:
        if(!b[i])
                goto BAR2;
        i++;
        if(i^np3)
                goto BAR2;

        return;
}                                                                   40
```

Shared Memory Based Barrier II

In this code, rather than atomically incrementing shared memory, each process effectively sets a bit in a bitstring and then checks to see if all the bits are set. As above, this is split into three stages. This method has the advantage of not needing each process to perform different tasks. In the previous, the primary process was responsible for locking

gates, here each process locks it's share of the next gate before unlocking it's portion of the present gate.

5.3.5 Comparison of Barriers

Benchmarks of each of the four methods, the *MPI* barrier, the semaphore based barrier and the two shared memory based barriers are presented in Table 5.1. This is an average from 100000 samples with sets of 1000 samples drawn from each method sequentially to minimize the effect of a varied computational environment on the results. From these

Average Time per Barrier (μs)				
Process Count	1	2	3	4
MPI	0.7356	50.30	107.6	133.8
Semaphore	0.7302	10.72	30.09	32.71
Shared Mem I	0.7276	10.31	21.48	31.52
Shared Mem II	0.7310	11.13	21.05	32.07
Process Count	5	6	7	8
MPI	192.2	233.2	254.4	357.6
Semaphore	66.58	74.31	100.3	134.1
Shared Mem I	56.30	73.15	98.84	120.7
Shared Mem II	59.62	75.93	102.4	130.5

Table 5.1: Comparison of various barrier implementations

results it is clear that the MPI barrier also uses the short-circuit return when there is only one process in the current work group. It is also obvious that MPI is not being used in the

proper environment, it is designed for systems with high latency between nodes such as clusters. It does not make efficient use of shared memory or semaphores which is why it is consistently 3 times slower than the other methods listed. This example of performance loss with MPI's barrier call is typical in a shared memory environment and affects all of it's data sharing and synchronization functions. If we had chosen to use an industry standard parallel library it is clear that this would not have been the optimal solution.

The execution times of the semaphore and shared memory barriers are extremely close but there is a consistent order when ranking the runtimes. We have decided to use the Shared Memory Barrier I in our simulation while running on the SGI Altix hardware. We have left the other barriers in the code and they can be activated with compile time flags, this allows a user of the library to choose the best barrier for their combination of operating system and hardware.

A note on implementation, it is possible to create a two stage shared memory barrier if you can guarantee that you have 2 separate barriers and you alternate between them. The easiest way to do this is to create a shared array of 4 integers and have a pointer which swaps between the first and third element on each execution of the barrier. This pointer is then used as the b variable in the methods above. This was implemented but resulted in no savings over our best method.

5.4 Interacting with Real-time Simulations

Our shared memory library also affords us interesting methods of interacting with the simulation in "real-time". Generally, interaction with a simulation requires periodic event

polling in the main loop. This both increases the complexity of the code and has an adverse impact on the performance of the application.

A positive side effect of our memory system is that when we claim memory, a shared memory identifier is returned. We can export this identifier from our simulation and other programs can then use the information to get either read-only or read-write access to the shared memory of the simulation.

The main simulation code has the capability to export information which allows other processes to directly access the memory which contains spin configuration, interaction strengths, damping, noise amplitude and adaptive step tolerance. Using this, we constructed several small server programs which make this data available to remote clients via network connections. This allows the remote clients to both visualize the simulation as it procedes on the high performance computer and alter the parameters of the simulation.

This provides researchers with dynamic, real-time control over the simulation and the ability to visualize and analyze the evolution of the simulation to rapidly explore configuration space for interesting areas. This is a significant enhancement over previous methods which would require many tens or hundreds of batch jobs to examine a range of parameters.

5.5 Conclusion

In this chapter we described how the numerical solution to the LLG equation could be parallelized to optimize the run time. The tools required to implement the optimal parallelization scheme for the solution required the development of a set of shared memory utilities, which were incorporated into a library. This library addresses many of the short-

comings inherent in the main existing parallel libraries, MPI and OpenMP which made them unsuitable for the parallelization strategies we wished to implement. Specifically, MPI falls short when considering latency as it is based on a cluster architecture. OpenMP fares much better in this regard as it was designed as a parallel solution for SMP machines. Neither strategy optimally selects a computational topology nor allows zero-copy access to the simulation memory.

Chapter 6

Data Management, Analysis and Visualization Tools

A challenge with all complex simulations is how to manage, analyze and visualize the potentially vast amounts of data that is produced in the course of a large scale simulation. The development of methods to accomplish these tasks in an efficient manner that can be readily adapted to study a wide variety of problems is critical in acquiring insight into the underlying physics. In this chapter we discuss how we have approached data management, that is, how we transfer the data produced by the simulation to other programs, the analysis of the data, examining in detail the evolution of the system energy and the visualization tools that we have developed.

6.1 Data Management

During a simulation, system states are written to a datastream which is an abstraction realized either by writing to a file or forwarded across a network to remote clients. In our terminology, we refer to each recorded state as a frame. Each frame is a container that holds information about the system at a particular time such as spin configuration, local fields, system parameters and energies. Both the layout and format of these data are variable which allows considerable flexibility both in the way data can be written to and read from the datastream.

Each datastream begins with a header that defines the layout and format of the data, initial conditions, system parameters and the particular algorithms used to compute the data as well as system sizes and frame counts. Following the header, the frames appear sequentially in the datastream.

The header for a sample datafile is presented in Table 6.1. This datafile begins with a description of the data in the frames. In this case, the identifier is `#3dxyz` which indicates that we will be dealing with a potentially multilayered system with spin data represented in Cartesian coordinates. The next 17 lines, each also starting with a hash mark (#), give information about the initial spin configurations, parameters and algorithms used. Here we see the strengths of the exchange, dipolar, anisotropy and thermal fields. Next, information of the damping parameter and applied fields is presented. The "Total Time" value defines how much simulated time is requested, "Report At" defines the frequency that frames are written to the datafile and "Tolerance" controls the adaptive timestep. "Initial Position" defines one of 40 hard coded configurations that the simulation can start with, "Random Seed" is the seed used in the random number generators.

#3dxyz	# Initial Position: 6
#System Dimensions: 64x 64x 1	# Random Seed: 223130
# J: 8.900000	# Exclude Z: False
# g: 1.000000	# Old Style Full: False
# Kappa: 0.000000	# Truncated: False
# Epsilon: 0.500000	64
# Alpha: 0.750000	64
# Applied X: 0.000000	1
# Applied Y: 0.000000	20
# Applied Z: 0.000000	1.0 0.0 0.0
# Total Time: 10	-1.0 0.0 0.0
# Report At: 0.5	1.0 0.0 0.0
# Tolerance: 1.000E-05	-1.0 0.0 0.0

Table 6.1: Sample datafile

The remaining three comments describe some of the internal workings of the simulation. "Exclude Z" is true if the spins are restricted to in-plane motion, in this example case they are not. "Old Style Full" is true when we want to use the $O(N^2)$ method for dipolar interaction and "Truncated" is true if we wish to restrict the dipolar field to the nearest neighbours.

The next three lines, 64, 64 and 1 define the system dimensions and final line of the header information, 20, states the number of frames of data that will be present in this file.

As the simulation proceeds, the frames are converted and inserted into the datastream in one of several formats. The frame formats that are currently supported are listed in Table 6.2 together with the data that is recorded in each frame. For example, the header shown in Table 6.1 states that the frame will be in the frame format referred to as `#3dxyz`. In this particular format each frame includes the spin configuration in which each spin is recorded as a 3 component Cartesian vector. Other frame formats will include the spin configuration in other coordinate representations as well as information of fields, energies and interaction strengths.

Identifier	Elements
`#3dcomplete`	`THREE-DIM, COMPONENT-THETA-PHI-RAD`
`#3dxyz`	`THREE-DIM, COMPONENT-XYZ`
`#socket`	`THREE-DIM, COMPONENT-XYZ, TEMPERATURE, TIME`

Identifier	Elements
	EXCHANGE, DIPOLE, ANISOTROPY, EXTERNAL-H
#3dfieldcomplete	THREE-DIM, COMPONENT-THETA-PHI-RAD, FIELD
#3dbintemptimefieldcomplete	THREE-DIM, COMPONENT-THETA-PHI-RAD,
	FIELD, TEMPERATURE, TIME, BINARY
#3dtemptimefieldcomplete	THREE-DIM, COMPONENT-THETA-PHI-RAD,
	FIELD, TEMPERATURE, TIME
#3dEnergyComplete	THREE-DIM, COMPONENT-THETA-PHI-RAD, ENERGY
#3dfull	THREE-DIM, COMPONENT-THETA-PHI
#vacancies_t	TWO-DIM, COMPONENT-THETA, VACANCIES
#vacancies_3d	TWO-DIM, COMPONENT-THETA-PHI, VACANCIES
#ising_2d	TWO-DIM, COMPONENT-ISING

Table 6.2: Header identifiers and corresponding elements

Each identifier is the result of the bitwise OR between each element. The elements themselves indicates the presence or format of data in each frame. For instance, COMPONENT_THETA_PHI_RAD indicate that the vectors defining the spins are in spherical coordinates where as COMPONENT_ISING states that the spins will be -1 or 1. If the element ENERGY is included in the identifier then data regarding the energy is provided in each frame. The same follows for VACANCIES, TEMPERATURE and EXTERNAL_H. Most other elements define the structure of the data.

These identifiers, made up of elements, gives us a method to flexibly store data related to our spin systems. This flexibility allows us to record only the aspects of the simulation that are interesting for a given study. It also allows us to improve or add information to

what is recorded from the simulation and to read both new and old data files.

The frame definitions are part of a unified library for interacting with simulation data. We have developed a C++ library, libSpin, specifically to aid us in developing all other applications which operate on simulation output. It is responsible for opening files in various formats and, depending on the identifier, read each part of the frame.

The information for each of these frames are stored in instances of classes which provide convenient routines such as energy calculations, topological singularity detection and access to vectors defining site data in Cartesian and spherical coordinates. The library also manages each of these instances, holding as many in local memory as needed and freeing instances of frames which have not been used recently so that very large datasets can be accessed without consuming all the memory of the computer.

The following will demonstrate the ease of use and capabilities of libSpin.

6.2 System Energy

Our libSpin library allows us to interact with simulation data without needing to recode tedious low level formatting and caching for each task. Understanding the dynamics of the energies in the simulation is a key element to understanding the overall system. Calculating energies is such a fundamental part of studying magnetic systems that we decided to move these routines as defined by Equation 2.15 into our libSpin library. This renders the implementation of a program to output energy quite trivial. The following figure demonstrates how we would code a simple case using C++.

```
#include <iostream>
```

```
#include <libspin.h>
void main()
{
  fileReader FR; //create a (libSpin) file reader for simulation data
  FR.open("spinData.txt"); //open a datafile and determine the proper parsing method
  SpinSystem* sys = FR.getSystem(0); //read the first frame and assign its address to sys
  std::cout << sys->getAverageTotalEnergy() << std::endl; //print the average energy
}
```

Calculating Frame 0 average energy with libSpin

In this code we use a file reader, which is a custom class provided by libSpin, to open and interact with a data source. This can operate on any combination of text, binary, compressed or socket based sources and frame formats. If the energy is provided in the file, it will sum that data and return it when needed. If spin orientation and field data are present then it will return the average dot product. If no field or energy data is found in the frame then it will calculate the field using either the $O(N^2)$ method, or by FFTs if a library is available. The method "getSystem" instructs the file reader to return the frame information, in this case frame number zero, in whatever manner is most appropriate for the file format. Frame information such as spin orientations, energies and local fields are encapsulated in the SpinSystem class which provides a host of methods to operate its the data. One of these is the "getAverageTotalEnergy" function which hides all the details required to calculate the average system energy. This short program preforms a complicated task that would require hundreds of lines of code if we did not have libSpin.

Our main tool for calculating system energy has more features, flexibility and error checking but it follows the same basic structure: open a data source (our tools accept

command line input to specify data sources and parameters), select a frame and output an energy. Generally we wish to see the energy as it changes over frames so we place our select frame and output steps in a loop.

6.3 Visualization Methods

The code we have developed is mainly used to study system dynamics so we are particularly interested in visualizing the evolution of the systems. To do this we have developed two visualization tools, sdlglspin and povspin, both of which make extensive use of the routines available through libspin described in the previous sections. While both programs can visualize the data in a variety of different ways, the approaches used in the two applications are quite different.

6.3.1 SDL/OpenGL

The first application, sdlglspin, uses the routines in spinlib to convert the datastream generated by the simulation into a sequence of 3D images in realtime. This is done with the aide of both OpenGL, which provides the 3D pipeline between virtual 3D structures and a raster image, and Simple DirectMedia Layer (SDL), which offers a cross platform environment for window management and user interaction.

OpenGL was selected because it allows accelerated graphics which are essential to maintain the high frame rate required for real time interaction. A software based 3D pipeline would not have been sufficient to display our images since it is completely executed on a local CPU, from scene description to raster projection, and then sent to the

video card as a texture. OpenGL provides us with an industry standard API which allows us to send scene descriptions to the Graphics Processing Unit (GPU) for rendering. This both frees the CPU for other computations and avoids the bandwidth bottleneck inherent in streaming large textures from system memory to graphics memory.

SDL provides a richer and more modern API to manage windows and user input than the traditional GL Utility Toolkit (GLUT) while maintaining a cross platform code base. GLUT manages user events via callbacks, pointers to functions, which cannot be methods of classes. To achieve a mixing of GLUT and Object Oriented Programming, wrapper functions must be created to stitch the two paradigms together. SDL handles events in an event loop which allows the explicit handling of desired events via either functions, object methods or hard coded statements inside the event loop itself. SDL also supports True Type fonts which allows us to overlay textual information on the graphics.

Using a blend of the above technologies, we are able to present spin orientation as 3D colour coded or static coloured arrows, local fields and mark the locations of topological singularities (see Section 7.1) as well as a continuous mesh which can represent spin orientations, site energies or changes in rotation. Figure 6.1 demonstrates these capabilities by rendering a datastream generated by a simulation with exchange interaction strength set to 2.0, a dipolar strength of 0.2, no surface anisotropy, external fields or thermal perturbations.

6.3.2 POV-Ray

While sdlglspin is designed to generate 3D graphics at a sufficiently high frame rate that the images smoothly blend together simulate motion, POV-Ray reads the datastream from the simulation using libSpin to generate a sequence of POV-Ray files. These files are human

readable descriptions of a 3D scene representing the data generated by the simulation at a particular point in time. These files are then rendered by the POV-Ray ray tracing program which creates a sequence of raster images. Since the POV-Ray files consist of a mathematical description of the volumes and surfaces in the scene, the 3D ray traced image can be rendered at any arbitrary resolution.

Individual magnetic moments in povspin can be rendered in one of two styles, colour coded 3D arrows with a ring around the head and dot on the tail to help differentiate the orientation and a black cone with a white dot on the base. These two styles are used in different settings. The coloured 3D arrow are best used in presentations while the black cones offer a more professional style for papers or articles.

The advantage of this method over the OpenGL method is image quality. Shadows are properly calculated, both in terms of face normals and objects casting shadows on others in the scene, and edges are antialiased. These high quality images can then be used to encode a high quality video. Generally though, this will take more time to render than the original simulation which created the data.

6.4 Conclusion

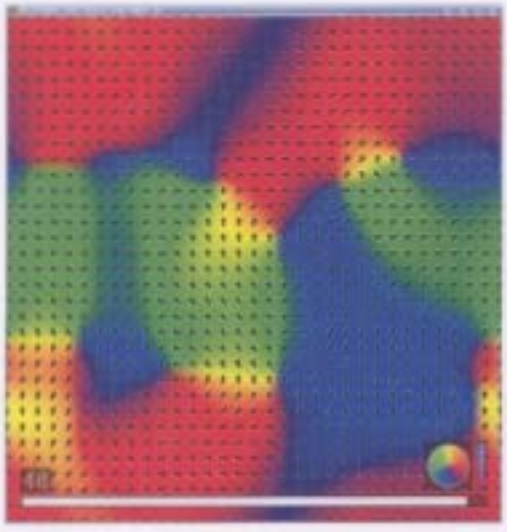
In this chapter we presented a set of steps which translate from a datastream realized as a file or socket to final, useful products such as energy calculations or 3D renderings. We have strategies to structure the information in our datastreams which give us great flexibility in terms of what is described within and in what fashion that information itself is formatted. From this point we can convert the data into a unified structure defined in libSpin which

has a host of analysis functions which can operate upon it. When using libSpin all of this is transparent, it is supplied with a resource name which is either a file in text, binary, gzipped or bzipped format or a remote resource which consists of a host name and port number and the library determines how to properly extract the information from that source.

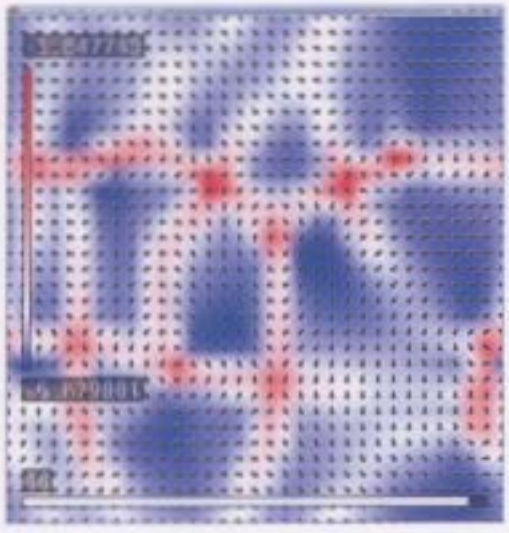
With the use of libSpin, tasks which would generally take hundreds of lines of code to complete can be accomplished in a handful of lines of code as seen in the energy example in Section 6.2. More complex tasks, such as OpenGL applications or translating scenes into POV-Ray files, are now readably achievable as a framework exists to take care of all tedious low level formatting, memory management and coordinate conversions.

A possible next step in data management would be to migrate from our custom representation and format of the data to the Hierarchical Data Format[14]. This would allow external programs such as IBM's OpenDX or Wolfram's Mathematica to readily import our simulated data. We have not yet undertaken this migration because of time constraints, increased code complexity and an unwillingness to lose access to gigabytes of legacy data.

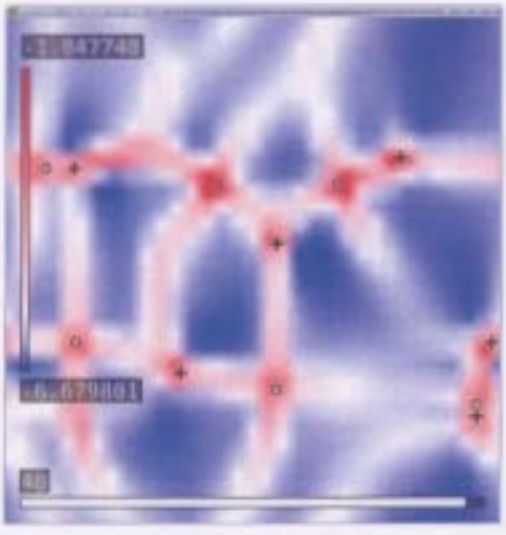
In the next chapter we will present several studies which make use of the simulation we have defined and implemented as well as libSpin to extract information from the generated data.



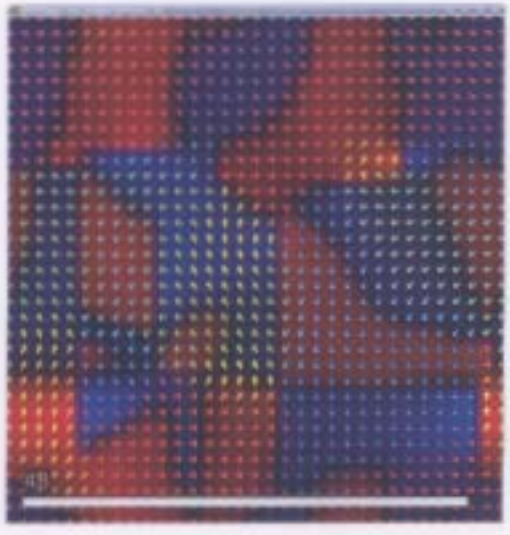
(a) Spin configuration represented by mesh and monochrome arrows



(b) Spin energy distribution with overlayed monochrome arrows representing spin orientation



(c) Spin energy distribution with overlayed icons representing topological singularity type and position



(d) In plane rotation direction and strength with overlayed 3D coloured arrows representing spin orientation

Figure 6.1: Four samples of display options using sdlglspin for a single frame of data

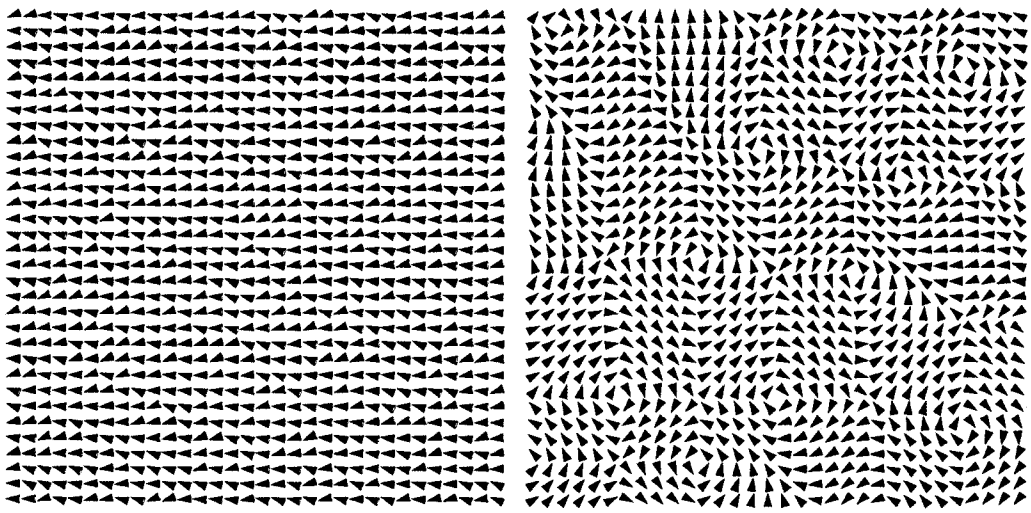
Chapter 7

Simulation Examples

This chapter focuses on presenting simulation examples of various systems. We will examine a single layer, square lattice, multilayer systems and non-square lattices. Our goal is to demonstrate the strength and flexibility of the code described in this thesis.

7.1 Relaxation in 2D systems

In magnetic systems there are two distinct types of disorder that can arise in ordered systems. The first type of disorder is simply the disorder created by the thermal fluctuations of the spins about an ordered state. The nature of these fluctuations are such that, below the transition temperature, the magnetic system can maintain long range magnetic order. The second type of disorder is topological disorder in which the entropic contributions of certain topological singularities can be sufficient to overcome the energy required to create the singularities. While the topological disorder can destroy long range magnetic order, at low temperatures, the system can exhibit short range order. The disappearance of short

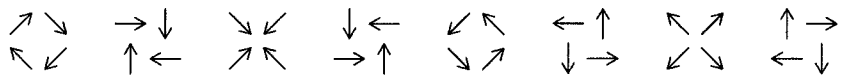


(a) Microscopic Disorder (b) Macroscopic Disorder

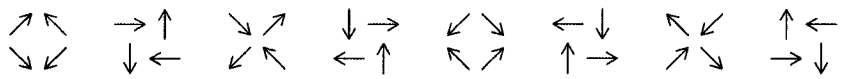
Figure 7.1: These figures show the two classifications of disorder

range order at the transition temperature is associated with the unbinding of the pairs of topological singularities and is referred to as a Kosterlitz-Thouless transition[22]. Such behaviour is common in two dimensional magnetic systems.

Examples of these two types of disorder are illustrated in Figure 7.1 for a square lattice. Figure 7.1(a) shows the effect of thermal fluctuations in a ferromagnetic system. This system clearly exhibits long range ferromagnetic order. Figure 7.1(b) on the other hand shows the effect of topological disorder. Here the orientation of the spins are highly correlated over short distances but, because of the topological singularities, the system does not exhibit long range magnetic order. These topological singularities can be classified as belonging to one of two groups which we label Type +1 and Type-1. The first, Type+1 singularities, are shown in Figure 7.2(a), and are analogous to centres, vortexes, sinks and sources in fluid systems. In the figure we show a range of both Type +1 and Type -1 spin



(a) Type +1 Singularities



(b) Type -1 Singularities

Figure 7.2: Classification of Topological Singularities

configurations, each one can be generated by rotating each spin in the previous by $\pi/4$. The second set, Type -1 singularities, are displayed in Figure 7.2(b). These are reminiscent of saddle or stagnation points in fluid systems. Again, we show a range of configurations, each a rotation of the last. It is important to note that we cannot map from a Type +1(-1) singularity to a Type -1(+1) by a continuous rotation of all four spins. As such, they are topologically distinct. This classification of singularities is similar to the classification used by Chien et. al.[5]. In their work they referred to the winding number of a structure which could take on a value of +1 or -1 and equates to our Types.

These singularities alone are of considerable importance in the study of two dimensional magnetic systems. In the case of a square lattice with only an isotropic exchange interaction, the energy of both Type +1 and -1 singularities are continuously degenerate under the rotation that maps between the singularities shown in Figures 7.2(a) and 7.2(b). This degeneracy, however, is removed by the addition of the dipolar interaction and certain topological structures are preferred, reflecting the fourfold symmetry of the underlying lattice.

The dynamics of topological singularities play an important role in determining the magnetic response of two dimensional systems, particularly in the relaxation process from

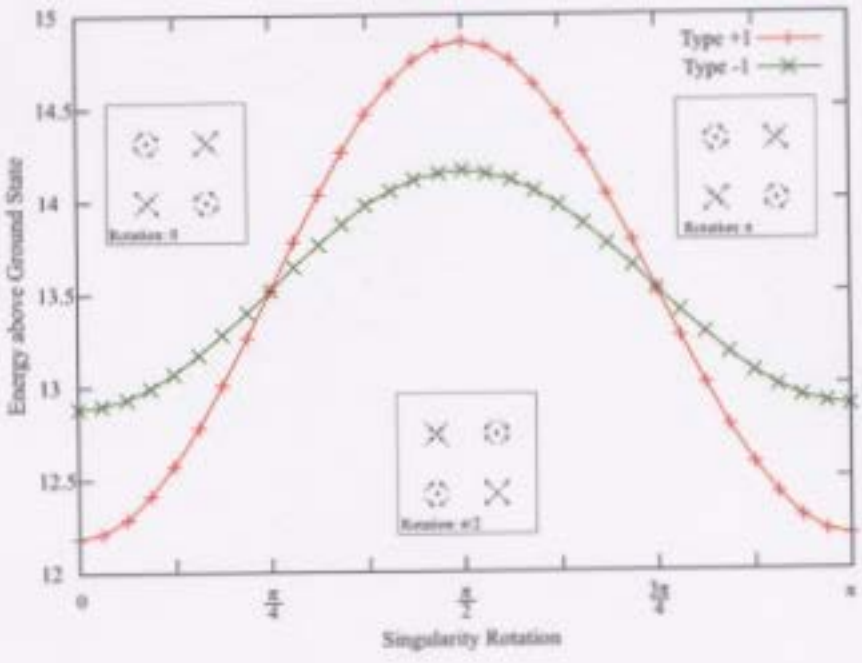
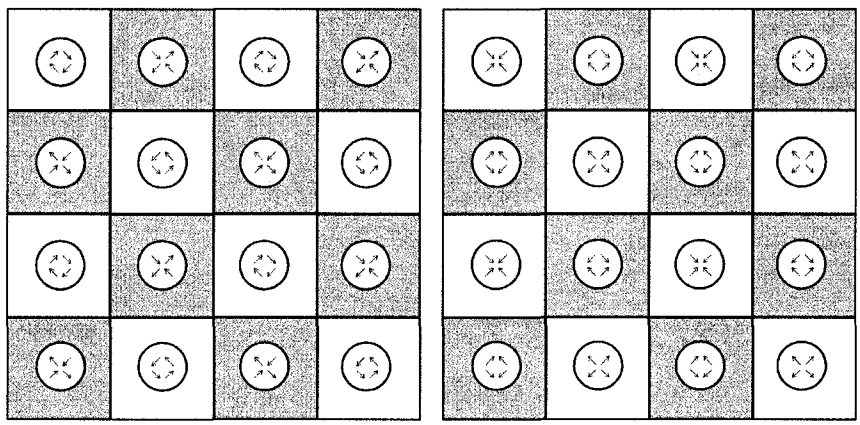


Figure 7.3: Singularity energy averaged over unit cell. J=3.0, g=2.0.

a high energy disordered state to a low energy ordered state. Our simulation shows that the singularities generally locate near the centre of a unit cell and migrate by making brief hops between sites where they remain while the system relaxes around the new configuration. This is similar to the dynamics of an atom adsorbed on a solid surface.

In Figure 7.3 we plot the local energy of the Type +1 and Type -1 singularities[1] as a function of singularity rotation. The data is extracted from calculations of the energy of a periodic array of singularities setup in a checkerboard manner with 32 lattice sites separating them laterally. The low and high energy rotation configurations, under dipolar

[1] We define the local singularity energy as the scalar product of average local field and magnetic moments for the 4 encompassing sites



(a) Low energy configuration (b) High energy configuration

Figure 7.4: Checkerboard arrangement of singularities

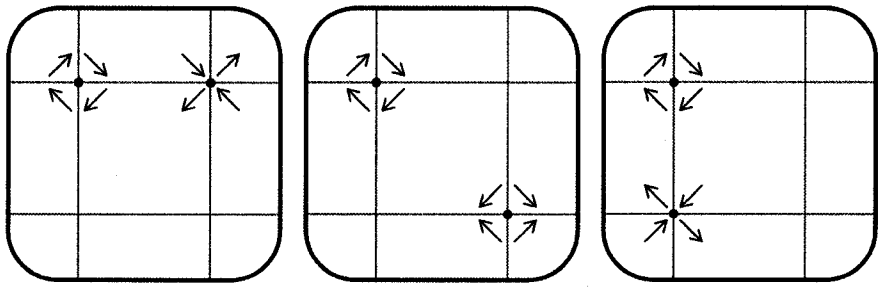
interaction, are depicted in Figure 7.4. From Figure 7.3, we can state that:

1. The energy of the Type +1 and Type -1 singularities have a different dependence on orientation

2. There are certain maximum and minimum energy configurations that correspond to certain high symmetry spin configurations

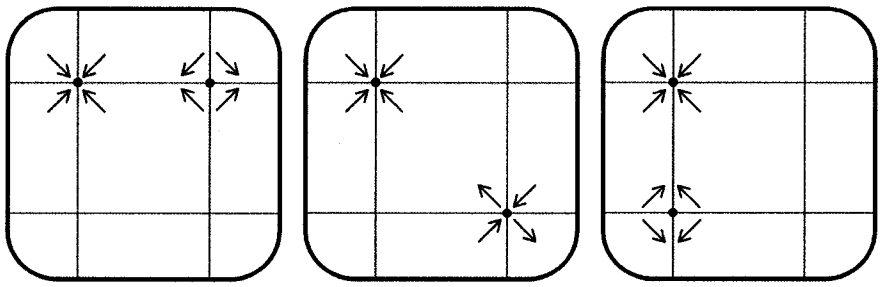
Specifically, at low energy rotations (0, π), Type +1 singularities have lower local energy than Type -1 while at higher energy rotations ($\pi/2$) the reverse is true. This leads to preferred configurations both in terms of local singularity orientation and relative locations on the lattice.

At lower energy rotations, singularities will tend to line up laterally rather than diagonally. Figure 7.5 depicts 3 pairings with a fixed, low energy Type +1 singularity. In these configurations, the Type -1 singularity will be in a low energy state when it is aligned either



(a) Low energy pair (b) High energy pair (c) Low energy pair

Figure 7.5: Singularity pair configurations with a fixed low energy Type +1



(a) High energy pair (b) Low energy pair (c) High energy pair

Figure 7.6: Singularity pair configurations with a fixed high energy Type +1

horizontally or vertically with the Type +1 singularity. Figure 7.6 demonstrates the configuration when a Type -1 singularity is paired with a fixed, high energy, Type +1 singularity. In these cases we can see that aligning vertically or horizontally results in a high energy Type -1 singularity while diagonal alignment allows a lower energy Type -1 singularity. By varying the location of a singularity within a unit cell we are able to map out the local energy well of these structures. Figure 7.7 shows that a singularity is at the lowest energy when it is centred in the unit cell. This unitless figure is representative of both Type +1 and Type -1 singularities but the slope of the well is related to the energy at the centre

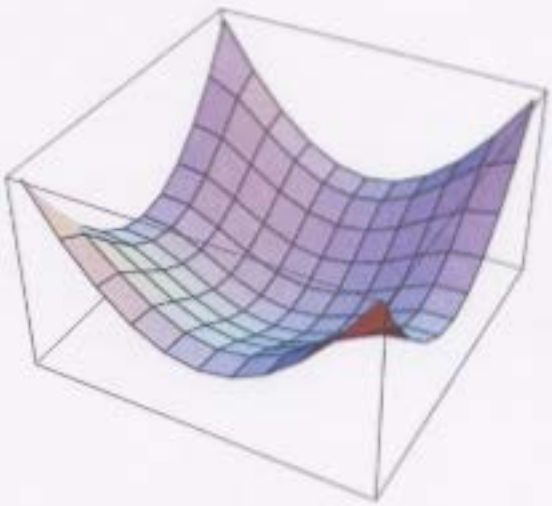


Figure 7.7: Local energy well of a singularity in a unit cell

of the well. Higher energy singularities have shallow wells while lower energy singularities have deeper wells. This impacts the translational dynamics of the singularities; lower energy singularities will make hops from cell centre to cell centre whereas higher energy singularities will move more easily.

These observations of isolated and paired topological singularities allow us to predict and understand the kinematics of two dimensional magnetic systems with both exchange and dipolar interaction. If we assume system kinematics are driven by energy dissipation then we may state the following phenomenological descriptions based the simulations we have studied.

Observation 1 *Singularity Energy Dissipation*

(i) High energy singularity pairs will migrate to a diagonally offset relative alignment

(ii) Low energy singularity pairs will migrate to a laterally offset relative alignment

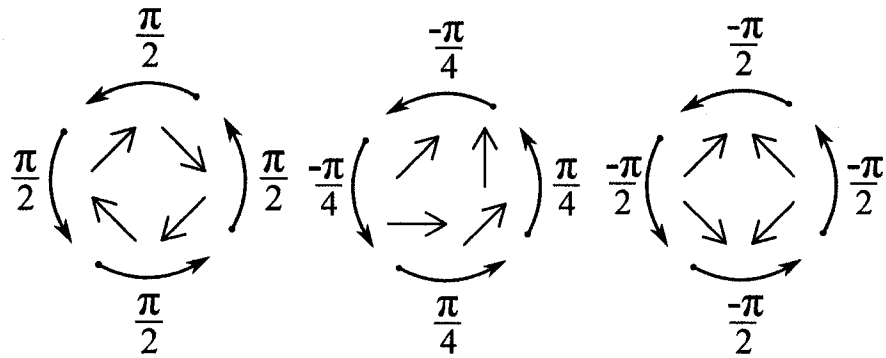
(iii) High energy singularities will rotate to lower energy singularities of the same type

(iv) High energy singularities will migrate more easily than low energy singularities

However, studying the kinematics of these topological singularities is not straightforward. While they may be easily detected by the eye, locating and tracking them is not straightforward. The technique we use to solve this problem is borrowed from the field of fingerprint analysis. Maltoni[27] characterizes a fingerprint by a set of loops, whorls and deltas. These structures do not translate perfectly to our lattice of spins but the method used to locate them does. Loops and deltas do not have corresponding configurations but the whorl is analogous to structures 1 and 5 of our Type +1 singularities.

Maltoni utilizes the Poincaré Index as a method to locate these structures. This is done by summing the offsets between consecutive angles around a closed loop on a vector field. For our lattices, we consider loops to be the 4 spin sites which define a unit cell. With this definition, our Poincaré index will always be either 2π, 0, or -2π. Figure 7.8 illustrates this concept clearly. Figure 7.8(a) is the first structure in Figure 7.2(a) and as we sum angle offsets in a clockwise manner, we arrive at 2π which indicates that this is a Type +1 Topological Singularity. Summing around Figure 7.8(b) results in 0 signalling that there is no singularity present and the Poincaré Index of Figure 7.8(c) is -2π which correctly indicates that this a Type -1 Topological Singularity.

This detection method is computationally efficient as it can determine if a singularity is present inside a unit cell using only 4 subtracts and 4 adds. To pinpoint the position inside the cell we drop streamlines into the search region and trace either along the interpolated fluid flow for Type +1 singularities or perpendicularly for Type -1. Global rotations of the vectors may be required to quickly detect some of the structures.



(a) 2π: Type 1 Singularity (b) 0: No Singularity (c) -2π: Type -1 Singularity

Figure 7.8: Poincaré Indices of example spin configurations

7.1.1 Relaxation through annihilation of Type ± 1 Singularities

In order to illustrate how our LLG code and singularity detection algorithm can be used to study the dynamics of singularities we consider the evolution of a pair of Type ± 1 singularities. We prepare an initial state consisting of two idealized singularities on a 64×64 lattice with an in plane spin configuration described by

$$\Theta_{\vec{r}} = \mathrm{Arg}\left(\frac{x-x_+ + i(y-y_+)}{x-s_- + i(y-y_-)}\right) + R_\Theta, \tag{7.1}$$

where $\vec{r}_\pm = (x_\pm, y_\pm)$ denotes the position of the Type ± 1 singularities and R_Θ denotes a global rotation of the spins.

In this example, we set

$$\begin{aligned} x_+ &= 32.5 \quad x_- = 32.5 \\ y_+ &= 21.5 \quad y_- = 43.5 \\ R_\Theta &= 0 \quad\quad \alpha = 0.5 \\ g &= 1.0 \quad\quad J = 3.0 \end{aligned} \tag{7.2}$$

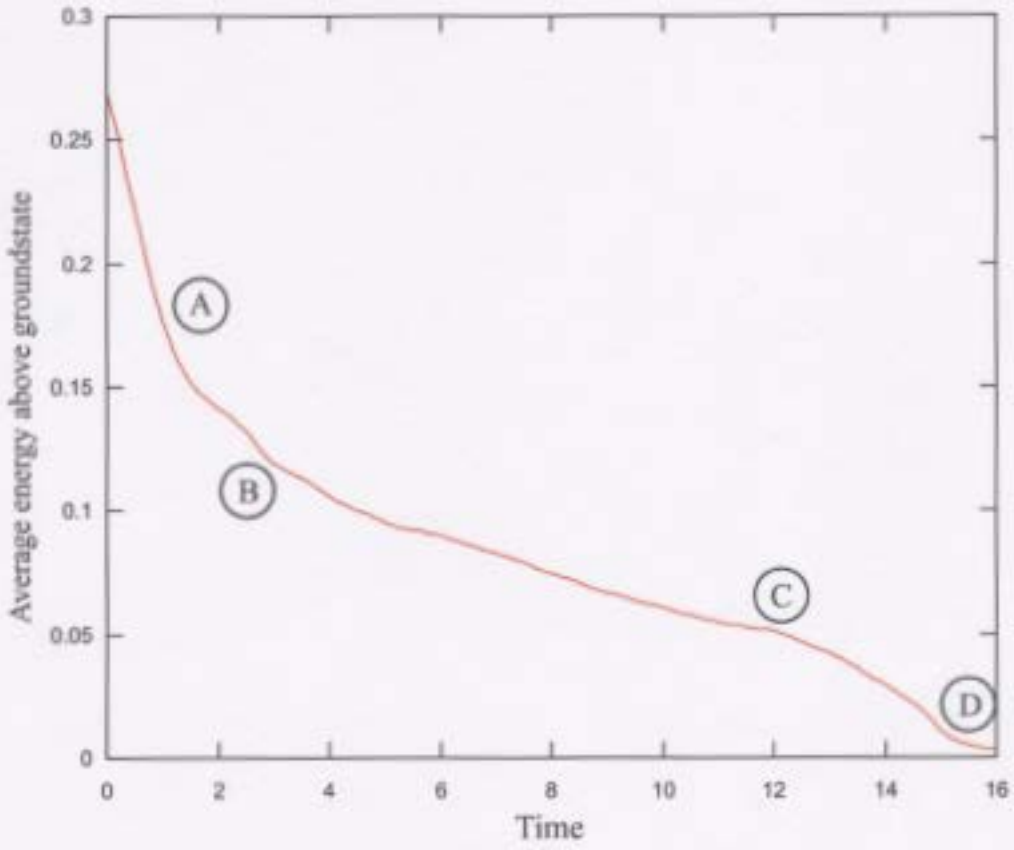


Figure 7.9: Time-Average Energy above Groundstate plot of two singularities annihilating

and all other fields and parameters equal to zero. The above choices corresponds to a high energy Type +1 singularity at $\vec{r}_+$ and a low energy Type -1 singularity at $\vec{r}_-$.

A naïve expectation for this relaxation process would be the migration of the two singularities toward each other leading to an eventual annihilation. Indeed this is involved in the process but Figure 7.9 shows that the process is obviously more complicated.

Considerable insight into the process can be gained by tracking the singularities using the method described earlier. The initial rapid relaxation (region Ⓐ on the figure) core-

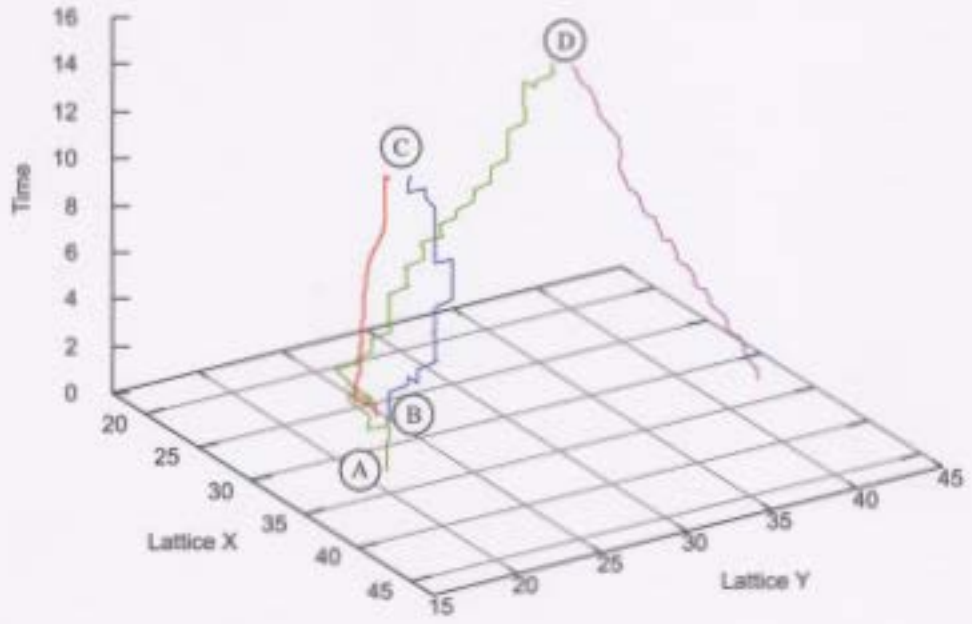


Figure 7.10: Space-Time Plot of singularity creation, evolution and annihilation.

sponds to the spins rearranging themselves around the initial idealized pair of singularities defined by Equation 7.1. Once this equilibration process is complete, the system further relaxes by the Type +1 singularity, initially located at $\vec{r}_+$, moving horizontally and thus increasing the distance from the Type -1 singularity at $\vec{r}_-$. This motion is consistent with Theorem 1, statement *i*. This motion is coupled with rotation of the singularity from high energy to low energy which is expected from statement *iii*. The local energy shed from the rotation and translation is absorbed by the system by the spontaneous creation of a singularity pair which is marked as (B) in the figures and can be seen in the space-time plots of Figures 7.10 to 7.13.

The system then relaxes further, the newly created pair move apart until local energy is

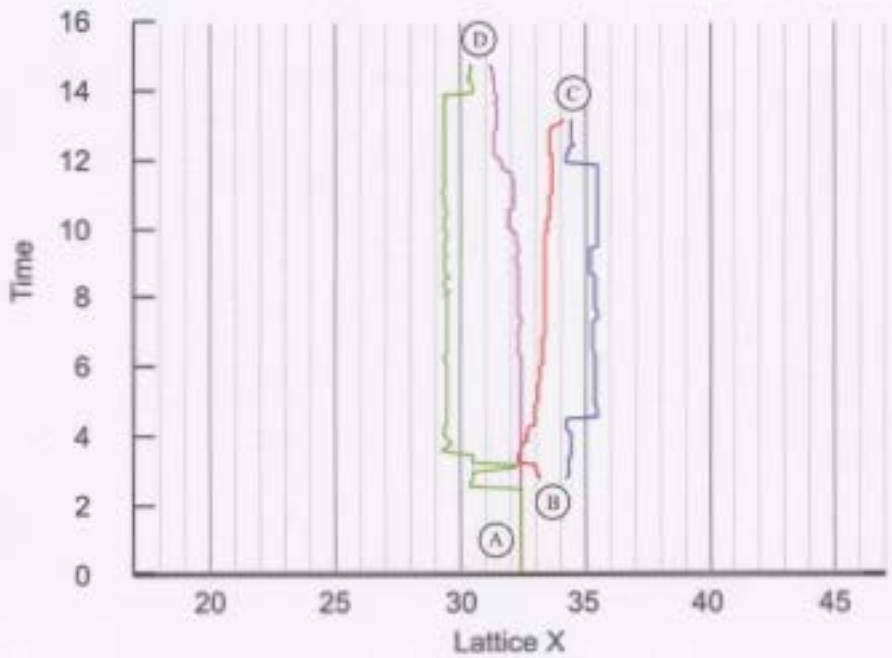


Figure 7.11: LatticeX-Time Plot.

low enough to allow them to move together and annihilate at Ⓒ. The Type +1 of the initial pair, after rotating into a low energy singularity moves toward the initial Type -1 singularity and also annihilates at Ⓓ.

From these plots, we can see that there is a distinct difference in the motion of the initial singularities as they move from unit cell to unit cell. After rotations, the Type -1 singularity is at a higher energy level than the Type +1. This corresponds to a rotation of 0 or π in Figure 7.3. The differences in energy alter the depths of the energy wells which leads to more punctuated or fluid motion.

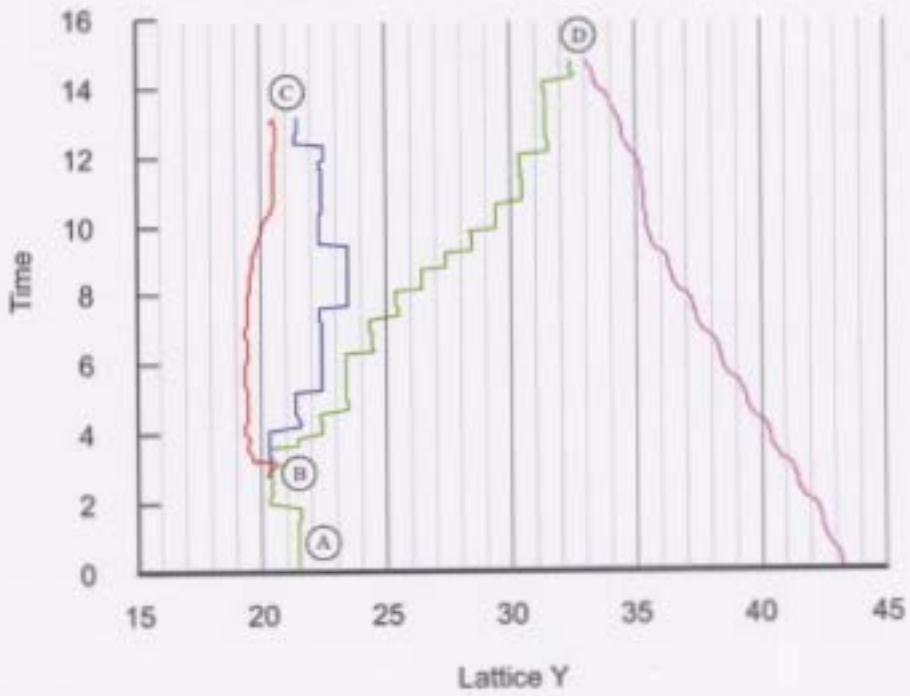


Figure 7.12: LatticeY-Time Plot.

7.2 Bulk Structures

In this example we simulate a 16 layer magnetic system in which the spins interact through an isotropic exchange interaction. This describes an important model of bulk ferromagnetic systems.

7.2.1 Simulation Parameters

This simulation was carried out with exchange interaction set as $J = 1.0$, damping of 2.0, periodic boundary conditions in the X, Y and Z directions and random initial spin directions

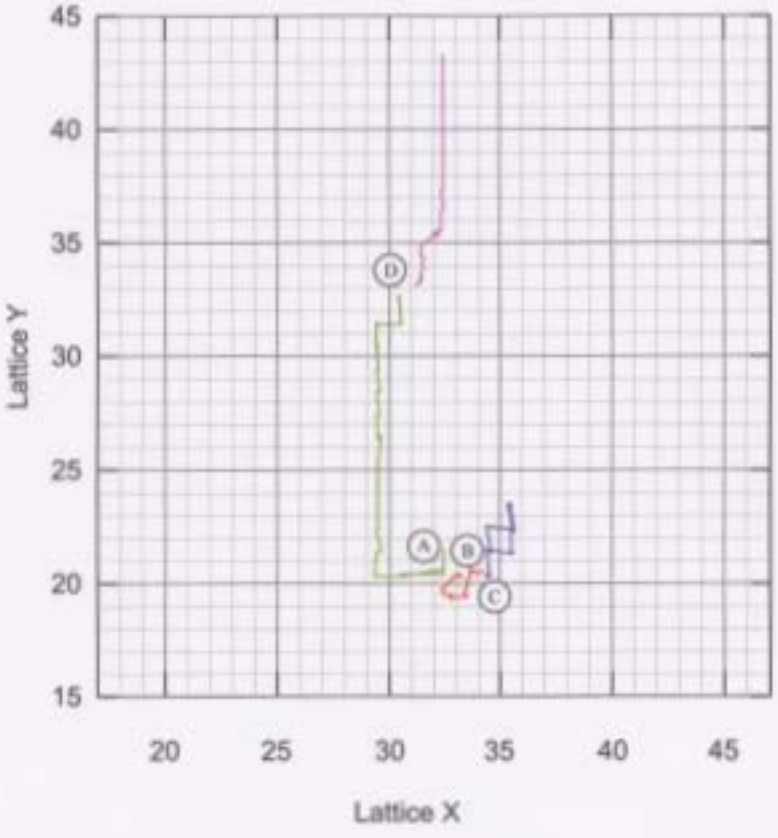


Figure 7.13: LatticeX-LatticeY Plot.

with spins of unit length. No thermal noise, dipolar interaction, anisotropy or external fields were present. Simulation states were sampled every 10 time units up to 5000 time units with an adaptive tolerance of 0.005.

The command which generated the output was:

```
./magsim -J 1 -g 0 -S 981932 -N 16 -n 16
-T 5000 -d 10 -o 0.005 -p 0 -pbjz
```

which specifies the exchange (`-J 1`) and dipolar (`-g 0`) strengths, provides a seed for

the random number generators (`-S 981932`), a layer dimension of 16x16 (`-N 16`) with 16 layers (`-n 16`), maximum runtime of 5000 (`-T 5000`) sampling every 10 (`-d 10`) with an adaptive tolerance of 0.005 (`-o 0.005`), random initial spin orientation (`-p 0`) and periodic boundary conditions in the z direction for the exchange interaction (`-pbjz`).

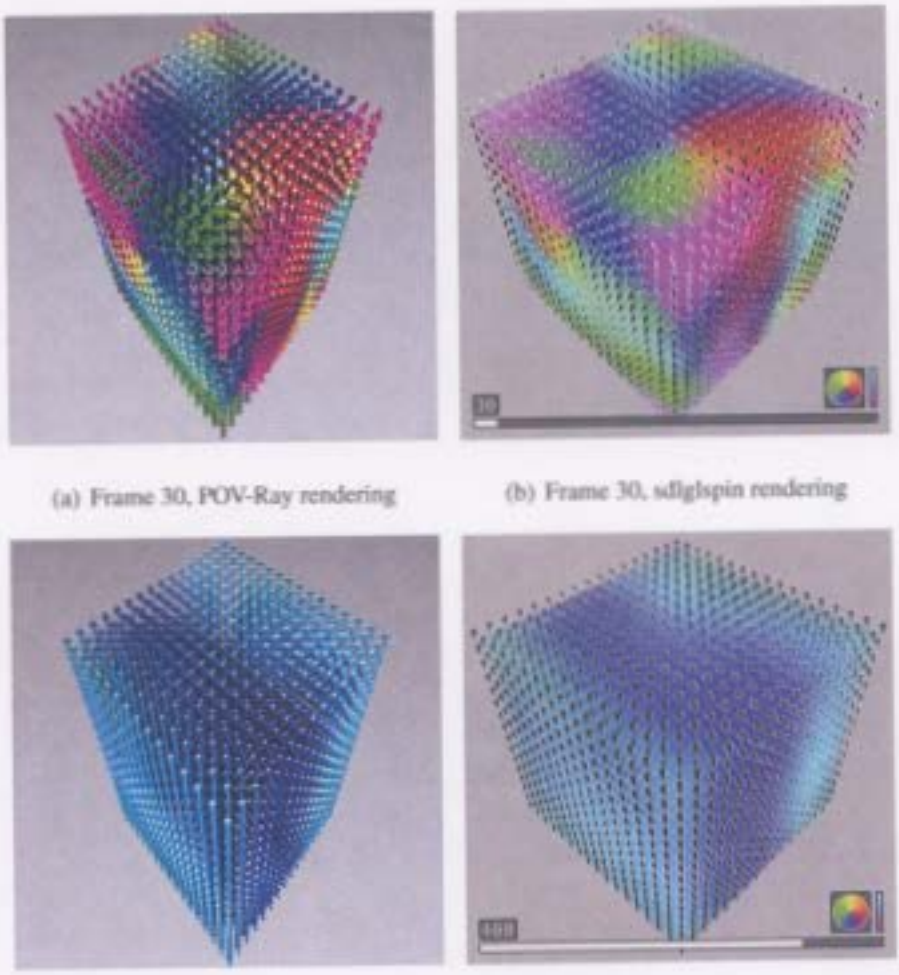
7.2.2 Results

This simulation took marginally less than 10 seconds to complete with an additional 5 seconds to write to the datafile. The multilayered nature of the system introduces new problems to overcome in terms of visualization and analysis. We have adapted both our SGL/OpenGL and POV-Ray rendering tools to display multiple layers as depicted in Figure 7.14. The media accompanying this thesis contains both videos rendered using these two programs and the datafile which the simulation generated.

7.2.3 Discussion

The problem with visualizing layered data is that the outer data obstructs the view of the inner data which is clear in the sample images. Both tools have the ability to turn off the rendering of certain layers but we have not yet developed a way to identify and isolate regions of significance in the relaxation process.

Another problem with visualizing is the time it can take to render the data. This simulation runs in approximately 10 seconds but the POV-Ray rendering of the 500 frames of data took 9 hours. Similar rendering using sdlglspin, when only rendering to screen, takes less than a minute to present all the data but capturing screenshots to encode and write to files increases the rendering time to 10 minutes.



(a) Frame 30, POV-Ray rendering

(b) Frame 30, sdlglspin rendering

(c) Frame 400, POV-Ray rendering

(d) Frame 400, sdlglspin rendering

Figure 7.14: Sample renderings of multilayer data using POV-Ray and sdlglspin

The differences in visualization times depends on the quality of the output. POV-Ray can generate very high quality images with proper light calculations and antialiasing via it's ray-tracing engine. However, this higher quality is computationally expensive and the rendering program does not make use of accelerated hardware on the video card. The other method, using sdlglspin, is of lower quality but has much faster render times. It does not make use of antialiasing and face illumination is based solely on dot product between the surface normal and light direction, that is, intervening objects do not cast shadows.

This inability to easily view the interior of a volume of data is a shortcoming of our visualization tools but we will leave improvements as an area of future research and development.

7.3 Non-Square Lattices

A further extension of the simulation is to consider non-square lattices. Adapting the simulation code to work with variable shaped lattices has proved to be trivial. As an example, the difference between square lattice simulation code and triangular or hexagonal lattices is less than 10 lines of code in our 10000 line project. This modification allows us, and other researchers, to examine a much wider range of systems. For this demonstration, we will look at a honeycomb mesh, which is a triangular lattice with specifically placed vacancies, and probe the phase space.

7.3.1 Simulation Parameters

This simulation was carried out in the simulation's realtime mode. This sets up special parameters to make realtime interaction easier such as imposing a maximum on the adaptive timestep, disabling reporting to file and setting the maximum simulated time to a very large value. These parameters make sure that the simulation neither completes in a small amount of time nor writes a very large data file. The command line used to start the simulation was:

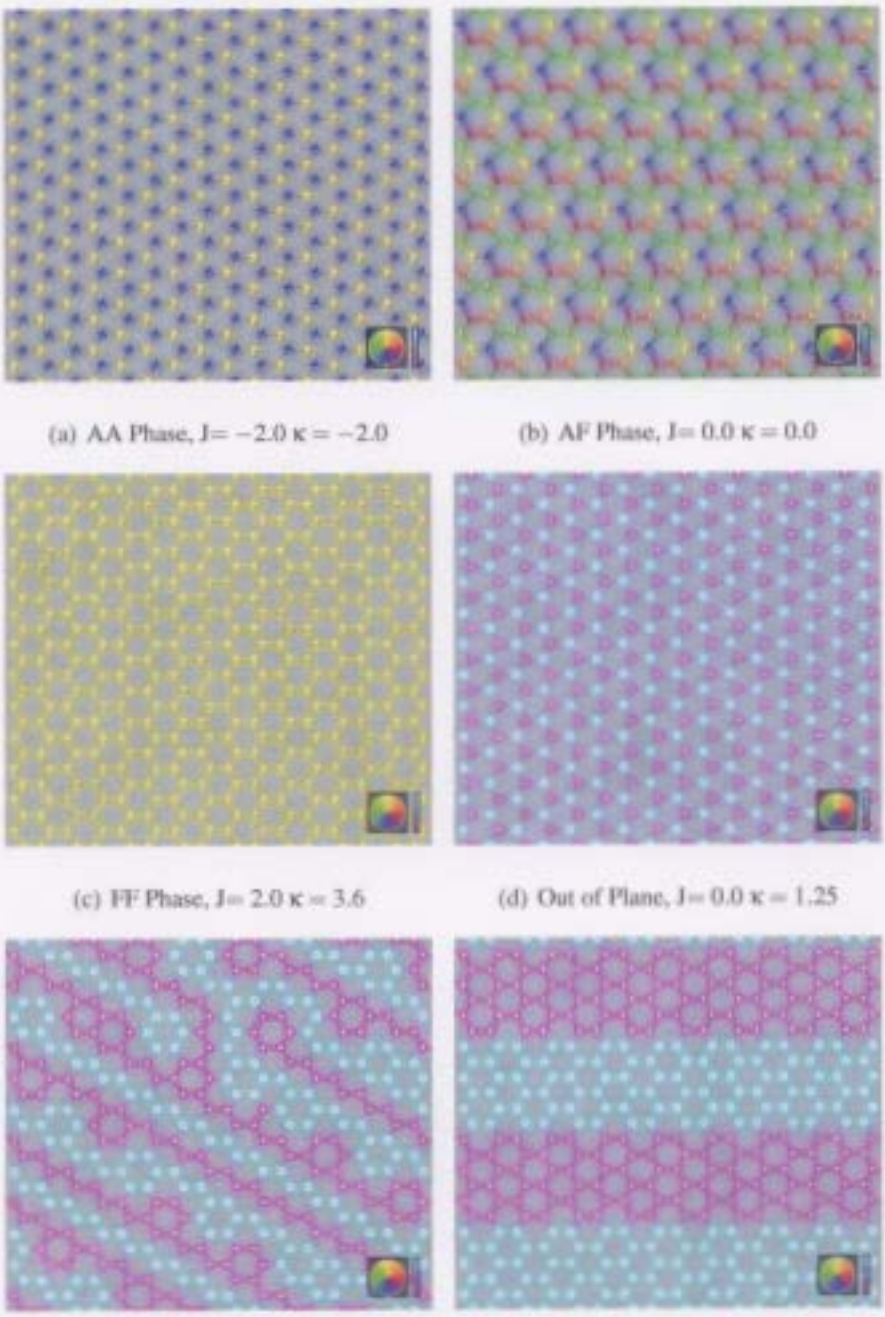
```
./magsim -tri -p 52 -E 0.1 -P_flat -N 24 -fftw -rt
```

which specifies a triangular lattice (`-tri`), with honeycombed vacancies as the initial position (`-p 52`), starting with a small amount of thermal fluctuations (`-E 0.1`), using precomputed dipolar interaction matrices for triangular lattices from the proper directory (`-P trilat`), grid dimension of 24×24 (`-N 24`), making use of the fftw library for FFT calculations (`-fftw`) since our grid dimension is not a power of 2 and running with custom realtime parameters (`-rt`) for convenience.

In conjunction with the simulation code, the shared memory programs were also used to export simulation states and modify simulation parameters from a remote workstation. This fast, realtime interaction with the simulation allowed us to explore the phase space for this lattice type.

7.3.2 Results

Throughout this simulation, the dipolar interaction strength was fixed at 0.5, no external fields were present and thermal perturbations were kept very low to facilitate dynamics. The only parameters varied were the strength of the exchange interaction and surface anisotropy. Figures 7.15 depicts distinct configurations for a honeycomb system.



(a) AA Phase, J= −2.0 κ = −2.0

(b) AF Phase, J= 0.0 κ = 0.0

(c) FF Phase, J= 2.0 κ = 3.6

(d) Out of Plane, J= 0.0 κ = 1.25

(e) Thin Stripes, J= 1.5 κ = 3.6

(f) Thick Stripes, J= 3.5 κ = 5.0

Figure 7.15: Honeycomb Simulation Snapshots (g= 0.5)

7.3.3 Discussion

Figure 7.15 shows the rich phase space in a honeycomb lattice. By varying surface anisotropy and the exchange strength we can identify 4 phases (AA, AF, FF and out of plane). The out of plane phase can be further subdivided into striped and non-striped and the stripes vary in characteristic width for positive J.

As stated, the code modifications to achieve non-square lattices were very minor, merely 10 lines for the simulation. This was also true for the modifications required in the visualization programs however we made no changes to our underlying libSpin. We have not adapted it to triangular or honeycomb lattices. If the datastream contains per-site fields then we can calculate energies otherwise the code will attempt to determine site energies assuming a square lattice. This is also true for topological singularities as we have not evaluated the Poincaré Index code for a triangular unit cell. Future work could involve abstracting the Poincaré calculations and implementing it for arbitrary lattice types.

7.4 Conclusion

We have presented several very different uses of our software package from understanding the dynamics in a single simulation to probing the phase space for given configuration and parameters. In presenting these demonstrations we have relied both on visualization of the system itself and extracting more abstract information such as the location of topological singularities. We believe that these examples clearly establish both the strength and flexibility of the code that was created for this thesis.

Chapter 8

Summary

This thesis describes a numerical integration scheme for the LLG equation that can be adapted to multiprocessor, shared memory architectures. A set of sophisticated visualization tools and parallelization methods have also been developed as part of this project and are described in the thesis. These parallelization methods, utilizing shared memory, allow remote visualization of running simulations at no computational cost to the simulation. These parallelization methods also allow zero cost interaction with the simulation which, when combined with the visualizations and efficient integration techniques, creates a complete suite which can be used to explore the dynamics of multilayered magnetic systems.

The principle components that were developed as part of this project included

1. The formulation of the LLG equation in terms of spin rotations (as opposed to the more conventional Cartesian approach) using quaternion algebra and an adaptive timestep numerical integrator.

2. The development of a set of shared memory routines (shm_utils) that are used to

calculate fields, update spin positions and allow interaction between the running program and remote computers. These utilities are more efficient, scalable and versatile than the standard parallelization packages (MPI, OpenMP) in the context of the particular class of problems considered in this thesis.

3. The shm_utils were also used to develop routines that calculate the most effective way to distribute a given set of problems across multiple processors in order to minimize the total execution time. This dynamic optimization technique implicitly considered computational topology and availability of resources.

4. The visualization techniques built on libSpin, SDL, OpenGL and POV-Ray provided methods of viewing both running and recorded simulations immediately at high framerates using accelerated graphics or afterward with raytraced images and videos.

5. The shm_utils contain a feature that allow remote modification of local memory. This allows simulation parameters to be modified at runtime without communication overhead.

We have demonstrated the capabilities of the code by applying it to study three problems of interest, namely relaxation processes in single layer films, magnetic ordering in bulk systems and magnetic phases in two dimensional honeycomb lattices. The results obtained are interesting and suggest possible avenues of research in conjunction with this code.

Research projects which are being actively investigated using this code include spinwaves in magnetic multilayers and striped phases in multilayer geometries. These two topics also demonstrate the versatility of the code. When investigating spinwaves at zero temperature, we must run the simulation through millions of timesteps at zero damping

without introducing or loosing energy through numerical drift. The quaternion based rotations and adaptive timesteps have allowed us to achieve this goal. Striped multilayer geometries were discovered by relaxing the tolerance and interactively exploring large sections of phase space which was possible again through the use of the adaptive timestep and the unique features of the parallel routines.

Future research is planned to explore the memory model used in shm_utils to create a generic high performance shared memory library to be used in research settings and automatic creation of visualization on hardware ranging from commodity desktop PCs to immersive visualization labs with haptic feedback.

Appendix A

Static Energy Calculations

In this appendix we will present the full derivation for the energy of a spin system with periodic boundary conditions.

The calculation is started by first writing the Hamiltonian as

$$\mathcal{H} = g\frac{1}{2}{\sum_{\vec{R}_i,\vec{R}_j}}' \left[\frac{\vec{\sigma}(\vec{R}_i)\cdot\vec{\sigma}(\vec{R}_j)}{|\vec{R}_{ij}|^3} - 3\frac{(\vec{\sigma}(\vec{R}_i)\cdot\vec{R}_{ij})(\vec{\sigma}(\vec{R}_j)\cdot\vec{R}_{ij})}{|\vec{R}_{ij}|^5} \right] \tag{A.1}$$

$$-g\left(J \sum_{<\vec{R}_i,\vec{R}_j>} \vec{\sigma}(\vec{R}_i)\cdot\vec{\sigma}(\vec{R}_j) + \sum_{\vec{R}_i} \vec{H}\cdot\vec{\sigma}(\vec{R}_i) + \sum_{\vec{R}_i} \kappa\sigma^z(\vec{R}_i)^2 \right), \tag{A.2}$$

where $\vec{R}_i$ and $\vec{R}_j$ are lattice positions, $\vec{R}_{ij}$ is the displacement between these positions, $\vec{\sigma}(\vec{R}_i)$ is the magnetic moment at a lattice site $\vec{R}_i$, $\vec{H}$ is an external applied field, g, J and κ are the strengths of the dipolar interaction, exchange interaction and surface anisotropy, the prime on the summation excludes the term when $\vec{R}_i$ is equal to $\vec{R}_j$ and the angled brackets indicate sums over nearest neighbours.

A.1 Dipole-Dipole Energy

Next we define a lattice of spins of dimensions A in the $\hat{x}$ direction and B in the $\hat{y}$ direction. Examining the average dipole-dipole energy from a spin in the lattice, we have

$$E_{dd} = \frac{1}{AB}\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{m=1}^{A}\sum_{n=1}^{B}{\sum_{\vec{G}}}' \frac{\vec{\sigma}(\vec{G}+m\hat{x}+n\hat{y})\cdot\vec{\sigma}(i\hat{x}+j\hat{y})}{|\vec{R}+\vec{G}|^3} - 3\frac{[\vec{\sigma}(\vec{G}+m\hat{x}+n\hat{y})\cdot(\vec{R}+\vec{G})][\vec{\sigma}(i\hat{x}+j\hat{y})\cdot(\vec{R}+\vec{G})]}{|\vec{R}+\vec{G}|^5}, \tag{A.3}$$

where $\vec{G}$ represents all replicated lattices and is defined as

$$\vec{G} = g_1 A\hat{x} + g_2 B\hat{y}, \tag{A.4}$$

with integers g_1 and g_2. The prime reminds us to exclude the terms when $\vec{G}$ and $\vec{R}$ sum to zero, $\vec{R} = \vec{R}_{ij} - \vec{R}_{mn}$ with $\vec{R}_{xy} = x\hat{x} + y\hat{y}$. We break Equation A.3 into 4 parts as follows

$$AB\,E_{dd} = \sum_{i=1}^{A}\sum_{j=1}^{B}{\sum_{\vec{G}}}' \frac{[\vec{\sigma}(\vec{G}+i\hat{x}+j\hat{y})]^2}{|\vec{G}|^3} \tag{A.5}$$

$$-\sum_{i=1}^{A}\sum_{j=1}^{B}{\sum_{\vec{G}}}' 3\frac{\vec{\sigma}(\vec{G}+i\hat{x}+j\hat{y})\cdot\vec{G}}{|\vec{G}|^5} \tag{A.6}$$

$$+\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{m=1}^{A}\sum_{n=1}^{B}{\sum_{\vec{G}}}' \frac{\vec{\sigma}(\vec{G}+m\hat{x}+n\hat{y})\cdot\vec{\sigma}(i\hat{x}+j\hat{y})}{|\vec{R}+\vec{G}|^3} \tag{A.7}$$

$$-\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{m=1}^{A}\sum_{n=1}^{B}{\sum_{\vec{G}}}' 3\frac{[\vec{\sigma}(\vec{G}+m\hat{x}+n\hat{y})\cdot(\vec{R}+\vec{G})][\vec{\sigma}(i\hat{x}+j\hat{y})\cdot(\vec{R}+\vec{G})]}{|\vec{G}|^5} \tag{A.8}$$

which represents a magnetic moment interacting with it's replicas in the first two terms (isotropic and anisotropic parts) which we will refer to as the self interaction, and all other interactions are represented in the second two. Here the prime on the summations applies the restriction $m \neq j$ if $n = i$.

A.1.1 Isotropic Self Interaction

We will deal with each term in turn, first we rewrite the summation in A.5 as

$$\sum_{i=1}^{A}\sum_{j=1}^{B}[\vec{\sigma}(i\hat{x}+j\hat{y})]^2\sum_{\vec{G}}{}'\frac{1}{|\vec{G}|^3} \tag{A.9}$$

since

$$\vec{\sigma}(i\hat{x}+j\hat{y}) = \vec{\sigma}(\vec{G}+i\hat{x}+j\hat{y}). \tag{A.10}$$

Using the identity based on the integral representation of the gamma function

$$\frac{1}{x^{2s}} = \frac{1}{\Gamma(s)}\int_0^\infty t^{s-1}e^{-x^2t}dt, \tag{A.11}$$

we express Equation A.9 in the form

$$\sum_{i=1}^{A}\sum_{j=1}^{B}\vec{\sigma}(i\hat{x}+j\hat{y})^2\sum_{\vec{G}\neq 0}\frac{1}{\Gamma(\frac{3}{2})}\int_0^\infty t^{\frac{3}{2}-1}e^{-|\vec{G}|^2t}dt. \tag{A.12}$$

A change in variable $t=\rho^2$, $dt = 2\rho\,d\rho$ and evaluation of the Gamma function yields

$$\sum_{i=1}^{A}\sum_{j=1}^{B}\vec{\sigma}(i\hat{x}+j\hat{y})^2\frac{4}{\sqrt{\pi}}\sum_{\vec{G}\neq 0}\int_0^\infty \rho^2 e^{-|\vec{G}|^2\rho^2}d\rho. \tag{A.13}$$

We now split the integral into two ranges, 0 to η and η to ∞ with η being an arbitrary value

$$\sum_{i=1}^{A}\sum_{j=1}^{B}\vec{\sigma}(i\hat{x}+j\hat{y})^2\frac{4}{\sqrt{\pi}}\sum_{\vec{G}\neq 0}\left(\int_0^\eta \rho^2 e^{-|\vec{G}|^2\rho^2}d\rho + \int_\eta^\infty \rho^2 e^{-|\vec{G}|^2\rho^2}d\rho\right). \tag{A.14}$$

Each term of this expression must be dealt with separately, we start with the second expression. We will show its relation to the Incomplete Gamma Function, starting with the general definition

$$\Gamma(a,z) = \int_z^\infty t^{a-1}e^{-t}dt, \tag{A.15}$$

change the variable of integration $t = X^2\rho^2$, $dt = 2\rho X^2 d\rho$

$$\Gamma(a,z) = \int_{X^2\rho^2=z}^{\infty} (X^2\rho^2)^{a-1} e^{-X^2\rho^2} 2\rho X^2 d\rho, \tag{A.16}$$

and evaluate at $a = \frac{3}{2}$, $z = \eta^2 X^2$

$$\Gamma(\frac{3}{2},\eta^2X^2) = \int_{\eta}^{\infty} (X^2\rho^2)^{\frac{1}{2}} e^{-X^2\rho^2} 2\rho X^2 d\rho. \tag{A.17}$$

Since X is independent of ρ, we can take it outside of the integral and re-express the Incomplete Gamma Function after integration by parts

$$(\frac{3}{2}-1)\Gamma(\frac{3}{2}-1,\eta^2X^2) + (\eta^2X^2)^{\frac{3}{2}-1} e^{-\eta^2X^2} = 2X^3 \int_{\eta}^{\infty} \rho^2 e^{-X^2\rho^2} d\rho. \tag{A.18}$$

Evaluating the Incomplete Gamma Function at $a = \frac{1}{2}$ yields $\Gamma(\frac{1}{2},x) = \sqrt{\pi}\,\mathrm{erfc}(\sqrt{x})$ where erfc is the complementary error function and so it follows that

$$\frac{1}{X^3}\left(\frac{\sqrt{\pi}}{4}\,\mathrm{erfc}(\eta X) + \frac{1}{2}\eta X e^{-\eta^2X^2}\right) = \int_{\eta}^{\infty} \rho^2 e^{-X^2\rho^2} d\rho, \tag{A.19}$$

which is the second term of Equation A.14 when $X = |\vec{G}|$.

The first term of Equation A.14 is treated in a different manner. The second, when summed over $\vec{G} \neq 0$, results in a rapidly converging series but the same method applied to the lower range of integration will not work. Here we must make use of the reciprocal lattice space to achieve rapid convergence. We start by drawing the summation over $\vec{G} \neq 0$ into the sum of terms and write

$$\sum_{\vec{G}\neq 0} \int_0^{\eta} \rho^2 e^{-|\vec{G}|^2\rho^2} d\rho. \tag{A.20}$$

Which can be expanded to the full sum over $\vec{G}$ by subtracting the term with $\vec{G} = 0$

$$\sum_{\vec{G}} \int_0^{\eta} \rho^2 e^{-|\vec{G}|^2\rho^2} d\rho - \int_0^{\eta} \rho^2 d\rho. \tag{A.21}$$

Changing variables $\rho = \sqrt{t}$, $d\rho = \frac{1}{2\sqrt{t}}dt$ and evaluating the second integral we get

$$\sum_{\vec{G}} \int_0^{\eta^2} te^{-|\vec{G}|^2 t} \frac{1}{2\sqrt{t}} dt - \frac{\eta^3}{3}. \tag{A.22}$$

Using the Poisson summation formula for 2D space, this is transformed into

$$\frac{\pi}{2AB} \int_0^{\eta^2} \frac{1}{\sqrt{t}} \sum_{\vec{Q}} e^{-\frac{\vec{Q}^2}{4t}} dt - \frac{\eta^3}{3}, \tag{A.23}$$

where $\vec{Q} = 2\pi(\frac{k_1}{A}\hat{x} + \frac{k_2}{B}\hat{y})$ with integers k_1, k_2. Next we change variables $t = \frac{1}{\rho^2}$, $dt = \frac{-2}{\rho^3}d\rho$ and write

$$\frac{\pi}{AB} \sum_{\vec{Q}} \int_{\frac{1}{\eta}}^{\infty} \frac{1}{\rho^2} e^{-\frac{\vec{Q}^2}{4}\rho^2} d\rho - \frac{\eta^3}{3}. \tag{A.24}$$

We will now start from the definition of the Incomplete Gamma Function and work backwards to get Equation A.24,

$$\Gamma(a,z) = \int_z^{\infty} t^{a-1} e^{-t} dt \tag{A.25}$$

Changing variables $t = \left(\frac{\vec{Q}}{2}\rho\right)^2$, $dt = \frac{\vec{Q}^2}{2}\rho\, d\rho$ gives

$$\Gamma(a,z) = \int_{\sqrt{z\frac{4}{\vec{Q}^2}}}^{\infty} \left(\frac{\vec{Q}}{2}\rho\right)^{2a-2} e^{-\left(\frac{\vec{Q}}{2}\rho\right)^2} \frac{\vec{Q}^2}{2}\rho\, d\rho, \tag{A.26}$$

and selecting $a = -\frac{1}{2}$ and $z = \left(\frac{\vec{Q}}{2\eta}\right)^2$ evaluates to

$$\Gamma\left(-\frac{1}{2}, \left(\frac{\vec{Q}}{2\eta}\right)^2\right) = \frac{4}{\vec{Q}} \int_{\frac{1}{\eta}}^{\infty} \frac{1}{\rho^2} e^{-\left(\frac{\vec{Q}}{2}\rho\right)^2} d\rho. \tag{A.27}$$

Integrating the Incomplete Gamma Function by parts gives

$$\Gamma(a+1,z) = a\Gamma(a,z) + z^a e^{-z}, \tag{A.28}$$

inserting A.27 into the first term on the right yields

$$\Gamma(\frac{1}{2},\left(\frac{\vec{Q}}{2\eta}\right)^2)-\left(\frac{\vec{Q}}{2\eta}\right)^{-1}e^{\left(\frac{\vec{Q}}{2\eta}\right)^2}=-\frac{1}{2}\Gamma(-\frac{1}{2},\left(\frac{\vec{Q}}{2\eta}\right)^2). \tag{A.29}$$

Evaluating the Incomplete Gamma Functions result in

$$\int_{\frac{1}{\eta}}^{\infty}\frac{1}{\rho^2}e^{-\left(\frac{\vec{Q}}{2}\rho\right)^2}d\rho=\eta e^{-\left(\frac{\vec{Q}}{2\eta}\right)^2}-\frac{\vec{Q}}{2}\sqrt{\pi}\,\mathrm{erfc}\left(\frac{\vec{Q}}{2\eta}\right), \tag{A.30}$$

which is the integral in A.24. We can now use Equations A.30 and A.19 to rewrite Equation A.14 and express the summation in A.5 as two rapidly converging series, one in reciprocal space and the other in real space

$$\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{\vec{G}\neq 0}\frac{(\vec{\sigma}(\vec{G}+n\hat{x}))^2}{|\vec{G}|^3}=\sum_{i=1}^{A}\sum_{j=1}^{B}\vec{\sigma}(i\hat{x}+j\hat{y})^2\frac{4}{\sqrt{\pi}}\left[\frac{\pi}{AB}\sum_{\vec{Q}}\left((\eta e^{-\left(\frac{\vec{Q}}{2\eta}\right)^2}-\frac{\vec{Q}}{2}\,\mathrm{erfc}(\frac{\vec{Q}}{2\eta})\right)\right.$$
$$\left.-\frac{\eta^3}{3}+\sum_{\vec{G}\neq 0}\frac{1}{X^3}\left(\frac{\sqrt{\pi}}{4}\,\mathrm{erfc}(\eta|\vec{G}|)+\frac{1}{2}\eta|\vec{G}|e^{-\eta^2|\vec{G}|^2}\right)\right] \tag{A.31}$$

A.1.2 Anisotropic Self Interaction

The next term to expand, found in A.6, follows a similar pattern to the last. We must first extract the $\vec{\sigma}$ from the summation over $\vec{G}\neq 0$. This is accomplished by first recalling Equation A.10 and then performing the following

$$-3\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{\vec{G}\neq 0}\frac{(\vec{\sigma}(i\hat{x}+j\hat{y})\cdot\vec{G})^2}{|\vec{G}|^5}=-3\sum_{i=1}^{A}\sum_{j=1}^{B}(\vec{\sigma}(i\hat{x}+j\hat{y})\cdot\vec{\nabla}_{\vec{\alpha}})^2\sum_{\vec{G}\neq 0}\frac{e^{\vec{\alpha}\cdot\vec{G}}}{|\vec{G}|^5}|_{\vec{\alpha}=0}. \tag{A.32}$$

The numerator on the right will disapear when α is set to zero, it's only purpose is to help match up components of $\vec{G}$ with $\vec{\sigma}$ and so we can set it aside and write

$$-3\sum_{i=1}^{A}\sum_{j=1}^{B}(\vec{\sigma}(i\hat{x}+j\hat{y})\cdot\vec{\nabla}_{\vec{\alpha}})^2(\sum_{\vec{G}\neq 0}\frac{1}{|\vec{G}|^5})(\sum_{\vec{G}\neq 0}e^{\vec{\alpha}\cdot\vec{G}})|_{\vec{\alpha}=0}, \tag{A.33}$$

and only deal with the $|\vec{G}|^{-5}$ term. As before, we will start with the identity based on the integral representation of the gamma function (A.11) with $a = \frac{5}{2}$ and perform the change in variables $t = \rho^2$, $dt = 2\rho d\rho$ to get

$$-3\sum_{i=1}^{A}\sum_{j=1}^{B}(\vec{\sigma}(i\hat{x}+j\hat{y})\cdot\vec{\nabla}_{\vec{\alpha}})^2(\sum_{\vec{G}\neq 0}\frac{3\sqrt{\pi}}{2}\int_0^{\infty}\rho^4 e^{-|\vec{G}|^2\rho^2}d\rho)(\sum_{\vec{G}\neq 0}e^{\vec{\alpha}\cdot\vec{G}})|_{\vec{\alpha}=0}. \tag{A.34}$$

Splitting the integral at an arbitrary η gives

$$\frac{-9\sqrt{\pi}}{2}\sum_{i=1}^{A}\sum_{j=1}^{B}(\vec{\sigma}(i\hat{x}+j\hat{y})\cdot\vec{\nabla}_{\vec{\alpha}})^2(\sum_{\vec{G}\neq 0}\int_0^{\eta}\rho^4 e^{-|\vec{G}|^2\rho^2}d\rho+\int_{\eta}^{\infty}\rho^4 e^{-|\vec{G}|^2\rho^2}d\rho)(\sum_{\vec{G}\neq 0}e^{\vec{\alpha}\cdot\vec{G}})|_{\vec{\alpha}=0}. \tag{A.35}$$

We will now start with the definition of the Incomplete Gamma Function (A.15), change variables $t = X^2\rho^2$, $dt = 2\rho X^2 d\rho$ and set $a = \frac{5}{2}$, $z = \eta^2 X^2$ giving

$$\Gamma(\frac{5}{2},\eta^2X^2) = 2X^5\int_{\eta}^{\infty}\rho^4 e^{-X^2\rho^2}d\rho, \tag{A.36}$$

which can be evaluated by integrating by parts twice and so setting $X = |\vec{G}|$ we can express the upper integral range as

$$\int_{\eta}^{\infty}\rho^4 e^{-|\vec{G}|^2\rho^2}d\rho = \frac{1}{2|\vec{G}|^5}\left(\frac{3}{4}\sqrt{\pi}\,\mathrm{erfc}(\eta|\vec{G}|)+(\frac{3}{2}\eta|\vec{G}|+\eta^3|\vec{G}|^3)e^{-\eta^2|\vec{G}|^2}\right). \tag{A.37}$$

We will now follow the procedure to evaluate the lower range of the integral in a similar manner as the method which starts at Equation A.20. We will include the summation over $\vec{G} \neq 0$ and transform it into a full summation by subtracting the zero term

$$\sum_{\vec{G}}\int_0^{\eta}\rho^4 e^{-\rho^2|\vec{G}|^2}d\rho - \int_0^{\eta}\rho^4 d\rho. \tag{A.38}$$

Changing variables $\rho = \sqrt{t}$, $d\rho = \frac{1}{2\sqrt{t}}$ and applying the Poisson Summation over 2 dimensions transforms the above to

$$\frac{\pi}{2AB}\int_0^{\eta^2}\sqrt{t}\sum_{\vec{Q}}e^{-\frac{\vec{Q}^2}{4t}}dt - \frac{\eta^5}{5}, \tag{A.39}$$

where $\vec{Q}$ holds the same meaning as above. One more change in variables $t = \rho^{-2}$, $dt = -2\rho^{-3}d\rho$ results in

$$\frac{\pi}{AB}\int_{\frac{1}{\eta}}^{\infty}\frac{1}{\rho^4}\sum_{\vec{Q}}e^{-\rho^2\vec{Q}^2 4^{-1}}d\rho - \frac{\eta^5}{5}, \tag{A.40}$$

which we will show to be related to the Incomplete Gamma Function. Starting with Equation A.25, we will change variables $t = \frac{1}{4}\vec{Q}^2\rho^2$, $dt = \frac{1}{2}\vec{Q}^2\rho d\rho$ and set $a = -\frac{3}{2}$, $z = \left(\frac{\vec{Q}}{2\eta}\right)^2$ to get

$$\Gamma(-\frac{3}{2}, \left(\frac{\vec{Q}}{2\eta}\right)^2) = \int_{\frac{1}{\eta}}^{\infty}\frac{2^4}{\vec{Q}^3}\frac{1}{\rho^4}e^{-\frac{1}{4}\vec{Q}^2\rho^2}d\rho. \tag{A.41}$$

Integrating Equation A.25 by parts twice lets us evaluate the left term as

$$\frac{4}{3}\sqrt{\pi}\,\mathrm{erfc}(\frac{\vec{Q}}{2\eta}) + \left(\frac{8\eta}{3\vec{Q}} + \frac{2^4\eta}{3\vec{Q}^3}\right)e^{-\left(\frac{\vec{Q}}{2\eta}\right)^2} = \frac{2^4}{\vec{Q}^3}\int_{\frac{1}{\eta}}^{\infty}\frac{1}{\rho^4}e^{-\frac{1}{4}\vec{Q}^2\rho^2}d\rho, \tag{A.42}$$

and so we can rewrite Equation A.38 as

$$\sum_{\vec{Q}}\frac{\pi}{AB}\frac{\vec{Q}^3}{2^4}\left(\frac{4\sqrt{\pi}}{3}\,\mathrm{erfc}(\frac{\vec{Q}}{2\eta}) + \left(\frac{8\eta}{3\vec{Q}} + \frac{2^4\eta}{3\vec{Q}^3}\right)e^{-\left(\frac{\vec{Q}}{2\eta}\right)^2}\right) - \frac{\eta^5}{5}. \tag{A.43}$$

This result combined with Equation A.37 allows us to express Equation A.6 as the sum of two rapidly converging series

$$\begin{aligned}
&-3\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{G\vec{\neq}0}\frac{(\vec{\sigma}(\vec{G}+i\hat{x}+j\hat{y})\cdot\vec{G})^2}{|\vec{G}|^5} = \\
&-3\sum_{i=1}^{A}\sum_{j=1}^{B}(\vec{\sigma}(i\hat{x}+j\hat{y})\cdot\vec{\nabla}_{\vec{\alpha}})^2\Bigg\{\sum_{\vec{Q}}\frac{\pi}{AB}\frac{\vec{Q}^3}{2^4}\left[\frac{4\sqrt{\pi}}{3}\,\mathrm{erfc}(\frac{\vec{Q}}{2\eta}) + \left(\frac{8\eta}{3\vec{Q}} + \frac{2^4\eta}{3\vec{Q}^3}\right)e^{-\left(\frac{\vec{Q}}{2\eta}\right)^2}\right] - \frac{\eta^5}{5} \\
&+\sum_{\vec{G}\neq 0}\frac{1}{2|\vec{G}|^5}\left(\frac{3}{4}\sqrt{\pi}\,\mathrm{erfc}(\eta|\vec{G}|) + (\frac{3}{2}\eta|\vec{G}| + \eta^3|\vec{G}|^3)e^{-\eta^2|\vec{G}|^2}\right)\Bigg\}\sum_{\vec{G}\neq 0}e^{\vec{\alpha}\cdot\vec{G}}|_{\vec{\alpha}=0}.
\end{aligned} \tag{A.44}$$

A.1.3 Isotropic General Interaction

The third term of our original equation, A.7, can be expressed immediately as

$$\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{m=1}^{A}\sum_{n=1}^{B}\vec{\sigma}(i\hat{x}+j\hat{y})\vec{\sigma}(m\hat{x}+n\hat{y})\frac{4}{\sqrt{\pi}}\sum_{\vec{G}}\left(\int_0^{\eta}\rho^2 e^{-|\vec{G}+\vec{R}|^2\rho^2}d\rho+\int_{\eta}^{\infty}\rho^2 e^{-|\vec{G}+\vec{R}|^2\rho^2}d\rho\right) \tag{A.45}$$

by following the same steps as we took for our first term. The only differences are that this starts as a full sum over $\vec{G}$ and we have an $\vec{R}$ term in the exponent. The second integral can also be immediately re-expressed as a rapidly converging series by using Equation A.19

$$\frac{1}{|\vec{G}+\vec{R}|^3}\left(\frac{\sqrt{\pi}}{4}\operatorname{erfc}(\eta|\vec{G}+\vec{R}|)+\frac{1}{2}\eta|\vec{G}+\vec{R}|\,e^{-\eta^2|\vec{G}+\vec{R}|^2}\right)=\int_{\eta}^{\infty}\rho^2 e^{-|\vec{G}+\vec{R}|^2\rho^2}d\rho. \tag{A.46}$$

The first term, however, cannot be as quickly restated. The extra term in the exponent changes the evaluation of the Poisson Summation in 2 dimensions. For this term we will start at Equation A.22, suitably modified to include the extra term in the exponent and the full sum

$$\sum_{\vec{G}}\int_0^{\eta^2} te^{-|\vec{G}+\vec{R}|^2 t}\frac{1}{2\sqrt{t}}dt \tag{A.47}$$

and then apply the Poisson Summation in 2 dimensions

$$\sum_{\vec{R}}e^{-|\vec{R}+\vec{R}|^2 t}=\frac{\pi}{L^2 t}\sum_{Q}e^{i\vec{Q}\cdot\vec{R}}e^{-\frac{\vec{Q}^2}{4t}} \tag{A.48}$$

to get

$$\frac{1}{2}\frac{\pi}{AB}\sum_{\vec{Q}}\int_0^{\eta^2} t^{-\frac{1}{2}}e^{i\vec{Q}\cdot\vec{R}}e^{-\frac{\vec{Q}^2}{4t}}dt \tag{A.49}$$

which is Equation A.23 with an extra exponential and a missing $\frac{\eta^3}{3}$. We can skip to the end of the derivation for this expansion and write the expression similar to Equation A.31,

noting the differences, which is equal to Equation A.7

$$\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{m=1}^{A}\sum_{n=1}^{B}{}'\sum_{\vec{G}}\frac{\vec{\sigma}(n\hat{x}+\vec{G})\vec{\sigma}(m\hat{x})}{|\vec{R}+\vec{G}|^3} =$$
$$\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{m=1}^{A}\sum_{n=1}^{B}{}'\vec{\sigma}(n\hat{x})\vec{\sigma}(m\hat{x})\frac{4}{\sqrt{\pi}}\left[\frac{\pi}{AB}\sum_{\vec{Q}}e^{i\vec{Q}\cdot\vec{R}}\left((\eta e^{-\left(\frac{\vec{Q}}{2\eta}\right)^2}-\frac{\vec{Q}}{2}\,\mathrm{erfc}(\frac{\vec{Q}}{2\eta})\right)\right.$$
$$\left.+\sum_{\vec{G}}\frac{1}{|\vec{G}+\vec{R}|^3}\left(\frac{\sqrt{\pi}}{4}\,\mathrm{erfc}(\eta|\vec{G}+\vec{R}|)+\frac{1}{2}\eta|\vec{G}+\vec{R}|\;e^{-\eta^2|\vec{G}+\vec{R}|^2}\right)\right] \tag{A.50}$$

A.1.4 Anisotropic General Interaction

The last term in the dipolar expression, Equation A.8, can be expanded into two rapidly converging series in the same manner as those above. We start by extracting the magnetic moments from the summation over $\vec{G}$

$$-3\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{m=1}^{A}\sum_{n=1}^{B}{}'\sum_{\vec{G}}\frac{[\vec{\sigma}(\vec{G}+m\hat{x}+n\hat{y})\cdot(\vec{R}+\vec{G})][\vec{\sigma}(i\hat{x}+j\hat{y})\cdot(\vec{R}+\vec{G})]}{|\vec{R}+\vec{G}|^5} =$$
$$+3\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{m=1}^{A}\sum_{n=1}^{B}{}'(\vec{\sigma}(i\hat{x}+j\hat{y})\cdot\vec{\nabla}_\alpha)(\vec{\sigma}(m\hat{x}+n\hat{y})\cdot\vec{\nabla}_\alpha)\sum_{\vec{G}}\frac{e^{-i\vec{\alpha}\cdot(\vec{R}+\vec{G})}}{|\vec{R}+\vec{G}|^5}|_{\vec{\alpha}=0}. \tag{A.51}$$

Following steps similar to how we derive Equation A.33, we write

$$\frac{9\sqrt{\pi}}{2}\sum_{i=1}^{A}\sum_{j=1}^{B}\sum_{m=1}^{A}\sum_{n=1}^{B}{}'(\vec{\sigma}(i\hat{x}+j\hat{y})\cdot\vec{\nabla}_\alpha)(\vec{\sigma}(m\hat{x}+n\hat{y})\cdot\vec{\nabla}_\alpha)(\sum_{\vec{G}}\int_0^\infty\rho^4e^{-|\vec{R}+\vec{G}|^2\rho^2}d\rho)\sum_{\vec{G}}e^{-i\vec{\alpha}\cdot(\vec{R}+\vec{G})}|_{\vec{\alpha}=0}. \tag{A.52}$$

Splitting the integral into two ranges, we first convert the upper range into a rapidly converging series following Equation A.36

$$\sum_{\vec{G}}\int_\eta^\infty\rho^4e^{-|\vec{R}+\vec{G}|^2\rho^2}d\rho =$$
$$\sum_{\vec{G}}\frac{1}{2|\vec{R}+\vec{G}|^5}\left(\frac{3}{4}\sqrt{\pi}\,\mathrm{erfc}(\eta|\vec{R}+\vec{G}|)+(\frac{3}{2}\eta|\vec{R}+\vec{G}|+\eta^3|\vec{R}+\vec{G}|^3)e^{-\eta^2|\vec{R}+\vec{G}|^2}\right). \tag{A.53}$$

Following steps similar to those leading to A.43, we write

$$\sum_{\vec{G}} \int_0^{\eta} \rho^4 e^{-|\vec{R}+\vec{G}|^2\rho^2} d\rho =$$
$$\frac{\pi}{AB} \sum_{\vec{Q}} e^{i\vec{Q}\cdot\vec{R}} \frac{\vec{Q}^3}{2^4} \left(e^{-\left(\frac{\vec{Q}}{2\eta}\right)^2} \left(\frac{2^4\eta^3}{2\vec{Q}^3} + \frac{2^3\eta}{3\vec{Q}} \right) + \frac{4\sqrt{\pi}}{3} \operatorname{erfc}\left(\frac{\vec{Q}}{2\eta} \right) \right), \tag{A.54}$$

and so the final term in the dipolar interaction can be written as

$$-3 \sum_{i=1}^{A} \sum_{j=1}^{B} \sum_{m=1}^{A} \sum_{n=1}^{B} {}' \sum_{\vec{G}} \frac{(\vec{\sigma}(i\hat{x}+j\hat{y})\cdot(\vec{R}+\vec{G}))(\vec{\sigma}(m\hat{x}+n\hat{y})\cdot(\vec{R}+\vec{G}))}{|\vec{R}+\vec{G}|^5} =$$
$$\frac{9\sqrt{\pi}}{2} \sum_{i=1}^{A} \sum_{j=1}^{B} \sum_{m=1}^{A} \sum_{n=1}^{B} {}' (\vec{\sigma}(i\hat{x}+j\hat{y})\cdot\vec{\nabla}_\alpha)(\vec{\sigma}(m\hat{x}+n\hat{y})\cdot\vec{\nabla}_\alpha) \Bigg\{ \sum_{\vec{G}} e^{-i\vec{\alpha}\cdot(\vec{R}+\vec{G})}$$
$$\sum_{\vec{G}} \frac{1}{2|\vec{R}+\vec{G}|^5} \left(\frac{3}{4}\sqrt{\pi} \operatorname{erfc}(\eta|\vec{R}+\vec{G}|) + (\frac{3}{2}\eta|\vec{R}+\vec{G}| + \eta^3|\vec{R}+\vec{G}|^3) e^{-\eta^2|\vec{R}+\vec{G}|^2} \right)$$
$$+ \frac{\pi}{AB} \sum_{\vec{Q}} e^{i\vec{Q}\cdot\vec{R}} \frac{\vec{Q}^3}{2^4} \left(e^{-\left(\frac{\vec{Q}}{2\eta}\right)^2} \left(\frac{2^4\eta^3}{2\vec{Q}^3} + \frac{2^3\eta}{3\vec{Q}} \right) + \frac{4\sqrt{\pi}}{3} \operatorname{erfc}\left(\frac{\vec{Q}}{2\eta} \right) \right) \Bigg\} \Bigg|_{\vec{\alpha}=0} \tag{A.55}$$

A.2 Exchange Energy

Calculating the exchange energy is much simpler to calculate since it is short range. We immediately express the average exchange energy as

$$-\frac{J}{AB} \sum_{i=1}^{A} \sum_{j=1}^{B} \sum_{\vec{R}\in \mathrm{NN}(i,j)} \vec{\sigma}(i\hat{x}+j\hat{y})\cdot\vec{\sigma}(\vec{R}), \tag{A.56}$$

where J is the strength of the exchange interaction and $\mathrm{NN}(i,j)$ is the set of nearest neighbours around site $i\hat{x}+j\hat{y}$.

A.3 Aniotropy Energy

Finally, the average anisotropic energy is

$$-\frac{\kappa}{AB}\sum_{i=1}^{A}\sum_{j=1}^{B}(\vec{\sigma}(i\hat{x}+j\hat{y})\cdot\hat{n}_{i,j})^2, \tag{A.57}$$

with κ being the strength of the interaction and $\hat{n}_{i,j}$ representing the local easy axis at site $i\hat{x}+j\hat{y}$.

Appendix B

shm_utils

In this appendix we present the custom parallel library, shm_utils, which was designed to implement the simulation code in an efficient, parallel manner. This library is intended for use on Symmetric Multi-Processor systems which allow the use of shared memory and multiple computational units. Naming conventions for routines follow a standard which is reminiscent of MPI for ease of use.

Then entire library is less than 30k of source code divided between `shmutils.c` and `shmutils.h`. Rather than linking against static or dynamic libraries, these files are intended to be part of the set of source files which define the parallel application.

B.1 SHM_Comm

Most function calls associated with this library make use of a SHM_Comm structure. This contains information about the active process's id or rank in the work group, the number of processes that comprise the workgroup and information about where the process is phys-

ically being run on the SMP machine. Data for pipe based communication is held in this structure along with shared memory for barriers and a pointer to the parent workgroup, if one exits.

The structure itself the following form

```
struct SHM_Comm //communication struct
{
        //identity and environment
        int id, np, cpu, node;
        //explicit pipe based communication
        int *pipe_fd, *send_pipe, *recv_pipe;
        //shared memory
        struct SHM_Segment* barrier;
        //counter to help with debuggung
        long barrierCount;
        //member array
        int* member;
        struct SHM_Comm* parent;
        //semaphore based barriers
        int* semid;
        struct sembuf lock;// = {0, -1, 0};
        struct sembuf unlock;// = {0, 1, 0};
};
```

SHM_Comm

This is analogous to the MPI_Comm structure and plays much the same role. A default structure is provided similar to MPI_COMM_WORLD which under shm_utils is referred to

as SHM_Default.

B.2 Initialization

Before any shared memory or parallel functions can be called, initialization must occur. This is carried out via the following function

- int **SHM_Init**(int *np*, struct SHM_Comm* *comm*)

where *np* is the number of processes you wish to use and *comm* is generally SHM_Default. This initialization routine forks to the required number of processors and populates all values in SHM_Comm appropriately for each process.

B.3 Work Groups

Dividing the overall set of processes into smaller workgroups allows control over the granularity of the parallel task. These three routines are provided to aid in the use of workgroups

- struct SHM_Comm* **SHM_New_Group**(int *num*, int* *mem*, struct SHM_Comm* *comm*)
- void **SHM_Free_Group**(struct SHM_Comm* *WG*)
- int **SHM_In_Group**(struct SHM_Comm* *WG*)

The first creates a subgroup from the set of processes associated with *comm* of size *num* consisting of the processes whose id is in *comm* are given in the array *mem*. The returning value of this routine is a newly created SHM_Comm structure with values populated to allow exclusive communication and synchronization among members of the group. The parent of the returned workgroup is set to *comm*.

The second routine is used to deallocate all shared memory, semaphores, structures and pipes created for the workgroup. After this call, actions associated with *WG* are no longer valid.

The last routine is used to check if the executing process is part of a workgroup. If it is part of the given workgroup 1 is returned, otherwise 0.

B.4 Shared Memory

Access to common, shared memory between processes is an integral part of this parallel library. Initialization, allocation, access and freeing of shared memory are non trivial processes and as such, a set of routines has been provided to encapsulate these tasks.

Shared memory has more information than simply the pointer to the resource. A structure has been defined which hold the information required to operate on the data.

```
struct SHM_Segment //shared memory segment
{
        int local;
        int shmid;
        int size;
        void* memory;
};
```

SHM_Segment

The first member of SHM_Segment is the integer *local* which is interpreted as a boolean value and indicates that the shared memory is truly local, created with the malloc command, or is shared, created with the underlying system commands to access shared memory. The

variable *shmid* is the system-wide identifier for the memory and *size* is the size, in bytes, of the shared memory. The last variable, a void pointer, points to a physical resource that acts as the shared memory. This pointer either points to local memory which has been memory mapped to external memory or directly points to the shared memory, depending on implementation.

The functions which allocate and operate on the shared memory are listed below

- struct SHM_Segment* **SHM_Malloc**(int *owner*, int *size*, struct SHM_Comm* *comm*)
- struct SHM_Segment* **SHM_Monitor**(int *shmid*, int *size*)
- void **SHM_Free**(struct SHM_Segment* *ss*)

SHM_Malloc is the shared memory, parallel version of malloc. It allocates a segment of shared memory which it returns, encapsulated in the shared memory segment structure SHM_Segment, to the requesting processes. Each allocation of a shared memory segment must initially have a primary owner, *owner*, which is responsible for requesting the resource from the underlying Operating System and then sharing information with the other processes which will enable them to access the memory. The other processes are those who are part of the group defined by *comm*. The *size* of the shared memory is exactly the same as the size given to the malloc command and is measured in bytes.

SHM_Monitor connects to a shared memory segment defined by the system identifier *shmid* of size *size* bytes. This is primarily used to allow external processes or programs to access a shared memory segment.

SHM_Free deallocates the shared memory, if the *local* flag in *ss* is nonzero then the system free command is used, otherwise the proper set of commands to detach from the shared memory and possibly deallocate the system resources are executed.

B.5 Explicit Interprocess Communication

While all communication can occur via shared memory and proper synchronization calls, it is sometimes useful to explicitly move data between processes. This ability is required at some stages of initialization and memory allocation since no shared memory pathways have been established. These calls are analogous to the MPI functions of similar names and should be avoided if performance is an issue since they use pipes as the communication mechanism and are much slower than shared memory.

- int **SHM_Send**(int *toID*, void* *data*, int *size*, struct SHM_Comm* *comm*)
- int **SHM_Broadcast**(void* *data*, int *size*, struct SHM_Comm* *comm*)
- int **SHM_Recv**(int *fromID*, void* *data*, int *size*, struct SHM_Comm* *comm*)

SHM_Send is a one to one communication method between the sender and the process with id *toID* of the set of processes defined by *comm*. Unlike the MPI routines for sending and receiving, this data is typeless and the data to be sent is at the location pointed to by the void pointer *data* of length *size* bytes. The reason these are typeless is because there are no endien mismatches across nodes in a single SMP machine.

SHM_Broadcast is a one to many communication method. The data to be sent to all other members of *comm* is described in the same method as in SHM_Send. Internally, the data is sent to each process sequentially.

SHM_Recv is the corresponding call made by the process which will receive data by either SHM_Send or SHM_Broadcast from the process identified by *fromID*. The received data will be placed at the location specified by *data* and will be *size* bytes long.

B.6 Synchronization

Process synchronization is accomplished via barrier calls. This halts the execution of code until all members of a specified work group have arrived at the barrier. The shm_utils library offers the following function

- **void SHM_Barrier**(struct SHM_Comm* *comm*)

The structure *comm* contains the shared memory segment or semaphore and semaphore operators required to execute the barrier, depending on barrier implementation. Care should be given to prevent workgroups which do not belong to the group defined by *comm* from attempting to execute the barrier.

B.7 Cleanup

When a parallel program is finished several tasks must be performed to deallocate any shared resources that may have been allocated during initialization. These tasks are carried out by the following routine

- int **SHM_Finalize**(struct SHM_Comm* *comm*)

where *comm* is generally SHM_Default.

B.7.1 External Cleanup

Unlike programs which use local memory, it is not guaranteed that shared memory will be deallocated upon program termination. To help deal with this, the following function is provided.

- void **SHM_GC**(int *start*, int *range*)

This function iterates over the set of shared memory id values from *start* to *start* + *range* and deallocating every shared memory resource that is not currently attached to a process. This function is particularly useful when developing a shared memory application.

B.8 Sample Application

Here we present a simple demonstration of a program using shm_utils featuring workgroup creation and synchronization.

```
#include <stdio.h>
#include <unistd.h>
#include "shm_utils.h"

int main(int argc, char** argv)
{
        int gmb[3] = {0, 3, 4}; //work group members
        int gsz = 3; //work group size
        int np = 6; //total number of processes

                                                                    10
        SHM_Init(np, SHM_Default);

        struct SHM_Comm* WG = SHM_New_Group(gsz, gmb, SHM_Default);

        if(SHM_In_Group(WG))
        {
```

```
            printf("%i in subtask (called %i in work group)\n", SHM_Default->id, WG->id);
            sleep(2*WG->id+2);
            printf("group member %i at group barrier\n", WG->id);
            SHM_Barrier(WG);
        }

        SHM_Free_Group(WG);

        printf("%i at final barrier\n", SHM_Default->id);
        SHM_Barrier(SHM_Default);
        printf("%i complete\n", SHM_Default->id);

        SHM_Finalize(SHM_Default);

        return 0;
}
```

Demonstration of a parallel program using shm_utils

This program executes as 6 processes, 3 of which are part of an exclusive work group. Members of this work group pause for several seconds before destroying their work group and meeting the remaining members at the final barrier. The output from this demonstration is given below.

```
0 in subtask (called 0 in work group)
1 at final barrier
3 in subtask (called 1 in work group)
2 at final barrier
5 at final barrier
```

```
4 in subtask (called 2 in work group)
group member 0 at group barrier
group member 1 at group barrier
group member 2 at group barrier
3 at final barrier
0 at final barrier
4 at final barrier
3 complete
4 complete
5 complete
1 complete
0 complete
2 complete
```

Output from Demonstation of shm_utils

Bibliography

[1] Document for a standard message-passing interface. Technical report, Knoxville, TN, USA, 1993.

[2] I. Booth, A. B. MacIsaac, J. P. Whitehead, and K. De'Bell. Domain structures in ultrathin magnetic films. *Phys. Rev. Lett.*, 75(5):950953, Jul 1995.

[3] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *Ann. Math. Stat.*, 28:610–611, 1958.

[4] W. F. Brown. Thermal fluctuations of a single-domain particle. *Phys. Rev.*, 130(5):1677, 1963.

[5] C. L. Chien, Frank Q. Zhu, and Jian-Gang Zhu. Patterned nanomagnets. *Phys. Today*, 60(40), June 2007.

[6] R Courant, K Friedrichs, and H Lewy. On the partial difference equations of mathematical physics. *IBM J*, 11:215–234, 1967.

[7] G. C. Danielson and C. Lanczos. Some improvements in practical fourier analysis and their application to x-ray scattering from liquids. *J. Franklin Inst.*, 233(4):356–380, April 1942.

[8] K. De'Bell, A. B. MacIsaac, and J. P. Whitehead. Dipolar effects in magnetic thin films and quasi-two-dimensional systems. *Rev. Mod. Phys.*, 72(1):225257, Jan 2000.

[9] E.W. Dijkstra. *Programming Languages*, chapter Cooperating Sequential Processes, pages 43–112. Academic Press, New York, 1968.

[10] Allen B. Downey. *The Little Book of Semaphores*. 2005.

[11] Alan M. Ferrenberg, D. P. Landau, and Y. Joanna Wong. Monte carlo simulations: Hidden errors from "good" random number generators. *Phys. Rev. Lett.*, 69(23):33823384, Dec 1992.

[12] J. L. García-Palacios and F. J. Lzaro. Langevin-dynamics study of the dynamical properties of small magnetic particles. *Phys. Rev. B*, 58:14937–14958, December 1998.

[13] T. L. Gilbert. A lagrangian formulation of the gyromagnetic equation of the magnetization field. *Phys. Rev.*, 100:1243–1255, 1955.

[14] The HDF Group. Hierarchical Data Format. Website. http://www.hdfgroup.org/.

[15] A. Grzybowski, E. Gwóźdź, and A. Brdka. Ewald summation of electrostatic interactions in molecular dynamics of a three-dimensional system with periodicity in two directions. *Phys. Rev. B*, 61(10):67066712, Mar 2000.

[16] Olle. G. Heinonen, 2006. Private Communication.

[17] Rebecca L. Honeycutt. Stochastic runge-kutta algorithms. i. white noise. *Phys. Rev. A*, 45(2):600603, Jan 1992.

[18] ITL/NIST. oommf. *http://math.nist.gov/oommf/.*

[19] J. D. Jackson. *Classical Electrodynamics.* J. Wiley and Sons, 1999.

[20] H J F Jansen, G S Schneider, and H Y Wang. *Calculation of Magneto-crystalline Anisotropy*, chapter 2, page 57. Electronic Structure and Magnetism of Complex Materials. Springer, April 2003.

[21] Robert J. Jenkins Jr. Isaac. In Dieter Gollmann, editor, *Fast Software Encryption*, volume 1039 of *Lecture Notes in Computer Science*, pages 41–49. Springer, 1996.

[22] J. M. Kosterlitz and D. J. Thouless. Ordering, metastability and phase transitions in two-dimensional systems . *Journal of Physics C Solid State Physics*, 6:1181–1203, April 1973.

[23] Pierre L'Ecuyer and Richard Simard. Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4):22, 2007.

[24] E M Lifshitz and L P Pitaevskii. *Statistical Physics, Part 2*, volume 9 of *Landau and Lifshitz Course of Theoretical Physics.* Elsevier, January 1980.

[25] C. K. Lo and K. W. Yu. Field-induced structure transformation in electrorheological solids. *Phys. Rev. E*, 64(3):031501, Aug 2001.

[26] A. B. MacIsaac. *The Magnetic Properties of a Model Two-Demensional Dipolar Thin Film.* PhD thesis, Memorial University of Newfoundland, St. John s, NL A1C 5S7, P.O. Box 4200, Canada, 1997.

[27] Davide Maltoni, Dario Maio, Anil K. Jain, and Salil Prabhakar. *Handbook of Fingerprint Recognition.* Springer, 1 edition, March 2005.

[28] George Marsaglia. Choosing a point from the surface of a sphere. *Ann. Math. Stats.*, 43(2):645–646, 1972.

[29] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[30] L. Nel. Theorie du trainage magnetique des ferromagnetiques en grains fins avec applications aux terres cuites. *Ann. Geophys.*, 5:99–136, 1949.

[31] ['OpenMP']. *OpenMP Fortran Application Program Interface.* OpenMP Architecture Review Board, version 1.0 edition, 1997.

[32] T. Schrefl, W. Scholz, D. Sss, and J. Fidler. Langevin micromagnetics of recording media using subgrain discretization. *IEEE Trans. Magn.*, 36:3189–3191, 2000.

[33] Thomas Schrefl, Josef Fidler, Rok Dittrich, Dieter Suess, Werner Scholz, Vassilios Tsiantos, and Hermann Forster. *Fast Switching of Mesoscopic Magnets*, page 1. Spin Dynamics in Confined Magnetic Structures II. Springer, 2003.