Master Thesis
Optimization of Langevin simulations for frustrated spin systems

Timo Vinke
Supervised by Kim Lefmann and Henrik Jacobsen

May 2022

Abstract

The Copenhagen Langevin Spin Simulation Code (CLaSSiC) is a simulation suite developed to investigate the absolute temperature effects in frustrated spin systems. Previous versions of CLaSSiC were not fast enough to run system sizes on the order of 10.000 atoms. This report aims to increase its performance to a level where it can run these systems overnight, which would mean a 400x speedup. This will be done by rewriting it in C++ and making better use of the available hardware. To verify the level of optimization, Intel Vtune is used. It shows that there are still unoptimized sections, such as the Gaussian number generation. Afterwards the model has been validated thoroughly to check if it corresponds to theory, which it does. Overall the optimization has been successful and CLaSSiC is now almost 10.000 times faster, running system sizes of 10.000 atoms in under 20 minutes.

Contents

Chapter 1

Introduction

In the field of condensed matter physics a lot of research is spent on frustrated magnetism. Frustrated magnetism happens when due to the geometry of the system the ground state is (infinitely) degenerate. This gives rise to complex dispersions and effects such as the theorized spin liquids. However when a small amount of temperature is added to the system it is unclear as to what happens theoretically since it becomes very hard to solve these systems analytically. What is known is that the degeneracy lifts (partially) and thus properties change.

To measure the magnetic properties generally neutron scattering is used since they can interact with the magnetic moments of the material. However these are real measurements and therefore will always be at some non zero temperature. Which makes it even more relevant to investigate the effects of temperature in frustrated systems.

To solve this issue multiple attempts have been made at finding a numerical solution. The first iteration of this simulation was developed in 2011 in Matlab by Jakob Garde which dealt with nanoparticles that have two submagnetizations[1]. A decade later this code was rewritten in python by Jonas Hyatt such that it could handle more general systems[2]. Estrid Naver validated the python model and got the first preliminary results[3]. However the simulations were very long. A kagome lattice consisting of 27 atoms took 8 hours to run 1.000.000 time steps.

However to observe long range order the systems will have to be much larger. These systems will have to have 20-50 units cells in each direction. So that would mean that a kagome lattice will need at least 3600 magnetic moments. In the current state it is not realistic to run such large systems for any reasonable amount of steps.

Therefore the goal of this thesis is to make a faster implementation that can run a system on the order of 10.000 spins for 1.000.000 time steps overnight. To realize this a speedup on the order of 400x would be required.

To reach the goal the **C**openhagen **La**ngevin **S**pin **Si**mulation **C**ode (CLaSSiC) package will be rewritten in C++. It should be able to run different geometries for various magnetic and anisotropic fields at a finite temperature. The optimization aspect of the implementation will look at efficient memory layout as well as SIMD instructions and possible parallelization to multiple cores. GPU's will not be considered within the scope of this thesis due to availability and the time involved in developing GPU specific software.

This reports starts with laying the theoretical foundation for the physics involved. Chapter 2 will go over the workings of magnetic moments and derive the necessary equations that will be used in the simulation as well as the theoretical results to validate against. Then chapter 3 will give the physics behind neutron scattering such that the simulation can be compared to the experimental data. Then with the physics done, chapter 4 goes in the theory behind creating a fast simulation package. After the theory has been implemented it will need to be checked if it has been done correctly, hence chapter 5. When the validation is done a look is taken how to the code is structured followed by an in depth analysis of the performance. Then a quick guide is given how to install and setup the simulation package. Lastly the conclusion will tie everything together and give an outlook on possibilities for further development and usage.

Chapter 2

Magnetism

This section will go over the theory for magnetic moments. It will first go through the various interactions that play a role in magnetism. Then the dynamics of two simple systems will be described, the ferromagnetic and antiferromagnetic spin waves. The last part will be on how to add absolute temperature to the system and the effects. All of this should provide the necessary equations to create the simulation as well as the result it should be verified against.

2.1 Larmor precession

The basis of magnetism is the magnetic moment $\boldsymbol{\mu}$ defined as the current I running around area $d\mathbf{S}$.

$$d\boldsymbol{\mu} = I d\mathbf{S} \tag{2.1}$$

$$\boldsymbol{\mu} = \int d\boldsymbol{\mu} = I \int d\mathbf{S} \tag{2.2}$$

Since the current loop exists of moving electrons which have mass. Therefore angular momentum will also play a role. The magnetic moment can be written as a function of the angular momentum,

$$\boldsymbol{\mu} = \gamma \mathbf{L} \tag{2.3}$$

Where gamma is gyromagnetic ratio defined as $\gamma = -g\mu_B/\hbar$. When an magnetic field is applied to the magnetic moment it wants to minimize the energy which is given by:

$$E = -\boldsymbol{\mu} \cdot \mathbf{B} \tag{2.4}$$

The energy is minimal when $\boldsymbol{\mu}$ and $\mathbf{B}$ are aligned. This creates a torque on the magnetic moment and with the help of equation 2.3 the equation of motion can be determined as:

$$\frac{d\boldsymbol{\mu}}{dt} = \gamma \boldsymbol{\mu} \times \mathbf{B} \tag{2.5}$$

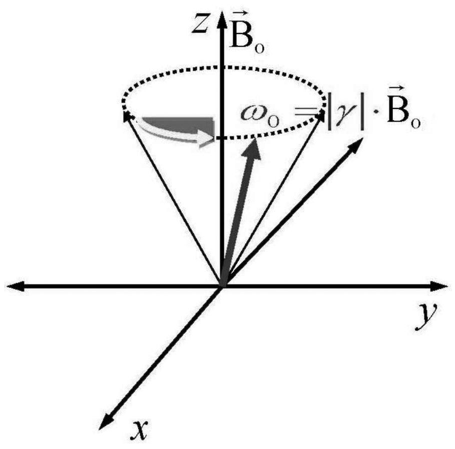


Figure 2.1: The larmor precession for a magnetic field along the z-axis. From Vesna Berec[5].

Concretely this means that the magnetic moment will move in a direction perpendicular to both itself and the magnetic field which means it will rotate around the magnetic field axis. This rotation has a frequency which is known as the Larmor precession frequency[4] and can be found as:

$$\omega = \gamma B \tag{2.6}$$

2.2 Interactions

2.2.1 Magnetic field

As discussed in section 2.1, the magnetic moment is associated with the angular momentum of the electrons orbiting the nucleus. Quantum mechanically this is described by the orbital angular momentum operators $\hat{\mathbf{L}}^2$ and $\hat{\mathbf{L}}_z$ with eigenvalues $l(l+1)\hbar^2$ and $m_l\hbar$ respectively[6]. However electrons also have an intrinsic angular moment which is described by the spin operator $\hat{\mathbf{S}}$. Analogous to the orbital angular momentum the spin operators $\hat{\mathbf{S}}^2$ and $\hat{\mathbf{S}}_z$ have eigenvalues $s(s+1)\hbar^2$ and $m_s\hbar$, respectively. Since both the orbital and intrinsic are always either integer or half integer values of $\hbar$ it is convenient to drop out $\hbar$. Therefore the angular and orbital angular momentum operators will be redefined as $\hbar\hat{\mathbf{L}}$ and $\hbar\hat{\mathbf{S}}$. This is the same as saying that measurements are done in units of $\hbar$. By combining equations 2.2 and 2.4 the energy for an electron in a magnetic field becomes,

$$E = g\mu_B \mathbf{s} \cdot \mathbf{B} \tag{2.7}$$

This means that the energy levels of an electron split by $g\mu_B B$, this effect is called Zeeman splitting. The hamiltonian for this term will be,

$$H_Z = -g\mu_B \mathbf{s} \cdot \mathbf{B} \tag{2.8}$$

2.2.2 Single ion anisotropy

Anisotropy is the phenomenon where the magnetic moment has a preferential axis, known as the easy axis, in the material. This effect is uniaxial meaning that it does not matter which way it is aligned along the axis. B_{anis} can be both positive and negative. When the anisotropy is positive the energy for the magnetic moments is minimized when they lie along the easy axis. When the anisotropy is negative however the magnetic moments minimize their energy when they are in the plane perpendicular to the easy axis. The Hamiltonian for the anisotropy is written as:

$$H_{\text{anis}} = -g\mu_B B_{\text{anis}} \sum_i \mathbf{s}_i^T \kappa \mathbf{s}_i \tag{2.9}$$

Where κ is the matrix defining the direction of the uni axial easy axis. The easy axis for the anisotropy is chosen along the z-axis so equation 2.9 simplifies to:

$$H_{\text{anis}} = -g\mu_B B_{\text{anis}} \sum_i \mathbf{s}_i^z \mathbf{s}_i \tag{2.10}$$

2.2.3 Exchange interaction

The exchange interaction is a purely quantum mechanical effect that happens when two electrons interact with another. Electrons follow Pauli's exclusion principle stating that no two electrons in the same atom can have the same quantum numbers. This means that there are only two ways a pair of electrons in the same orbital can overlap their wave functions, either in the singlet state or in the triplet state.

$$\Psi_S = \frac{1}{\sqrt{2}}[\psi_a(\mathbf{r_1})\psi_b(\mathbf{r_2}) + \psi_a(\mathbf{r_2})\psi_b(\mathbf{r_1})]\chi_S \tag{2.11}$$

$$\Psi_T = \frac{1}{\sqrt{2}}[\psi_a(\mathbf{r_1})\psi_b(\mathbf{r_2}) - \psi_a(\mathbf{r_2})\psi_b(\mathbf{r_1})]\chi_T \tag{2.12}$$

For the singlet state the wave functions are symmetric and the spin state χ_S (S=0) is anti symmetric. For the triplet state it is reversed, the wave functions are anti symmetric and spin state χ_T (S=1)is symmetric The Hamiltonian of the two interacting electrons is,

$$H = \frac{1}{4}(E_S + 3E_T) - (E_S - E_T)\mathbf{s}_1 \cdot \mathbf{s}_2 \tag{2.13}$$

The first term is constant and is therefore unimportant but the second term is interesting since it depends on the orientations of the spins. By defining the exchange constant J as $\frac{1}{2}(E_S - E_T)$ and dropping out the constant term the Hamiltonian can be rewritten to

$$H = -2J\mathbf{s}_1 \cdot \mathbf{s}_2 \tag{2.14}$$

This situation can then be generalized to a many spin system by using the Heisenberg Hamiltonian which is simply the sum of all pairwise interactions.

$$H = -\sum_{ij} J_{ij}\mathbf{s}_i\mathbf{s}_j \tag{2.15}$$

If it is then also assumed that interactions are short ranged the sum can go over only the nearest neighbours thus reducing complexity from $\mathcal{O}(n^n)$ to $\mathcal{O}(2n)$.
Looking at equation 2.15 there are two possible configurations. One is for $J > 0$ and the other is for $J < 0$. In the first case the energy is minimized when the spins all point in the same direction, this is called the ferromagnetic state. When the value for the exchange constant is negative the system will be anti-ferromagnetic meaning that the spins will want to be in the opposite direction of their neighbours.

2.3 Spin dynamics

Now all the interaction Hamiltonians can be combing to get the total Hamiltonian. For the i'th spin the total Hamiltonian reads

$$H_i = -2J\sum_j \mathbf{s}_i \cdot \mathbf{s}_j + g\mu_B\mathbf{s}_i\mathbf{s}_i^z B_{\text{anis}} - g\mu_B\mathbf{s}_i \cdot \mathbf{B} \tag{2.16}$$

All of the terms can be rewritten such that there is one effective field, this is known as the mean field approximation

$$H_i = -g\mu_B\mathbf{s}_i\left(\frac{2J}{g\mu_B}\sum_j \mathbf{s}_j + \mathbf{s}_i^z B_{\text{anis}} + \mathbf{B}\right) \tag{2.17}$$

$$H_i = -g\mu_B\mathbf{s}_i\tilde{\mathbf{B}} \tag{2.18}$$

For this report the semi classical approach is taken where the spins are represented as vectors that can point in any direction. This approximation works on the assumption that $s \gg 1$.
To get from the Hamiltonian to the full equation of motion, Ehrenfests theorem is used.

$$\frac{d\mathbf{s}_i}{dt} = \frac{1}{i\hbar}\left[\mathbf{s}_i, H_i\right] \tag{2.19}$$

To solve this the commutators of the spin operators are needed:

$$[\mathbf{S}^x, \mathbf{S}^y] = i\mathbf{S}^z \qquad [\mathbf{S}^y, \mathbf{S}^z] = i\mathbf{S}^x \qquad [\mathbf{S}^z, \mathbf{S}^x] = i\mathbf{S}^y \tag{2.20}$$

Skipping the tedious math gives the following result for the equation of motion.

$$\frac{d\mathbf{s}_i}{dt} = \frac{-g\mu_B}{\hbar}\mathbf{s}_i \times \tilde{\mathbf{B}} = \gamma\mathbf{s}_i \times \tilde{\mathbf{B}} \tag{2.21}$$

2.4 Ferromagnetic spin waves

Here the semi-classical derivation for the dispersion of spin waves with a positive exchange constant, a magnetic field and anisotropy is given. Spin waves are a small perturbation of the ground state. These small perturbations are small deviations from the z-axis and together form a spin wave.

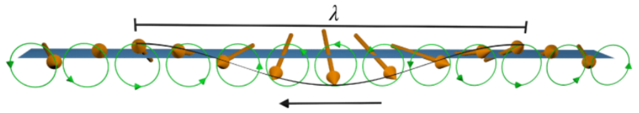


Figure 2.2: Excited state of a one dimensional spin wave with ferromagnetic coupling. Each spin will rotate around the z-axis but with some phase difference with respect to each neighbour creating spin waves. From Nature Nanotechnology[7].

To this extend the individual spins can be rewritten for $\delta \ll 1$ as

$$\mathbf{s}_i = s\hat{z} + \delta\mathbf{s}_i \tag{2.22}$$

Filling this into the equation of motion gives:

$$\frac{d}{dt}(s\hat{z} + \delta\mathbf{s}_i) = \gamma(s\hat{z} + \delta\mathbf{s}_i) \times \left(\frac{2J}{g\mu_B} \sum_j \mathbf{s}_j + \mathbf{s}_i^z B_{\text{anis}} + \mathbf{B} \right) \tag{2.23}$$

To reduce the complexity only nearest neighbour interactions are taken into account, the z-component of the spin is assumed to be constant and $\delta s_i^z = 0$.

$$\frac{1}{\gamma}\frac{d}{dt}(\delta\mathbf{s}_i) = (s\hat{z} + \delta\mathbf{s}_i) \times \left(\frac{2J}{g\mu_B}(2s\hat{z} + \delta\mathbf{s}_{i+1} + \delta\mathbf{s}_{i-1}) + \mathbf{s}_i^z B_{\text{anis}} + \mathbf{B} \right) \tag{2.24}$$

$$\frac{1}{\gamma}\frac{d}{dt}(\delta\mathbf{s}_i) = s\hat{z} \times \frac{2J}{g\mu_B}\left(\delta\mathbf{s}_{i+1} + \delta\mathbf{s}_{i-1}\right) + \delta\mathbf{s}_i \times \left(\frac{2J}{g\mu_B} 2s\hat{z} + \mathbf{s}_i^z B_{\text{anis}} + \mathbf{B}^z \right) \tag{2.25}$$

Now wave-like solutions in the x-y plane are assumed.

$$\delta s_i^x = A_x e^{i(\omega t - \mathbf{k}\cdot\mathbf{r}_i)} \tag{2.26}$$

$$\delta s_i^y = A_y e^{i(\omega t - \mathbf{k}\cdot\mathbf{r}_i)} \tag{2.27}$$

$$\frac{1}{\gamma}\frac{d}{dt}(\delta\mathbf{s}_i^x) = \frac{2Js}{g\mu_B}\left(2\delta\mathbf{s}_i^y - \delta\mathbf{s}_{i+1}^y - \delta\mathbf{s}_{i-1}^y\right) + \delta\mathbf{s}_i^y(\mathbf{B}_{\text{anis}}^z + \mathbf{B}^z) \tag{2.28}$$

$$\frac{1}{\gamma}\frac{d}{dt}(\delta\mathbf{s}_i^y) = -\frac{2Js}{g\mu_B}\left(2\delta\mathbf{s}_i^x - \delta\mathbf{s}_{i+1}^x - \delta\mathbf{s}_{i-1}^x\right) + \delta\mathbf{s}_i^x(\mathbf{B}_{\text{anis}}^z + \mathbf{B}^z) \tag{2.29}$$

$$\frac{1}{\gamma}\omega A_x = \frac{2Js}{g\mu_B} A_y \left(2 - e^{-i(\mathbf{k}\cdot(\mathbf{r}_{i+1}-\mathbf{r}_i))} - e^{-i(\mathbf{k}\cdot(\mathbf{r}_{i-1}-\mathbf{r}_i))}\right) + A_y(\mathbf{B}^z_{\text{anis}} + \mathbf{B}^z) \tag{2.30}$$

$$\frac{1}{\gamma}\omega A_y = -\frac{2Js}{g\mu_B} A_x \left(2 - e^{-i(\mathbf{k}\cdot(\mathbf{r}_{i+1}-\mathbf{r}_i))} - e^{-i(\mathbf{k}\cdot(\mathbf{r}_{i-1}-\mathbf{r}_i))}\right) + A_x(\mathbf{B}^z_{\text{anis}} + \mathbf{B}^z) \tag{2.31}$$

The distance between two neighbouring sites is a and the chain is along the x-axis.

$$\frac{1}{\gamma}\omega A_x = \frac{2Js}{g\mu_B} A_y \left(2 - e^{-ik_x a} - e^{ik_x a}\right) + A_y(\mathbf{B}^z_{\text{anis}} + \mathbf{B}^z) \tag{2.32}$$

$$\frac{1}{\gamma}\omega A_y = -\frac{2Js}{g\mu_B} A_x \left(2 - e^{-ik_x a} - e^{ik_x a}\right) + A_x(\mathbf{B}^z_{\text{anis}} + \mathbf{B}^z) \tag{2.33}$$

$$\frac{1}{\gamma}\omega A_x = \frac{4Js}{g\mu_B} A_y \left(1 - cos(k_x a)\right) + A_y(\mathbf{B}^z_{\text{anis}} + \mathbf{B}^z) \tag{2.34}$$

$$\frac{1}{\gamma}\omega A_y = -\frac{4Js}{g\mu_B} A_x \left(1 - cos(k_x a)\right) + A_x(\mathbf{B}^z_{\text{anis}} + \mathbf{B}^z) \tag{2.35}$$

Solving for ω gives:

$$\hbar\omega = 4Js(1 - cos(k_x a)) + g\mu_B(\mathbf{B}^z_{\text{anis}} + \mathbf{B}^z) \tag{2.36}$$

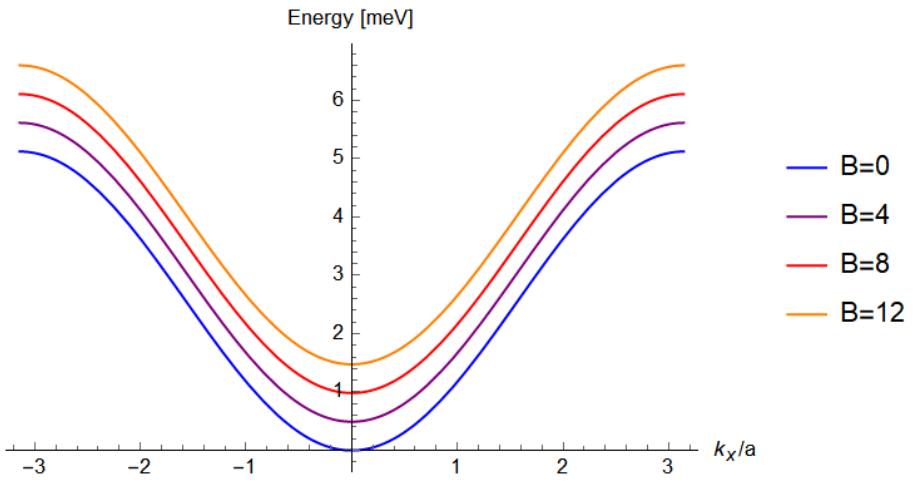


Figure 2.3: The dispersion for a ferromagnetic spin chain at various magnetic field strengths.

As can be seen in figure 2.3 the magnetic field has the effect of shifting the dispersion. The same holds when the anisotropy is varied.

2.5 Antiferromagnetic spin waves

In the case of the antiferromagnetic system the lattice is split up into two sub-lattices with the k sub-lattice having all the spins point up and the l sub-lattice having all the spins point down.

$$\frac{d\mathbf{s}_k}{dt} = \gamma \mathbf{s}_k \times \tilde{\mathbf{B}}_k \qquad \frac{d\mathbf{s}_l}{dt} = \gamma \mathbf{s}_l \times \tilde{\mathbf{B}}_l \tag{2.37}$$

with:

$$\tilde{\mathbf{B}}_k = \frac{2J}{g\mu_B}\sum_l \mathbf{s}_l - \mathbf{s}_k^z B_{\text{anis}} - B \qquad \tilde{\mathbf{B}}_l = \frac{2J}{g\mu_B}\sum_k \mathbf{s}_k - \mathbf{s}_l^z B_{\text{anis}} - B \tag{2.38}$$

$$\mathbf{s}_k = s\hat{z} + \delta\mathbf{s}_k \qquad \mathbf{s}_l = -s\hat{z} + \delta\mathbf{s}_l \tag{2.39}$$

Now everything is filled in and only the first order terms in $\delta\mathbf{s}$ are kept.

$$\frac{d\delta\mathbf{s}_l}{dt} = \gamma\delta\mathbf{s}_l \times \left(\frac{2J}{g\mu_B}\sum_k \mathbf{s}_k^z - \mathbf{s}_l^z B_{\text{anis}} - B\right) - \frac{2J}{g\mu_B}\sum_k s\hat{z}\times\mathbf{s}_k \tag{2.40}$$

$$\frac{d\delta\mathbf{s}_k}{dt} = \gamma\delta\mathbf{s}_k \times \left(\frac{2J}{g\mu_B}\sum_l \mathbf{s}_l^z - \mathbf{s}_k^z B_{\text{anis}} - B\right) - \frac{2J}{g\mu_B}\sum_l s\hat{z}\times\mathbf{s}_l \tag{2.41}$$

Just like in the ferromagnetic case wave like solutions are assumed.

$$\mathbf{A}(\mathbf{k}) = \sqrt{\frac{N}{2}}\sum_k e^{-i\mathbf{k}\cdot\mathbf{r_k}}\delta\mathbf{s}_k \tag{2.42}$$

$$\mathbf{B}(\mathbf{k}) = \sqrt{\frac{N}{2}}\sum_l e^{-i\mathbf{k}\cdot\mathbf{r_l}}\delta\mathbf{s}_l \tag{2.43}$$

$$\frac{\mathbf{A}(\mathbf{k})}{dt} = \gamma\mathbf{A}(\mathbf{k}) \times (\frac{2J}{g\mu_B} + sB_{\text{anis}}\hat{z} + B\hat{z}) + \frac{2J}{g\mu_B}e^{-i\mathbf{k}\cdot(\mathbf{r}_k - \mathbf{r}_l)}\mathbf{B}(\mathbf{k}) \times s\hat{z} \tag{2.44}$$

$$\frac{\mathbf{B}(\mathbf{k})}{dt} = \gamma\mathbf{B}(\mathbf{k}) \times (\frac{2J}{g\mu_B} + sB_{\text{anis}}\hat{z} + B\hat{z}) + \frac{2J}{g\mu_B}e^{-i\mathbf{k}\cdot(\mathbf{r}_k - \mathbf{r}_l)}\mathbf{A}(\mathbf{k}) \times s\hat{z} \tag{2.45}$$

Next the assumption is made that there are only interactions between neighbouring sites. Therefore the fourier term can be written as:

$$\sum_j Je^{-i\mathbf{k}\cdot(\mathbf{r}_k - \mathbf{r_l})} = zJ\gamma_q \tag{2.46}$$

with z the number of nearest neighbours and γ_q is the Fourier transform. To make progress the following change of variables is performed:

$$A_+(\mathbf{k}) = A_x(\mathbf{k}) + iA_y(\mathbf{k}) \tag{2.47}$$

$$B_+(\mathbf{k}) = B_x(\mathbf{k}) + iB_y(\mathbf{k}) \tag{2.48}$$

After adding the equations for the x and y components the new equations of motion become:

$$\frac{dA_+(\mathbf{k})}{dt} = \gamma i((\frac{2J}{g\mu_B}zs + sB_{\text{anis}} + B)A_+(\mathbf{k}) - 2Jzs\gamma_q B_+(\mathbf{k}))) \tag{2.49}$$

$$\frac{dB_+(\mathbf{k})}{dt} = \gamma i((\frac{2J}{g\mu_B}zs + sB_{\text{anis}} + B)B_+(\mathbf{k}) - 2Jzs\gamma_q A_+(\mathbf{k}))) \tag{2.50}$$

$$\begin{vmatrix} \hbar\omega_q - (2Jzs - g\mu_B sB_{\text{anis}} - g\mu_B B) & -2Jsz\gamma_q \\ 2Jsz\gamma_q & \hbar\omega_q + (2Jzs - g\mu_B sB_{\text{anis}} - g\mu_B B) \end{vmatrix} = 0 \tag{2.51}$$

Solving this gives:

$$\hbar\omega = \sqrt{(2Jzs)^2(1-\gamma_q^2) - 4Jzsg\mu_B B_{\text{anis}} + (g\mu_B B_{\text{anis}})^2} \pm g\mu_B B \tag{2.52}$$

For a one dimensional chain $z = 2$ and $\gamma_q = cos(k_x a)$ filling this in gives the dispersion as:

$$\hbar\omega = \sqrt{(4Js)^2(1-cos^2(k_x a)) - 8Jsg\mu_B B_{\text{anis}} + (g\mu_B B_{\text{anis}})^2} \pm g\mu_B B \tag{2.53}$$

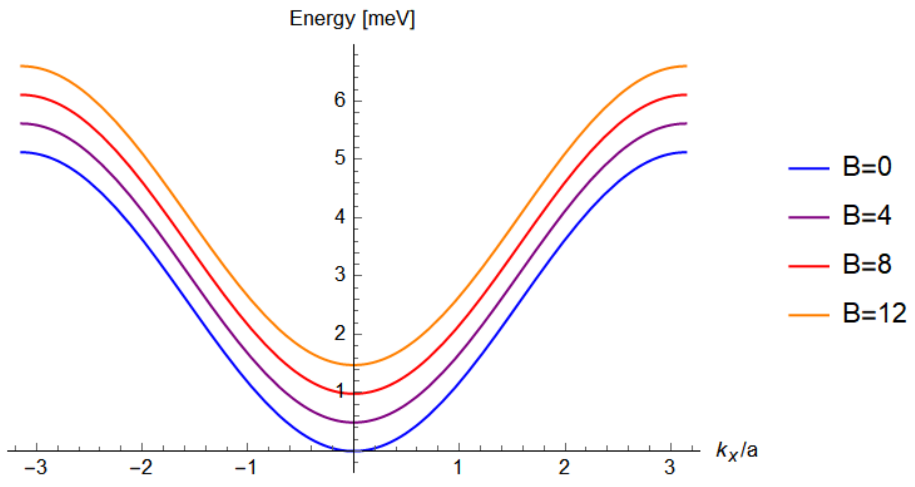


Figure 2.4: The dispersion for an antiferromagnetic system with various anisotropies.

Figure 2.4 shows that increase anisotropy has not only the effect of shifting the dispersion up but also changing the shape. While the magnetic field has the effect of splitting the dispersion.

2.6 Langevin equation

As was shown in section 2.1 the magnetic moment wants to align with the field to minimize its energy. However when there is temperature present the moments will experience random fluctuations which keep the system from reaching a minimum energy. The average magnetic moment along the z-axis in a material is given by the Langevin equation,

$$\frac{<\boldsymbol{\mu}^z>}{|\boldsymbol{\mu}|} = L(y) = coth(y) - \frac{1}{y} \tag{2.54}$$

Where y is defined as: $y = \frac{\mu B}{k_b T}$.

2.7 Temperature

Temperature is a collection of effects that cause excitation and fluctuations in the direction of the spins. On a very short time scale all of these effects are deterministic interactions between the phonon, electrons and magnetic fields. However this calculation is not feasible but since the timescales are long enough all of the deterministic effects will follow a Gaussian. Since it is important to have a model that can be compared to experiments the Gaussian variance will have to depend on the absolute temperature which will be derived here.

The thermal properties are modelled as if the system is in contact with a heat bath of temperature T. This means that the field that is created by the energy that is added to the system must have the following properties. It will be a Gaussian distribution that is isotropic[8]. The fluctuations are random deviations of the $\tilde{\mathbf{B}}$ field, so therefore the mean should be zero.

$$<\mathbf{b}_i(t)>= 0 \tag{2.55}$$

The strength of the random $\mathbf{b}$ field should be dependent on the frictional constant λ and the temperature T. These dependencies show up in the correlation function.

$$<\mathbf{b}_i(t)\mathbf{b}_j(t')>= D\delta_{ij}\delta(t-t') \tag{2.56}$$

Where D is the diffusion constant given by the Einstein relation[9]:

$$D = 2\frac{\lambda}{|\gamma|m}k_b T \tag{2.57}$$

The model without any temperature effects has been described in equation 2.21. Firstly the contact with the heat bath allows for dissipation from the system into the heat bath. This dissipation interaction can be seen as a sort of friction on the system. This interaction can be added by use of the Landau–Lifshitz–Gilbert equation[10].

$$\frac{d\mathbf{s}_i}{dt} = \gamma\mathbf{s}_i \times (\tilde{\mathbf{B}} - \lambda\frac{d\mathbf{s}_i}{dt}) \tag{2.58}$$

Which states that the friction term is the dissipation constants λ times the velocity of the spin.

Combing both the random fluctuations and the dissipation term into the Langevin equation gives,

$$\frac{d\mathbf{s}_i}{dt} = \gamma \mathbf{s}_i \times (\tilde{\mathbf{B}}_i + \mathbf{b}_i) \pm \frac{\lambda}{m} \mathbf{s}_i \times \frac{d\mathbf{s}_i}{dt} \tag{2.59}$$

Then $\frac{d\mathbf{s}_i}{dt}$ will be expanded once:

$$\frac{d\mathbf{s}_i}{dt} = \gamma \mathbf{s}_i \times (\tilde{\mathbf{B}}_i + \mathbf{b}_i) \pm \frac{\lambda}{m} \mathbf{s}_i \times \left(\gamma \mathbf{s}_i \times (\tilde{\mathbf{B}}_i + \mathbf{b}_i) \pm \frac{\lambda}{m} \mathbf{s}_i \times \frac{d\mathbf{s}_i}{dt} \right) \tag{2.60}$$

Now when the second order terms in λ are removed. This gives the Landau-Lifschitz equation:

$$\frac{d\mathbf{s}_i}{dt} \approx \gamma \mathbf{s}_i \times (\tilde{\mathbf{B}}_i + \mathbf{b}_i) \pm \frac{\lambda\gamma}{m} \mathbf{s}_i \times (\mathbf{s}_i \times \tilde{\mathbf{B}}_i) \tag{2.61}$$

Chapter 3

Scattering

In practice it is not possible to measure the real time movement of the spins. So instead neutron scattering is used to measure the magnetic properties of the system. This chapter will go over the physics behind neutron scattering.

3.1 The basics

Neutrons are ideal particles for doing these measurements. They are neutral and have spin 1/2. Since they are neutral they do not interact with charges and can therefore penetrate materials deeply. The spin on the other hand allows it to interact with the magnetic moments of the material, thus making it possible to measure their effects. Lastly by adjusting the energy of the neutrons it can have wavelengths comparable to interatomic distances.

When the experiment is performed a beam of neutrons is fired at the sample where they will scatter from the sample. Then the scattered neutrons are measured. This measurement is called scattering cross section. It is simply the number of measured neutrons normalized by the neutron flux from the beam. Since the angle at which they scatter is also important the differential scattering cross section is measured as well. This is the same as the regular cross section but now per solid angle $d\Omega$. The next part will go over how to calculate these cross sections.

3.2 Semi-classical elastic scattering

3.2.1 Single nucleus

Due to particle wave duality the neutron can behave both as a wave and particle. Therefore the incoming neutron will be written as a plane wave

$$\psi_i(\boldsymbol{r}) = \frac{1}{\sqrt{Y}} \exp(i\boldsymbol{k}_i \cdot \boldsymbol{r}) \tag{3.1}$$

where Y is a normalization constant. The time dependence of the wave has been left out since it will not effect the results in this case. The incoming plane wave will scatter isotropically on the nucleus at position $\boldsymbol{r}_j$. Since it is isotropic it will be spherically symmetric and can therefore be described by a spherical wave. Combining all of this gives the equation for the scattered (final) particle as:

$$\psi_f(\boldsymbol{r}) = \psi_i(\boldsymbol{r}_j) \frac{-b_j}{|\boldsymbol{r} - \boldsymbol{r}_j|} \exp(ik_f|\boldsymbol{r} - \boldsymbol{r}_j|) \tag{3.2}$$

Where b_j has units $[m]$ and is denoted as the scattering length. Note that equation 3.2 is only valid when the scattering length is much smaller than the distance between the scattering event and the observation.

3.2.2 System of nuclei

In general it is more interesting to look at systems of particles instead of a single nucleus. Therefore this section will go through the two particle system first which can then later be generalized for n particle systems. The incoming wave will scatter from both nuclei and the final wave will be a superposition of both scattered waves. It should be noted that the Born approximation is used which says that the wave is not noticeably attuned between the two nuclei. This results in the following equation for the final wave.

$$\psi_f(\boldsymbol{r}) = \left(\psi_i(\boldsymbol{r}_j) \frac{b_j}{|\boldsymbol{r} - \boldsymbol{r}_j|} \exp(ik_f|\boldsymbol{r} - \boldsymbol{r}_j|) + \psi_i(\boldsymbol{r}_{j'}) \frac{b_j'}{|\boldsymbol{r} - \boldsymbol{r}_{j'}|} \exp(ik_f|\boldsymbol{r} - \boldsymbol{r}_{j'}|) \right) \tag{3.3}$$

Next it is assumed that the nuclei are very close together compared to the distance to the observer. Thus for $|\boldsymbol{r}_j - \boldsymbol{r}_{j'}| \ll r$ it can be approximated that:

$$\frac{1}{|\boldsymbol{r} - \boldsymbol{r}_j|} \approx \frac{1}{|\boldsymbol{r} - \boldsymbol{r}_{j'}|} \approx \frac{1}{r} \tag{3.4}$$

With this approximation the equation for the final wave becomes

$$\begin{aligned} \psi_f(\boldsymbol{r}) = -\frac{1}{\sqrt{Y}} \frac{1}{r} [b_j \exp\{(i\boldsymbol{k}_i \cdot \boldsymbol{r}_j)\} \exp\{(ik_f|\boldsymbol{r} - \boldsymbol{r}_j|)\} \\ + b_{j'} \exp\{(i\boldsymbol{k}_i \cdot \boldsymbol{r}_{j'})\} \exp\{(ik_f|\boldsymbol{r} - \boldsymbol{r}_{j'}|)\}] \end{aligned} \tag{3.5}$$

Now unlike the approximation made before with the denominator the same will not hold for the exponential. Since the value in the exponent is the phase of the complex wave function a small shift might move it out of a maximum into a minimum. To make calculations easier the nuclear coordinate $\boldsymbol{r}_j$ can be written as a component parallel and a component perpendicular to $\mathbf{r}$.

$$|\boldsymbol{r} - \boldsymbol{r}_j| = |\boldsymbol{r} - \boldsymbol{r}_{j,\parallel} - \boldsymbol{r}_{j,\perp}| = \sqrt{|\boldsymbol{r} - \boldsymbol{r}_{j,\parallel}|^2 + |\boldsymbol{r}_{j,\perp}|^2} \tag{3.6}$$

Since $|\boldsymbol{r}_{j,\perp}|^2 \ll |\boldsymbol{r} - \boldsymbol{r}_{j,\parallel}|^2$ the approximation $\sqrt{x^2 + \delta^2} \approx |x|$ is valid. Thus $|\boldsymbol{r} - \boldsymbol{r}_j| \approx |\boldsymbol{r} - \boldsymbol{r}_{j,\parallel}|$ and with this term in the exponent in equation 3.5 can be rewritten as a scalar product as follows:

$$\begin{aligned} k_f|\boldsymbol{r} - \boldsymbol{r}_j| &= k_f|\boldsymbol{r} - \boldsymbol{r}_{j,\parallel}| && (3.7)\\ &= \boldsymbol{k}_f \cdot (\boldsymbol{r} - \boldsymbol{r}_{j,\parallel}) && (3.8)\\ &= \boldsymbol{k}_f \cdot (\boldsymbol{r} - \boldsymbol{r}_j) && (3.9) \end{aligned}$$

Where $\boldsymbol{k}_f$ is the vector that is parallel to $\boldsymbol{r}$ and has length k_f. Further it is used that the dot product of $\boldsymbol{k}_f$ with the $\boldsymbol{r}_{j,\perp}$ is zero. At last this gives the final equation for the total scattered wave equation

$$\begin{aligned} \psi_f(\boldsymbol{r}) = -\frac{1}{\sqrt{Y}}\frac{1}{r}&\exp(i\boldsymbol{k}_f \cdot \boldsymbol{r}) \\ &[b_j \exp(i(\boldsymbol{k}_i - \boldsymbol{k}_f) \cdot \boldsymbol{r}_j) + b_{j'} \exp(i(\boldsymbol{k}_i - \boldsymbol{k}_f) \cdot \boldsymbol{r}_{j'})] \end{aligned} \tag{3.10}$$

Which is a plane wave with wave vector $\boldsymbol{k}_f$ on which an interference pattern is imposed. From this the scattering intensity can be written as:

$$\frac{1}{Y}\frac{\hbar k_f}{m_n} d\Omega |b_j \exp(i\boldsymbol{q} \cdot \boldsymbol{r}_j) + b_{j'} \exp(i\boldsymbol{q} \cdot \boldsymbol{r}_{j'})|^2 \tag{3.11}$$

Where $\boldsymbol{q}$ is the neutron scattering vector which is defined as:

$$\boldsymbol{q} = \boldsymbol{k}_i - \boldsymbol{k}_f \tag{3.12}$$

For the simple case that $b_j = b_{j'} = b$ the differential scattering cross section becomes:

$$\frac{d\sigma}{d\Omega} = 2b^2(q + \cos(\boldsymbol{q} \cdot (\boldsymbol{r}_j - \boldsymbol{r}_j))) \tag{3.13}$$

To generalize this to a system of nuclei is straightforward since the interference effect applies for all particles as long as they are close enough together. So the general equation for the differential scattering cross section becomes:

$$\frac{d\sigma}{d\Omega} = \left| \sum_j b_j \exp(i\boldsymbol{q} \cdot \boldsymbol{r}_j) \right|^2 \tag{3.14}$$

3.3 Quantum mechanical inelastic scattering

The quantum mechanical treatment of scattering has the same starting point as the semi-classical approach namely the incoming neutron is given by a plane wave.

$$|\psi_i\rangle = \frac{1}{\sqrt{Y}} \exp\{i\boldsymbol{k}_i \cdot \boldsymbol{r}\} \tag{3.15}$$

However now instead of the scattered wave being spherical it will also be a plane wave.

$$|\psi_f\rangle = \frac{1}{\sqrt{Y}} \exp\{i\boldsymbol{k}_f \cdot \boldsymbol{r}\} \tag{3.16}$$

The Fermi Golden Rule governs this scattering process[11]. It gives the change between the initial state $|\psi_i\rangle$ of the incoming neutron and the final state $|\psi_f\rangle$ of the outgoing neutron.

$$W_{i\to f} = \frac{2\pi}{\hbar} \frac{\mathrm{d}n}{\mathrm{d}E_f} \left|\langle\psi_i|\, \hat{V}\, |\psi_f\rangle\right|^2 \tag{3.17}$$

Here $\hat{V}$ is the scattering potential and is defined as

$$\hat{V} = \frac{2\pi\hbar^2}{m_n} \sum_j b_j \delta(\boldsymbol{r} - \boldsymbol{R}_j) \tag{3.18}$$

with $\frac{\mathrm{d}n}{\mathrm{d}E_f}$ the density of states. In this case the density of states is a spherical shell in k-space in scattering direction $d\Omega$ and is given by

$$\frac{\mathrm{d}n}{\mathrm{d}E_f} = \frac{Y k_f m_n}{2\pi^2\hbar^2} \frac{d\Omega}{4\pi} \tag{3.19}$$

With this, equation 3.17 can be rewritten such that the it gives neutrons per solid angle $d\Omega$.

$$W_{i\to f, d\Omega} = \frac{Y k_f m_n}{(2\pi^2)^2\hbar^3} d\Omega \left|\langle\psi_i|\, \hat{V}\, |\psi_f\rangle\right|^2 \tag{3.20}$$

Here $W_{i\to f,d\Omega}$ is the rate of scattered neutrons in $d\Omega$ per second. Now that the scattering rate is know the differential scattering cross section can be determined as

$$\frac{\mathrm{d}\sigma}{\mathrm{d}\Omega} = Y^2 \frac{k_f}{k_i} \left(\frac{m_n}{2\pi\hbar^2}\right)^2 \left|\langle\psi_i|\, \hat{V}\, |\psi_f\rangle\right|^2 \tag{3.21}$$

When the scattering is inelastic the ration $k_f/k_i = 1$ and this gives the classical scattering equation 3.14, again. In elastic cases the energy of the neutrons will change. The following equation keeps track of the difference in energy between the initial state $|\boldsymbol{k}_f\lambda_f\rangle$ and the final state $|\boldsymbol{k}_f\lambda_i\rangle$.

$$\left.\frac{\partial^2\sigma}{\partial\Omega\partial E_f}\right|_{\lambda_i\to\lambda_f} = Y^2 \frac{k_f}{k_i} \left(\frac{m_n}{2\pi\hbar^2}\right)^2 \left|\langle\lambda_i\boldsymbol{k}|\, \hat{V}\, |\boldsymbol{k}_f\lambda_f\rangle\right|^2 \delta(E_{\lambda_i} - E_{\lambda_f} - \hbar\omega) \tag{3.22}$$

The δ function makes sure that the energy is conserved through the scattering process. Writing out the matrix elements gives:

$$\left|\langle\lambda_i \boldsymbol{k}|\,\hat{V}\,|\boldsymbol{k}_f\lambda_f\rangle\right|^2 = \left(\frac{2\pi\hbar^2}{m_n}\right)^2 \sum_{j,j'} b_j b_{j'} \langle\lambda_i|\exp(-i\boldsymbol{q}\cdot\boldsymbol{R}_j)|\lambda_f\rangle \langle\lambda_i|\exp(-i\boldsymbol{q}\cdot\boldsymbol{R}_{j'})|\lambda_f\rangle \tag{3.23}$$

Where $|\lambda_i\rangle$ and $|\lambda_f\rangle$ are the initial and final states of the sample which scatters the neutrons. The δ function can also be rewritten as

$$\delta(E_{\lambda_i} - E_{\lambda_f} - \hbar\omega) = \frac{1}{2\pi\hbar}\int_{-\infty}^{\infty} \exp\left(\frac{i(E_{\lambda_f} - E_{\lambda_i})t}{\hbar}\right)\exp(-i\omega t)dt \tag{3.24}$$

Using both of these equations the equation for the differential scattering cross can be written as:

$$\left.\frac{\partial^2\sigma}{\partial\Omega\partial E_f}\right|_{\lambda_i\to\lambda_f} = Y^2\frac{k_f}{k_i}\sum_{b_j b_{j'}} \frac{b_j b_{j'}}{2\pi\hbar} \langle\lambda_i|\exp(-i\boldsymbol{q}\cdot\boldsymbol{R}_j)\exp(-i\boldsymbol{q}\cdot\boldsymbol{R}_{j'})|\lambda_f\rangle$$
$$\times\exp\left(\frac{iE_{\lambda_f}t}{\hbar}\right)\exp\left(\frac{-iE_{\lambda_i}t}{\hbar}\right)\exp(-i\omega t)dt \tag{3.25}$$

$$= Y^2\frac{k_f}{k_i}\sum_{b_j b_{j'}} \frac{b_j b_{j'}}{2\pi\hbar} \langle\lambda_i|\exp(-i\boldsymbol{q}\cdot\boldsymbol{R}_j)|\lambda_f\rangle$$
$$\times\langle\lambda_i|\exp\left(\frac{iHt}{\hbar}\right)\exp(-i\boldsymbol{q}\cdot\boldsymbol{R}_{j'})exp(\frac{-iHt}{\hbar})|\lambda_f\rangle\exp(-i\omega t)dt \tag{3.26}$$

$$= Y^2\frac{k_f}{k_i}\sum_{b_j b_{j'}} \frac{b_j b_{j'}}{2\pi\hbar} \langle\lambda_i|\exp(-i\boldsymbol{q}\cdot\boldsymbol{R}_j(0))|\lambda_f\rangle$$
$$\times\langle\lambda_i|\exp(-i\boldsymbol{q}\cdot\boldsymbol{R}_{j'}(t))|\lambda_f\rangle\exp(-i\omega t)dt \tag{3.27}$$

Here it is used that the energy is an eigenvalue of the Hamiltonian H and that time dependence can be expressed by the time-dependent Heisenberg operators,

$$\boldsymbol{R}_j(t) = exp(\frac{iHt}{\hbar})\boldsymbol{R}_j exp(\frac{-iHt}{\hbar}) \tag{3.28}$$

In an experimental setting the actual final state of the sample is not measured, only the final state of the neutrons. Thus a sum can be taken over all final sample states, which should be equal to one by the completeness rule. Also it is assumed that the system is in thermal equilibrium and it is studied over a much longer time than the neutron frequency. This means the thermal average of the initial states of the sample can be taken. Combining all of these assumptions gives

$$\frac{\partial^2\sigma}{\partial\Omega\partial E_f} = \sum_{\lambda_i\lambda_f}\left.\frac{\partial^2\sigma}{\partial\Omega\partial E_f}\right|_{\lambda_i\to\lambda_f} \tag{3.29}$$

$$\frac{\partial^2\sigma}{\partial\Omega\partial E_f} = \frac{k_f}{k_i}\sum_{b_j b_{j'}} \frac{b_j b_{j'}}{2\pi\hbar} \times \langle exp(-i\boldsymbol{q}\cdot\boldsymbol{R}_j(0))exp(-i\boldsymbol{q}\cdot\boldsymbol{R}_{j'}(t))\exp(-i\omega t)\rangle \tag{3.30}$$

This is the observable scattering cross section and deals with both elastic and inelastic scattering.

3.4 Magnetic scattering

All that is left is adding the effects of magnetic moments to the differential scattering cross section. To do this the scattering potential needs to be modified. The total interaction between the unpaired electrons in the sample and the neutrons is given by the sum of nuclear Zeeman interactions at each of the magnetic sites j.

$$\hat{V} = \frac{\mu_0}{4\pi} g\mu_B\gamma\mu_N\hat{\sigma}\cdot\boldsymbol{\nabla}\times\left(\frac{\mathbf{s}_j\times(\mathbf{r}-\mathbf{r}_j)}{|\mathbf{r}-\mathbf{r}_j|^3}\right) \tag{3.31}$$

Inserting the new magnetic scattering potential into the equation for the scattering cross section results in

$$\left.\frac{\partial^2\sigma}{\partial\Omega\partial E_f}\right|_{\sigma_i\to\sigma_f} = \frac{k_f}{k_i}\left(\frac{\mu_0}{4\pi}\right)^2\left(\frac{m_n}{2\pi\hbar^2}\right)(g\mu_B\gamma\mu_N)^2\sum_{\lambda_i,\lambda_f} p\lambda_i \tag{3.32}$$

$$\times\, |\langle\mathbf{k}_f\lambda_f\sigma_f|\hat{\sigma}\cdot\boldsymbol{\nabla}\times\left(\frac{\mathbf{s}_j\times(\mathbf{r}-\mathbf{r}_j)}{|\mathbf{r}-\mathbf{r}_j|^3}\right)|\mathbf{k}_i\lambda_i\sigma_i\rangle|^2 \tag{3.33}$$

$$\times\,\delta(\hbar\omega + E_{\lambda_i} - E_{\lambda_f}) \tag{3.34}$$

It should be noted that the scattering cross section also depends on the neutron spin sate $|\sigma\rangle$. The magnetic scattering cross section will be rewritten by using the following identity[12]

$$\boldsymbol{\nabla}\times\left(\frac{\mathbf{s}\times\mathbf{r}}{r^3}\right) = \frac{1}{2\pi^2}\int\hat{\mathbf{q}}'\times(\mathbf{s}\times\hat{\mathbf{q}}')\exp(i\mathbf{q}'\cdot\mathbf{r})d^3\mathbf{q}' \tag{3.35}$$

Where $\hat{\mathbf{q}}'$ is the unit vector pointing in the direction of $\mathbf{q}'$. Now applying the identity to the matrix elements gives

$$\langle\mathbf{k}_f\lambda_f\sigma_f|\hat{V}|\mathbf{k}_i\lambda_i\sigma_i\rangle = \tag{3.36}$$

$$\frac{1}{2\pi^2}\langle\mathbf{k}_f\lambda_f\sigma_f|\sigma\cdot\int\hat{\mathbf{q}}'\times(\mathbf{s}_j\times\hat{\mathbf{q}}')\exp(i\mathbf{q}'\cdot(\mathbf{r}-\mathbf{r}_j))d^3\mathbf{q}'|\mathbf{k}_i\lambda_i\sigma_i\rangle = \tag{3.37}$$

$$\frac{1}{2\pi^2}\langle\lambda_f\sigma_f|\exp(i\mathbf{q}\cdot\mathbf{r})\times\exp(i\mathbf{q}'\cdot(\mathbf{r}-\mathbf{r}_j))\sigma\cdot(\hat{\mathbf{q}}'\times(\mathbf{s}_j\times\hat{\mathbf{q}}'))d^3\mathbf{q}'d^3\mathbf{r}|\lambda_i\sigma_i\rangle = \tag{3.38}$$

$$4\pi\langle\lambda_f\sigma_f|\exp(i\mathbf{q}\cdot\mathbf{r}_j)\sigma\cdot(\hat{\mathbf{q}}'\times(\mathbf{s}_j\times\hat{\mathbf{q}}'))|\lambda_i\sigma_i\rangle = \tag{3.39}$$

$$4\pi\langle\lambda_f\sigma_f|\exp(i\mathbf{q}\cdot\mathbf{r}_j)\sigma\cdot\mathbf{s}_{j,\perp}|\lambda_i\sigma_i\rangle \tag{3.40}$$

In the first step the integration over all the **k** states is performed. The next step used the identity:

$$\int \exp(i(\mathbf{q}+\mathbf{q}')\cdot\mathbf{r})d^3\mathbf{r} = (2\pi)^3\delta(\mathbf{q}+\mathbf{q}') \tag{3.41}$$

Finally the last step made use of,

$$\hat{\mathbf{q}}' \times (\mathbf{s}_j \times \hat{\mathbf{q}}') = \mathbf{s}_{j,\perp} \tag{3.42}$$

where $\mathbf{s}_{j,\perp}$ is the perpendicular component to the scattering vector on site j. This means that the perpendicular component is the only component that contributes to scattering cross section. Since σ is the neutron spin state it depends only on $|\sigma\rangle$ and $\mathbf{s}_{j,\perp}$ is only influenced by the sample state $|\lambda\rangle$ the inner product can be factorized.

$$\begin{aligned}
&\sum_{\sigma_f,\sigma_i} |\langle\lambda_f\sigma_f|\sigma\cdot\mathbf{s}_{j,\perp}|\lambda_i\sigma_i\rangle|^2 = \\
&\sum_{\sigma_f,\sigma_i}\left|\sum_{\alpha}\langle\sigma_f|\sigma^\alpha|\sigma_i\rangle\ \langle\lambda_f|s^\alpha_{j,\perp}|\lambda_i\rangle\right|^2 = \\
&\sum_{\sigma_f,\sigma_i}\sum_{\alpha,\beta}\langle\sigma_i|\sigma^\beta|\sigma_f\rangle\ \langle\sigma_f|\sigma^\alpha|\sigma_i\rangle\ \langle\lambda_f|\mathbf{s}^\alpha_{j,\perp}|\lambda_i\rangle\ \langle\lambda_f|\mathbf{s}^\beta_{j,\perp}|\lambda_i\rangle = \\
&\sum_{\sigma_i}\sum_{\alpha,\beta}\langle\sigma_i|\sigma^\alpha\sigma^\beta|\sigma_i\rangle\ \langle\lambda_f|\mathbf{s}^\alpha_{j,\perp}|\lambda_i\rangle\ \langle\lambda_f|\mathbf{s}^\beta_{j,\perp}|\lambda_i\rangle =
\end{aligned}$$

In the last step the completeness relation is used which says that $\sum_{\sigma_f}|\sigma_f\rangle\langle\sigma_f| = 1$. Then looking at the polarization of the atoms simplifies this further.

$$\begin{cases}\sum_{\sigma_i}\langle\sigma_i|\sigma^\alpha\sigma^\beta|\sigma_i\rangle = 1 & \text{if } \alpha=\beta\\ \sum_{\sigma_i}\langle\sigma_i|\sigma^\alpha\sigma^\beta|\sigma_i\rangle = 0 & \text{if } \alpha\neq\beta\end{cases} \tag{3.43}$$

Applying this result removes all of the sums and gives us the much simpler equation,

$$\sum_{\sigma_f,\sigma_i}|\langle\lambda_f\sigma_f|\sigma\cdot\mathbf{s}_{j,\perp}|\lambda_i\sigma_i\rangle|^2 = \langle\lambda_f|\mathbf{s}_{j,\perp}\cdot\mathbf{s}_{j,\perp}|\lambda_i\rangle \tag{3.44}$$

Now the perpendicular projection can be written as,

$$\mathbf{s}_{j,\perp} = \mathbf{s}_j - (\mathbf{s}_j\cdot\hat{\mathbf{q}})\cdot\hat{\mathbf{q}} \tag{3.45}$$

This is then applied to the dot product,

$$= \mathbf{s}_{j,\perp}\cdot\mathbf{s}_{j',\perp} = \sum_{\alpha,\beta}(\delta_{\alpha\beta}-\hat{q}_\alpha\hat{q}_\beta)s^\alpha_j s^\beta_{j'} \tag{3.46}$$

With this all of the components for the master scattering cross section for neutrons, with magnetism included, are gathered and can be added together[13].

$$\frac{\partial^2\sigma}{\partial\Omega\partial E_f} =(\gamma r_0)^2\frac{k_f}{k_i}\Big(\frac{g}{2}F(\mathbf{q})\Big)^2\sum_{\alpha,\beta}(\delta_{\alpha\beta}-\hat{q}_\alpha\hat{q}_\beta) \tag{3.47}$$

$$\times\sum_{\lambda_f,\lambda_i}p_{\lambda_i}\sum_{j,j'}\langle\lambda_i|\exp(-i\mathbf{q}\cdot\mathbf{r}_j)\mathbf{s}_j^\alpha|\lambda_f\rangle\ \langle\lambda_f|\exp(i\mathbf{q}\cdot\mathbf{r}_{j'})\mathbf{s}_{j'}^\beta|\lambda_i\rangle \tag{3.48}$$

$$\delta(\hbar\omega+E_{\lambda_i}-E_{\lambda_f}) \tag{3.49}$$

Where $F(\mathbf{q})$ is the magnetic form factor and is defined as

$$F_m(\boldsymbol{q})=\int\exp(i\boldsymbol{q}\cdot\boldsymbol{r})\rho_s(\boldsymbol{r})d^3\boldsymbol{r} \tag{3.50}$$

with ρ_s the normalized spin density in the unfilled orbitals. Just like in the inelastic calculation the δ function can be transformed into an integral by using the time dependent Heisenberg operators. Then using the completeness identity for the final states $|\lambda_f\rangle$ the following cross section is achieved,

$$\frac{\partial^2\sigma}{\partial\Omega\partial E_f} =(\gamma r_0)^2\frac{k_f}{k_i}\Big(\frac{g}{2}F(\mathbf{q})\Big)^2\sum_{\alpha,\beta}(\delta_{\alpha\beta}-\hat{q}_\alpha\hat{q}_\beta) \tag{3.51}$$

$$\times\frac{1}{2\pi\hbar}\sum_{j,j'}\int_{-\infty}^{\infty}\exp(-i\omega t) \tag{3.52}$$

$$\times\Big\langle\exp(-i\mathbf{q}\cdot\mathbf{R}_j(0))s_j^\alpha(0)\exp(-i\mathbf{q}\cdot\mathbf{R}_{j'}(t))s_{j'}^\beta(t)\Big\rangle\,dt \tag{3.53}$$

Here $\mathbf{R}$ is the nuclear position but this can be pulled outside of the thermal average by replacing it with $\mathbf{r}$, the nuclear equilibrium position.

$$\frac{\partial^2\sigma}{\partial\Omega\partial E_f} =\frac{(\gamma r_0)^2}{2\pi\hbar}\frac{k_f}{k_i}\Big(\frac{g}{2}F(\mathbf{q})\Big)^2\sum_{\alpha,\beta}(\delta_{\alpha\beta}-\hat{q}_\alpha\hat{q}_\beta) \tag{3.54}$$

$$\times\sum_{j,j'}\int_{-\infty}^{\infty}\exp(-i\omega t)\exp(i\mathbf{q}\cdot(\mathbf{r}_{j'}-\mathbf{r}_j))\Big\langle s_j^\alpha(0)s_{j'}^\beta(t)\Big\rangle\,dt \tag{3.55}$$

By going back to the semi classical representation $\mathbf{s}$ can be represented as vector again instead of an operator. Then dropping out all of the prefactors gives the scattering function $S^{\alpha,\beta}$.

$$S^{\alpha,\beta}(\mathbf{q},\omega)=\sum_j s_j^\alpha(0)\exp(-i\mathbf{q}\cdot\mathbf{r}_j)\sum_{j'}\int_{-\infty}^{\infty}\exp(i\mathbf{q}\cdot\mathbf{r}_{j'})\mathbf{s}_{j'}^\beta(t)\exp(i\omega t)dt \tag{3.56}$$

This is the equation that will be simulated and will show the dispersions. Since the prefactors are dropped out the absolute values will not be comparable but it will only differ by a constant prefactor. In most cases $\alpha=\beta$ will hold since sums over α and β will cancel out due to the anti symmetric property of the scattering function $S^{\alpha,\beta}=-S^{\beta,\alpha}$.

Chapter 4

Computational

4.1 Integration

To solve integrals numerically the midpoint method is used[14]. Most numerical integration methods are a variation on Euler's method. the most common ones are the midpoint method (2nd order) and the Runge-Kutta method (4th order). Here the midpoint method is used since precision is not a huge priority since every iteration will be perturbed by the random temperature fluctuation. Since more precision comes at the cost of computational speed the midpoint method is a good balance. The midpoint method works as follows,

$$y_{n+1} = y_n + hf\left(t_n + \frac{1}{2}h, \frac{1}{2}(y_n + \frac{1}{2}hf(t_n, y_n))\right) \tag{4.1}$$

where $y'(t) = f(t, y(t))$. First the differential equation is evaluated at $t + \frac{1}{2}h$ to give the slope. Then the point to be approximated (y_{n+1}) can be determined by moving one step h along this slope from the previous point (t_n). A visual explanation of this is given by figure 4.1.

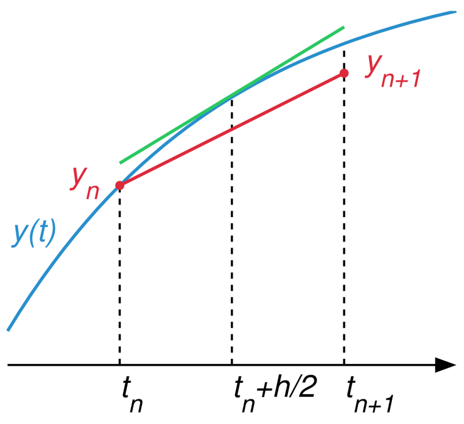


Figure 4.1: An illustration of the midpoint method where $y(t)$ is integrated from n to $n+1$. From wikipedia[15].

4.2 Temperature

In section 2.7 the relation for the random $\boldsymbol{b}$ field was derived. Now it needs to be implemented numerically. To do this the variance needs to be found such that at each iteration for each spin a random field can be generated that represents the temperature fluctuations. To do this the time evolution of equation 2.61 is discretized in time.

$$\boldsymbol{s}_i(t_N) - \boldsymbol{s}_i(t_0) = \int_{t_0}^{t_N} \frac{\partial \boldsymbol{s}_i}{\partial t} dt = \sum_{j=0}^{N-1} \int_{t_j}^{t_{j+1}} \frac{\partial \boldsymbol{s}_i}{\partial t} dt \tag{4.2}$$

The initial condition for $\boldsymbol{s}_i(t_0)$ is given which allows 4.2 to be rewritten as:

$$\boldsymbol{s}_i(t_n) = \boldsymbol{s}(t_0) + \sum_{t=0}^{n} \Delta \boldsymbol{s}_i(t) \tag{4.3}$$

With:

$$\Delta \boldsymbol{s}_i(t_n) = \int_{t_n}^{t_{n+1}} \frac{\partial \boldsymbol{s}_i}{\partial t} dt \tag{4.4}$$

Then the goal becomes to find a proper approximation for the terms $\Delta \boldsymbol{s}_i(t_n)$. To approximate the random term $\boldsymbol{b}(t)$ it is assumed that on this time interval the $\boldsymbol{s}_i(t)$ barely has any time dependence compared to $\boldsymbol{b}(t)$. This allows the random term in the equation of motion to be rewritten as

$$\begin{aligned}\Delta \mathbf{f} &= \int_{t_n}^{t_n+1} \gamma \boldsymbol{s}_i(t) \times \boldsymbol{b}(t) dt && (4.5)\\ &= \gamma \boldsymbol{s}_i \times \int_{t_n}^{t_n+1} \boldsymbol{b}(t) dt && (4.6)\\ &= \gamma \boldsymbol{s}_i \times \Delta \boldsymbol{b} && (4.7)\end{aligned}$$

with

$$\Delta \boldsymbol{b} = \int_{t_n}^{t_n+1} \boldsymbol{b}(t) dt \qquad (4.8)$$

The temperature fluctuation can be seen as a Gaussian random walk. According to random walk theory the variance is proportional to t. In the case for $\Delta \boldsymbol{b}(t)$ the time interval is:

$$t_{n+1} - t_n = \Delta t \qquad (4.9)$$

And therefore the variance will scale with Δt. The total variance can now be calculated from the correlation function:

$$\begin{aligned}< \Delta \boldsymbol{b}_i(t) \Delta \boldsymbol{b}_j(t') > &= \int_{t_n}^{t_n+1} \boldsymbol{b}_i(t) dt \int_{t_n}^{t_n+1} < \boldsymbol{b}_j(t') dt' > && (4.10)\\ &= \int_{t_n}^{t_n+1} \int_{t_n}^{t_n+1} < \boldsymbol{b}_i(t) \boldsymbol{b}_j(t') > dt dt' && (4.11)\\ &= D\delta_{ij} \int_{t_n}^{t_n+1} \boldsymbol{b}_i(t) dt \int_{t_n}^{t_n+1} \delta(t-t') dt dt' && (4.12)\\ &= D\delta_{ij} \int_{t_n}^{t_n+1} dt && (4.13)\\ &= D\delta_{ij} \Delta t && (4.14)\end{aligned}$$

To implement this in the software the following steps are taken. For every time step first generate N random numbers from a Gaussian distribution with variance equal to $D\delta_{ij}\Delta t$. Where N is the system size and Δt the time step. Then calculate the deterministic part of the equation and to this add the temperature fluctuations.

4.3 Coordinate system

A coordinate system is needed to represent the spin vectors and there are two options to consider. On the one hand there is the standard Cartesian coordinate system and on the other there is the spherical coordinate system.
The first intuition would be to choose the spherical coordinate system. Since the magnitude of the spin is constant it moves on the surface of a sphere. For a spherical coordinate system this means that the r coordinate is constant and the system effectively becomes 2D. This reduces the amount of memory needed

to store the orientations of the spin as well as having one less component to calculate. It is also an answer to one of the numerical problems that show up where the length does not stay constant due to integration errors and therefore normalization has to be applied at every time step (This will be discussed in more detail in section 5.1).

However there are two big flaws. The first has to do with the fact that there is a singularity when the moment aligns with the z-axis. This is because when the azimutal angle φ is zero the polar angle can take on any value. This problem showed up both when implementing this in the current CLaSSiC as well as in the previous[3] iteration by producing inconsistent results. The second issue has to do with speed. The equation of motion (equation 2.21) has a cross products and cross products are highly non-trivial in spherical coordinates. To do the cross product in spherical coordinates first the system is written in Cartesian coordinates, then the regular cross product is taken and finally it is transformed back to spherical coordinates. The equation of motion in spherical coordinates can be found in the appendix C. As can be seen this method makes use of a lot of trigonometric functions which are much slower then simple arithmetic functions. Both of these problems weigh much heavier then possible benefit of reducing the dimensionality of the problem as well as no longer needing to normalize.

4.4 Python vs C++

The original version of the model was written in Matlab and later moved on to Python and is now written in C++. This chapter will explain why moving the code to C++ makes the simulation time much shorter.
The first thing to note is that Python is an interpreted high level language while C++ is a compiled lower level language. So first interpreted vs compiled[16]. When a Python program is run it will look at a line, execute it and move on to the next one. On the other hand a compiler will look at all of the code and make a set of instructions called an executable, which is then be run. The big advantage is the compiler can use optimizations a lot better since it can for example predict what memory might be need later since it can look ahead. A disadvantage of compiled languages is that compiling can be slow, especially for larger projects.

Next low vs high level languages[17]. The lower level a programming language is the more the programmer has to do. This is both a good thing and a bad thing. The good part is that now there is a lot more control over the resources. However with the drawback is that ease of use goes down. For example when declaring a variable in Python there is no need for a type declaration (int, float, etc) while in C++ you must declare it beforehand and cannot change this afterwards. This makes for more difficult programming but allows for faster runtimes since now there is no need to check if the data type has changed.

4.5 Data structures

The next important thing in optimizing runtime is making sure the data is laid out efficiently. To understand what makes a good data structure a short explanation of memory architecture will be needed. To start of the storage is divided up into multiple tiers where the lower tiers have much higher access speed but that is offset by their size. For example the lowest tier data storage is the L1 data cache which is 128 KB in size and has access speed of 700 GB/s this in comparison to the ram which will have a size on the order of 10 GB and an access time of 10 GB/s.

When the CPU needs something from memory it will ask the memory controller for that piece of data. The memory controller is smartly designed however and gets more of the data than needed. The idea is that the CPU will not only need that single data point but also some of the surrounding data. To take full advantage of the it is important to put all of the data that is needed for a calculation together. In this case that means a structure where there is an separate array for each of the three Cartesian coordinates since most of the calculations involve only one coordinate at the time and thus multiple atoms can be done simultaneous.

4.6 Parallelization

One way to increase performance is by running tasks in parallel. Theoretically the speedup will be equivalent in how many ways the task is split. Most modern computers have 8 cores per CPU, that would mean a 8x increase in speed. However there are also GPUs which are specifically build for massively parallel tasks. They will have over a 1000 cores thus allowing for equally massive speedups. However for parallelism there is always an overhead thus there needs to be enough number crunching to make it worth it.

When talking about parallelizing the first thing one looks at are the parts that are pleasingly parallel. These are parts of the code where there is only sequential data dependence and it is just pure calculation. An example of this in CLaSSiC would be the random number generation for the temperature deviations. Here an array equal to the system size is filled with random numbers. These numbers have no dependence on each other so they can be generated independently, which means the array can be split among the CPU cores and each core can do a different part of the array. The issue here is in the fact that setting up parallelism has overhead and it is only as fast as the slowest core since it has to wait until every core is finished before it can continue. Otherwise race conditions may happen which produces strange results. Thus for small systems it is not worth doing this kind of parallelism. To break even in the random number generation case the array size would have to be on the order of 10.000 elements and right now the systems that are run with CLaSSiC are more on the order of a 1000.

Then there are the parts of the code that can be parallelized but the cores would need to communicate between each other since there is data dependence. This can be a very big bottleneck as well as making implementation much harder. Ideally the whole integration part would be parallelized since that is where all of the work is done but when calculating the effective field there is the term concerning the exchange interaction. Here the core would have to have the data of the neighbours. One could implement this smartly so that most of the neighbours are done by the same core but there will always be some that are on the border between two CPU cores. This would require all of the cores to synchronize after each evaluation in the midpoint method giving up a lot of the potential speed up. This has not been implemented since the systems that are simulated where already too small to parallelize random number generation thus it is expected the same would hold in this case.

Chapter 5

Validation

Chapter 5 will cover over the results from the model and compare them to the theory. This comparison is performed to validate the code for errors and its correspondence with the theory. All of the simulation parameters are given in table B.1. All of the data is available upon request.

5.1 Zeemann interaction

As was seen in section 2.1, a single spin in a magnetic field will precess around the magnetic field. The spin should have a periodic motion with an energy equal to:

$$\hbar\omega = -\hbar\gamma B \tag{5.1}$$

For a magnetic field of 5 T the theoretical energy is $E = 0.579509$ meV and the simulation gives $E = 0.578993$ meV so there is an error of $\Delta E = 0.0005$ meV.

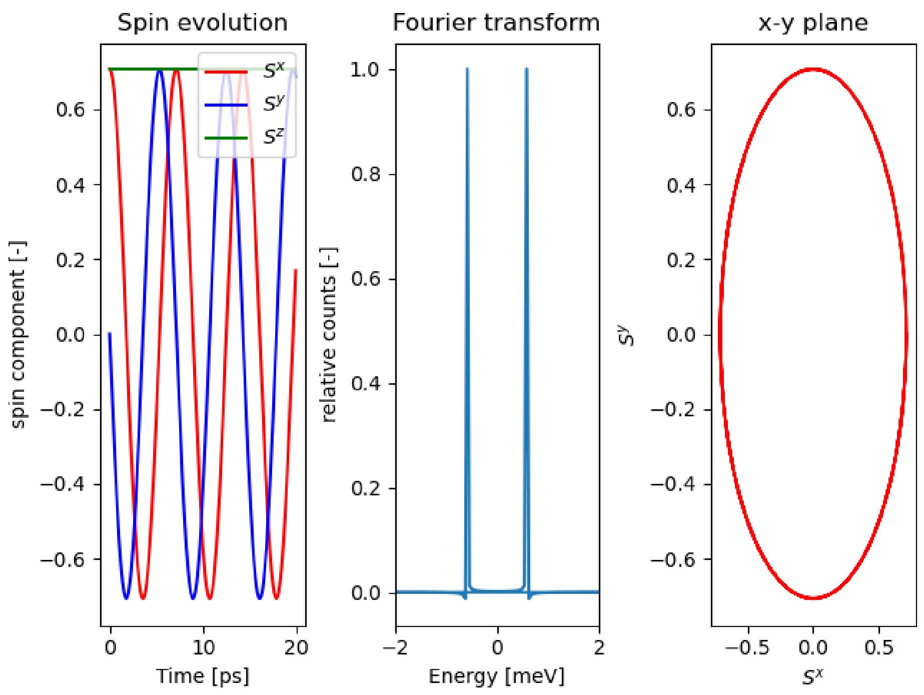


Figure 5.1: Single spin in 5 T magnetic field in z-direction. The first figure shows the time evolution, the second the real part of the Fourier transform and the last the time evolution in the x-y plane.

Figure 5.1 shows that the spin indeed rotates around the z-axis with a constant speed. One thing to note that cannot be seen from the figure, is that the system loses a little bit of energy since S^z goes slightly down, however this is on the order of 1×10^{-8} over the whole duration of the simulation which is 1 ns. This scales linearly with time so if it is run for 10 ns it will be on the order of 1×10^{-7}. This is caused by small integration errors. When doing the integration, the estimation will always be slightly too large which makes the spins a little bit longer. This is an unphysical effect and to combat this renormalization is introduced. When the spin is renormalized the orientation is changed slightly leading to a small error. When the system is then integrated over a long time the error accumulates. When the system is run without renormalization the z-difference is zero.

5.2 Anisotropy

When there is anisotropy in a single spin system the spin will precess around the anisotropy axis with the energy depending the angle with the perpendicular axis. This can be seen by rewriting the dot product in equation 2.9 to the cosine.

$$\hbar\omega = g\mu_B B_{anis}\cos(\theta) \tag{5.2}$$

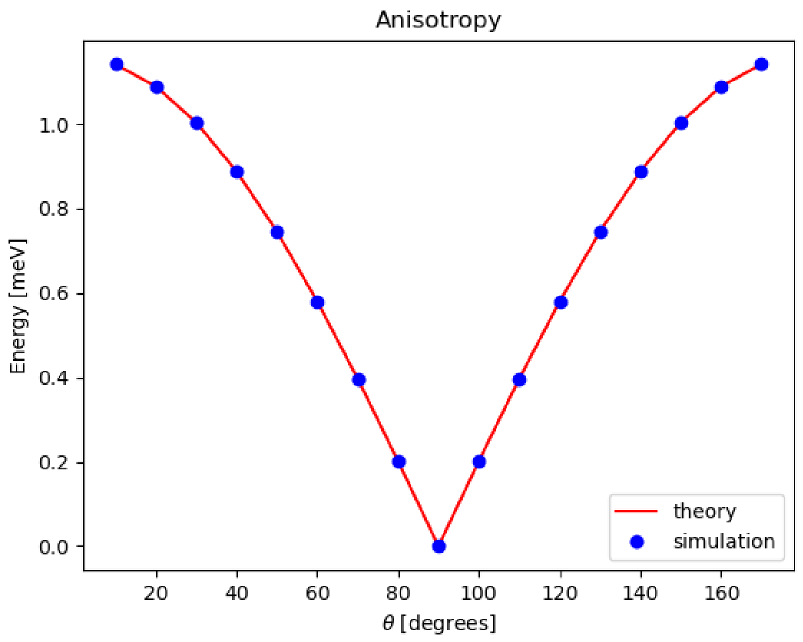


Figure 5.2: Single spin anisotropy with strength $B_{anis} = 10$ T in the z-direction. Here θ is the angle with respect to the z-axis, also known as the polar angle.

Figure 5.2 shows that the theory follows the simulation well. At the tip of the v-shape the spins are perpendicular to the easy axis. The more they align with the anisotropy axis the faster they will spin.

5.3 Temperature

Temperature introduces fluctuations in the spin orientation. The average of the spin orientation along the z-axis is given by the Langevin curve as was seen in equation 2.54:

$$\frac{<\boldsymbol{\mu}^z>}{|\boldsymbol{\mu}|} = L(y) = coth(y) - \frac{1}{y} \tag{5.3}$$

with

$$y = \frac{\mu B_z}{k_b T} \tag{5.4}$$

The simulation seen in figure 5.3 is run for 100 ns with dissipation term $\lambda = 0.001$ and starts aligned with the magnetic field. Note that the figure average values for the spin components over the last 10% of the simulation. Faster conversion can be achieved by taking a smaller window to average over or by increasing λ.

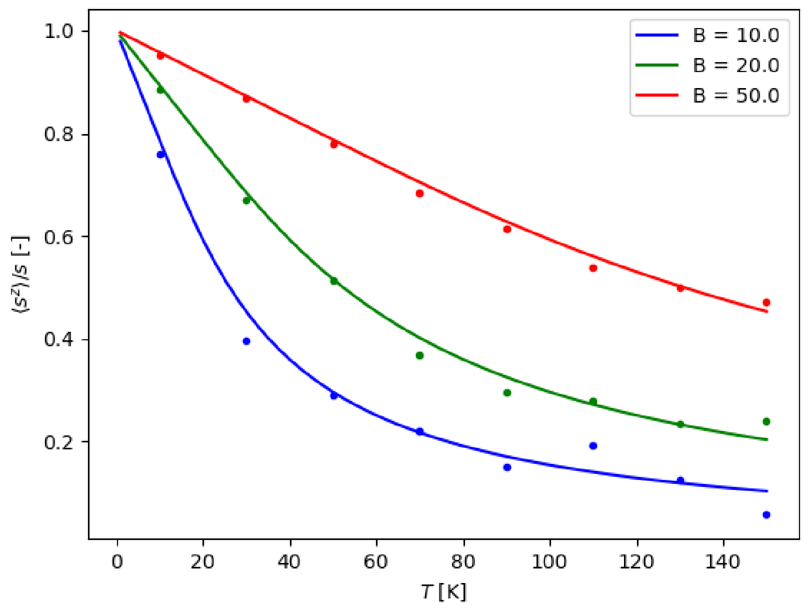


Figure 5.3: Average position for a single spin in various magnetic fields. The lines represent the theory and the points are the data produced by the simulation.

5.4 Rotor

The simplest case to test the exchange interaction is with two spins pointing anti parallel. They will start to rotate and the speed depends on the angle they make with x-y plane. The relation is

$$\hbar\omega = 2Js(2\sin(\theta) - 1) \tag{5.5}$$

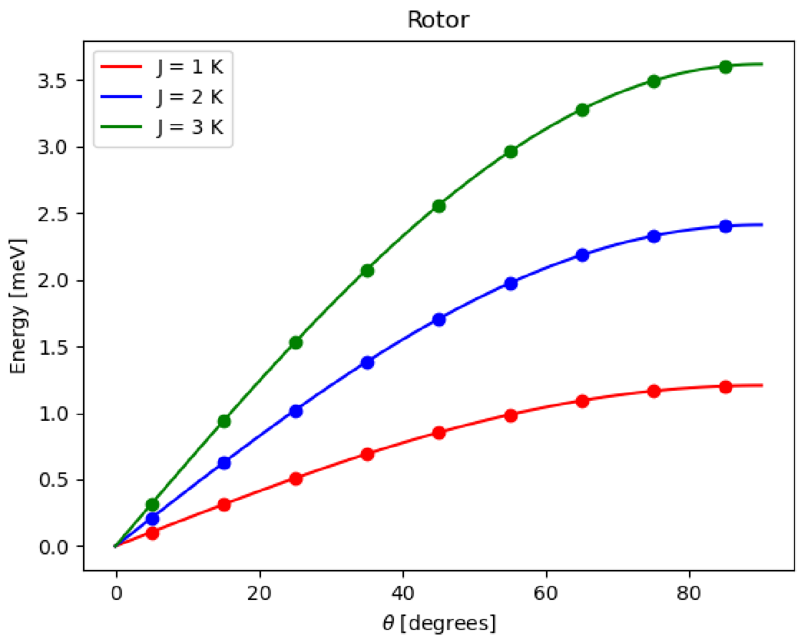


Figure 5.4: Two spins start out anti parallel to each other and are then rotated by an angle with respect to the x-y plane. This is shown for various exchange constants. The lines are equation 5.5 and the points are the data produced by the simulation.

As can be seen in figure 5.4, the spin will rotate faster for larger angles. The energy is evaluated by performing a Fourier transform thus, the 90° point is not included since the spins would not rotate.

5.5 Ferromagnetic spin chain

Figures 5.5 and 5.6 show the dispersion for a ferromagnetic spin chain where the x-axis is a path through the high symmetry points in k-space. The data points come from the peaks of the scattering function $\mathbf{S}^{\alpha\beta}(\mathbf{k}, \omega)$.

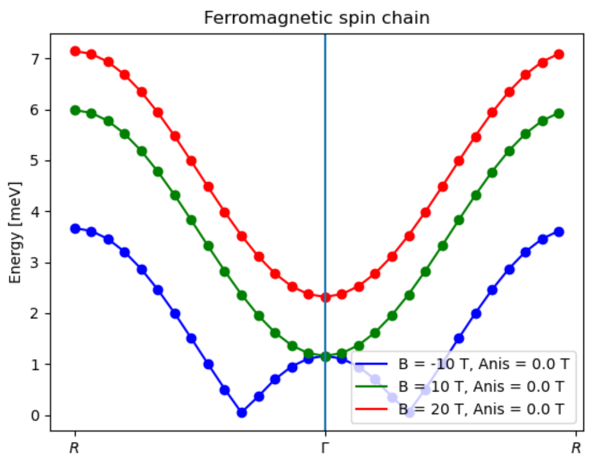


Figure 5.5: The dispersion of a ferromagnetic spin chain for various magnetic fields.

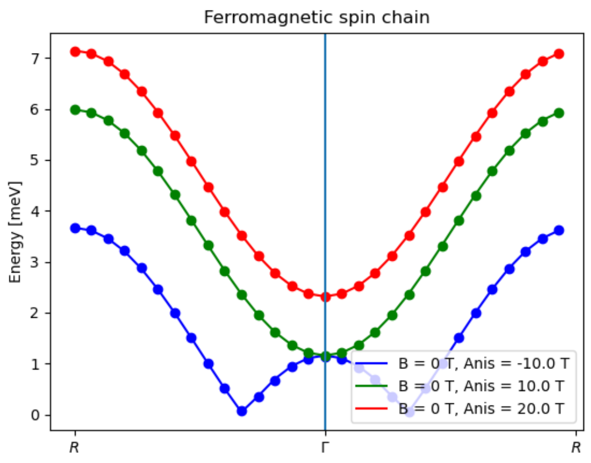


Figure 5.6: The dispersion of a ferromagnetic spin chain for various anisotropies.

The origin of the peaks are determined can be seen in figure 5.7. Here the top plot represents the same dispersion but now the dots are colored based on their intensity. The bottom plot shows the scattering function $\mathbf{S}^{\alpha\beta}(\mathbf{k}, \omega)$ at the red line in the top plot. As can be seen in figures 5.7 and 5.8 the peaks are very distinct, but the intensities vary wildly. However the peaks are still in the proper place, corresponding with the prefactors being dropped out, as explained in section 3.4. One of these, the magnetic form factor, has a dependency on **k** and might explain the results since it varies from point to point in k-space. As a side note the energy axis goes on for much longer but is cut off because there are no effects that have such high energy.

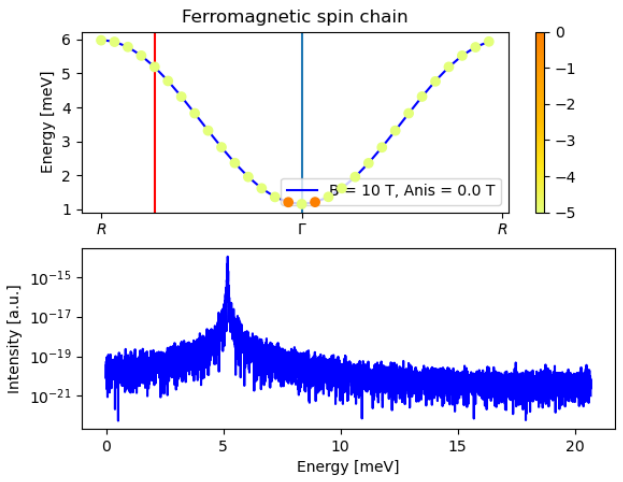


Figure 5.7: Intensities at various points in k-space for a spin chain started in the first mode. The scattering function is given at the 4th point in k space.

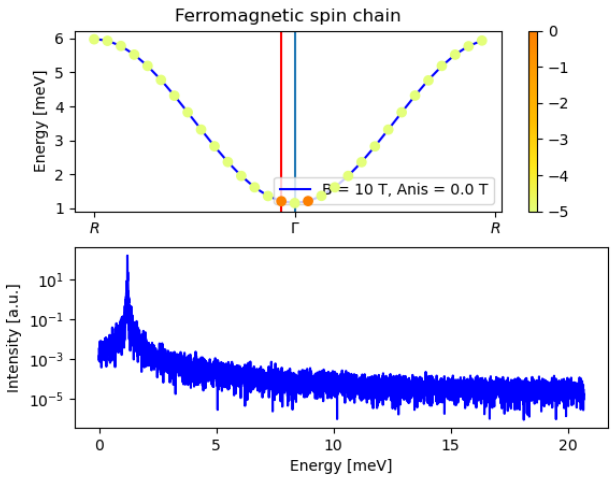


Figure 5.8: Intensities at various points in k-space for a spin chain started in the first mode. The scattering function is given at the 14th point in k space.

5.6 Antiferromagnetic spin chain

The correct dispersion is also found in the antiferromagnetic case as figures 5.9 and 5.10 show but again the scattering function shows a big variation in intensity. It should be noted that the antiferromagnetic system is rather temperature sensitive, especially when introducing a magnetic field making it hard to get the correct dispersion. But for the low temperature ($T = 1 \times 10^{-16}$) simulated here it produces good results.

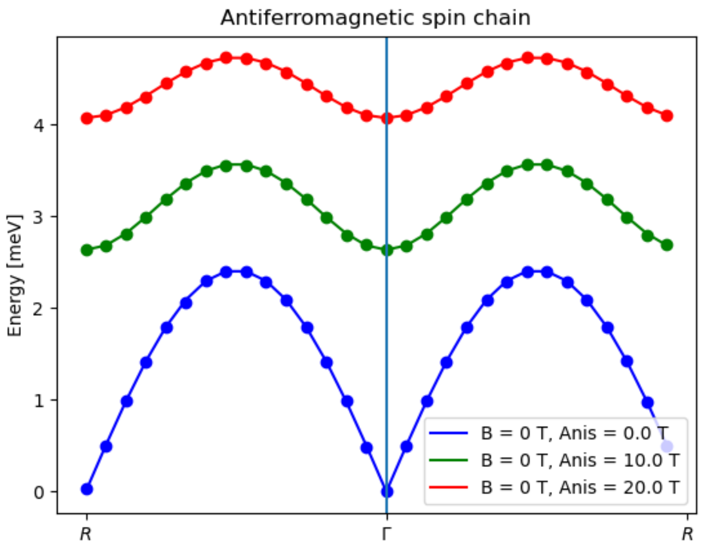


Figure 5.9: Antiferromagnetic chain with various easy axis anisotropies.

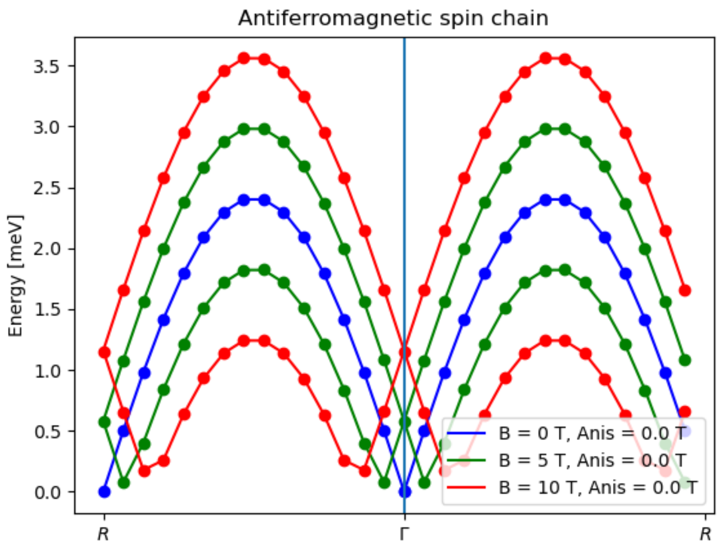


Figure 5.10: Antiferromagnetic chain with various easy axis magnetic fields.

When simulating the antiferromagnetic chain with zero temperature and no dissipation it produces additional peaks depending on the mode the system was started in. This can be seen in figure 5.11, where it clearly shows that the maximum does not follow the dispersion, however the peak corresponding to the dispersion is still there. This effect disappears when small temperature effects are introduced as figures 5.9 and 5.10 show. Thus this is most likely a result of numerical errors in the starting position.

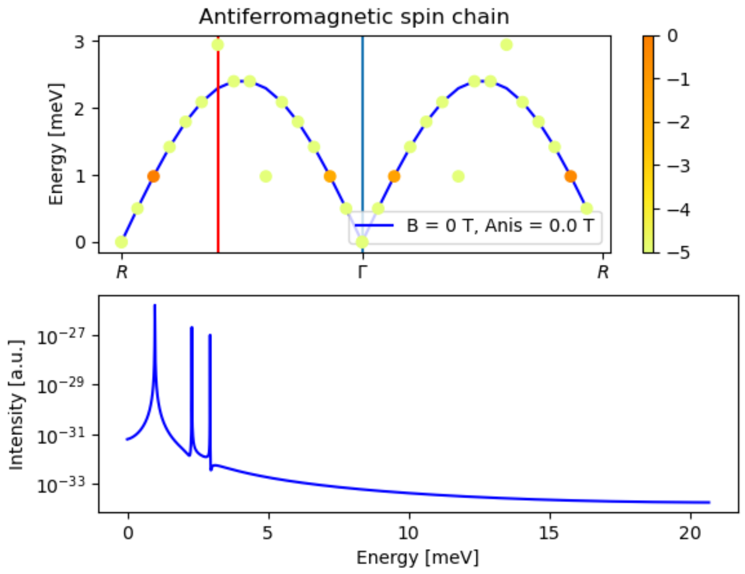


Figure 5.11: Antiferromagnetic chain with $T = 0$ and $\lambda = 0$.

Chapter 6

Benchmark

6.1 Setup

All of these benchmarks are run on a HP Z-book. It has an Intel(R) Core(TM) i7-7700HQ CPU which has a base clock speed 2.80 GHz and can turbo to 3.80 GHz. For the memory there is a single stick of 8GB, 2400 MHz Hynix DDR4 RAM. For reference these specs are decent but not top of the line and will usually be slower than desktop equivalents. This is because laptops can use only limited power since cooling is a major concern. So a more top of the line CPU will be able to boost to 5.0 GHz and RAM speeds can be up to 3200 MHz. Of course if enterprise hardware is taken into account more is possible.

6.2 Scaling

The model is expected to scale linearly with all parameters that change the amount of computation. These are the number of time steps and geometry, which affects number of nearest neighbours as well as the number of atoms in the system. Each time step is independent from the previous and it will perform the same calculations every time step. Thus it is expected to scale linearly. Figure 6.1 shows that it indeed scales linearly with the number of time steps.

When the geometry is varied it influences the number of calculation per iteration. Changing the number of nearest neighbours only has affect on the calculation for the exchange field. The number of calculation will scale as nA with n the number of neighbours and A the number of atoms and is thus linear. The results are shown in figure 6.2 and show indeed a linear dependence.

Lastly for different numbers of atoms. The mean-field will simply have to be calculated and applied more often but since a single calculation does not depend

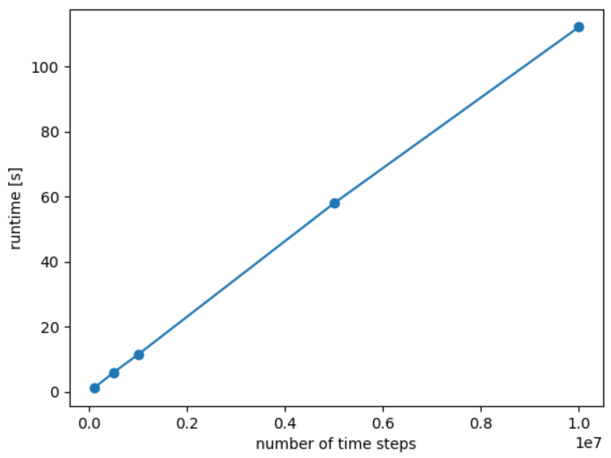


Figure 6.1: The runtime for different values of time steps. The system size was 100 atoms in the line geometry.

on the number of atoms it is linear as well. This not very clear from figure 6.3 since it is only approximately linear. However there are not that many data points and they are very close together. So fluctuations in the runtime will have a big effect on this plot.

The model being linear is a very nice property since it means that making bigger or more complex systems does not take that much additional time. If for example all spins could interact with each other, it would be become a N-body problem. This has n^n scaling which means increasing system size is extremely costly.

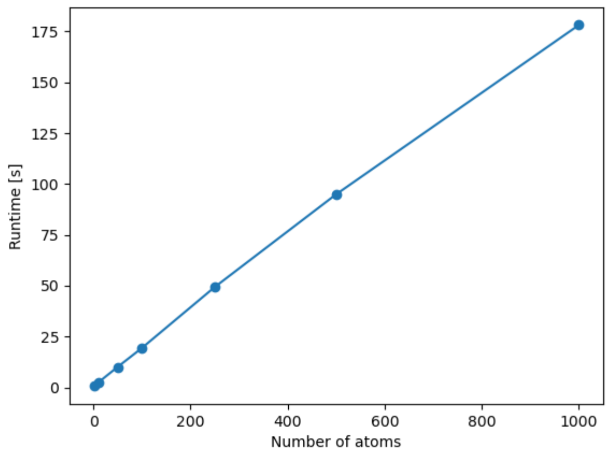


Figure 6.2: Benchmark for the line geometry for runs with 1.000.000 time steps and temperature enabled.

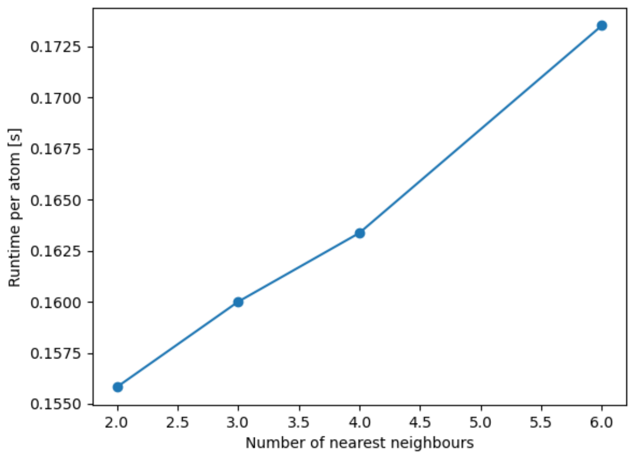


Figure 6.3: This figure shows the speed dependency on the number nearest neighbours. The systems are run for 1.000.000 time steps, temperature enabled and 100 atoms.

6.3 Intel VTune

Intel VTune[18] is a profiling tool for computer programs in a variety of languages. It gives insight in where the most time is spent, how well it parallelizes and where potential bottlenecks are. The tool is free and will work for both Intel and AMD CPUs. The next sections are based on an article by Intel called the Intel VTune Profiler Performance Analysis Cookbook[19]. It will explain what the measurements mean, where they come from and how they should be interpreted. All measurements are done for a line geometry with 500 atoms run for 1.000.000 time steps unless noted otherwise.

6.3.1 Background

The micro architecture is the inside of a CPU which has many different modules for different tasks. For example there is the division unit which handles division and the ALU (Arithmetic Logic Unit) which handles, as the name suggests, arithmetic and logic. A good way to analyze the efficiency of the hardware is by looking at pipelines.

First of all a pipeline is a chain of processes that are executed one after another where the output of one process is the input for the next. The process can be divided into two halves, front end and back end. The front end takes the software instructions and decodes them into hardware level operations called micro-operations (μOps). Then it moves these to the next step in the pipeline which is the back-end, this step is called allocation. The back-end is responsible for executing the μOps so it will fetch the required data and send it to an appropriate execution unit. If an execution is successful is said to have been retired. However not all of the micro-operations will be retired, this is because the front-end will speculate on future muOps and memory and these speculations will not always turn out to be correct.

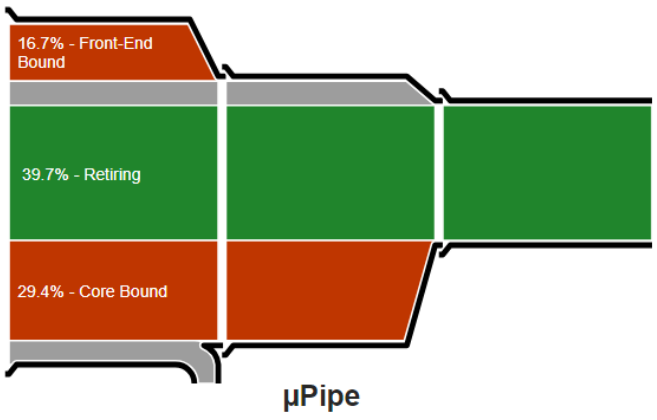


Figure 6.4: A μOps pipeline produced by intel VTune with the minGW compiler. From the top to the bottom there is Front-End Bound (red), memory bound (gray), retiring (green), Core bound (red) and lastly bad speculation (gray).

The pipeline in figure 6.4 is created in the following way. Every CPU cycle the front-end can allocate a set amount of μOps and the back-end can also process a set amount. Then a pipeline slot will be defined as the requirements for processing a single μOp. Each slot can be divided into 4 categories. If, for a given cycle, the slot does not have a μOp it will classified as a stall. If the front-end cannot provided μOps fast enough the stall is denoted as front-end bound. If it is empty because the back-end cannot accept a new μOp yet it will be classified as back-end bound. Then if the pipeline slot is not empty it will be classified one of two ways depending on if it will be retired or not. If it is retired eventually it will be classified as retired and otherwise it will be classified as bad speculation.

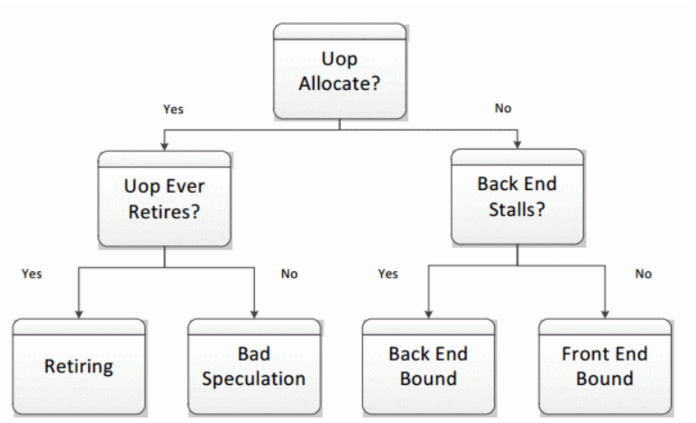


Figure 6.5: The decision tree for the categorization of pipeline slots. From Intel VTune Profiler Performance Analysis Cookbook[19].

Before talking about how to optimize these categories it is important that the focus will be on so called hotspots. Hotspots are the points in the software that use the most CPU cycles. Optimizing these parts of the code gives the biggest performance increase for the amount of work since a potential improvement will affect a larger portion of the run time. It should be noted that looking at hotspots is only relevant after any parallelization has been done as well as any algorithmic tuning. This is because otherwise the hotspots might change if the algorithm is changed and therefore tuning a certain section might not be worth it anymore.

6.3.2 Possible optimization

Most of the unoptimized programs turn out to be back-end bound. This is usually due to latency issues where the retirement process takes much longer than needed because it needs to wait on resources. There are two types of resources it might need to wait on, memory and execution units. To solve memory bound issues one would need to make sure that the required data is closer to the core. Since the closer it is the faster it can be accessed. Latency in execution units is called core bound and happens when there is only a very small amount of data on which a lot of computation is done. For example there are only so many divider units available in the CPU so doing many divisions will overload the units and causes stalls. This can be solved by spreading the work out if possible.

Front-end bound problems occur less often and are seen more in interpreted languages such as Python. This is because the instructions are created on the go and can therefore not be optimized. However the front-end also does speculation on what μOps it should create. This is mostly caused by writing lengthy code with a lot of if-else statements. This makes for a lot of possible ways the code can run since for each if-else statement it can branch one of two ways. Thus the compiler cannot properly predict what will happen. To solve this, reduce the complexity of the code and the amount of if-else statements. Also different compilers may produce different results.

6.3.3 Compiler choice

The main compiler choice for this project was the minGW compiler since it is open-source and works for both AMD and Intel platforms. However Intel VTune came with its one compiler so it seemed like a good idea to compare the two. Figure 6.4 shows the pipeline for the minGW compiler while figure 6.6 shows the results for the intel compiler. As it turns out the Intel compiler is much better at optimizing the front-end as it almost completely removes the front-end bound bottleneck. For minGW it accounts for 16.7% and in the Intel case it is only 2.5%. This is also seen in the run time of the program which is approximately twice as fast when compiled by the Intel compiler.

However where it does most of these optimizations is hard to say since the function naming was not working for the minGW compiled program. This means that it gives only the function pointer, which looks something like: func@0x1402324e7. Thus comparing the on a function basis is not realistic. For the rest of the analysis the Intel compiler is used since there the function naming works properly.

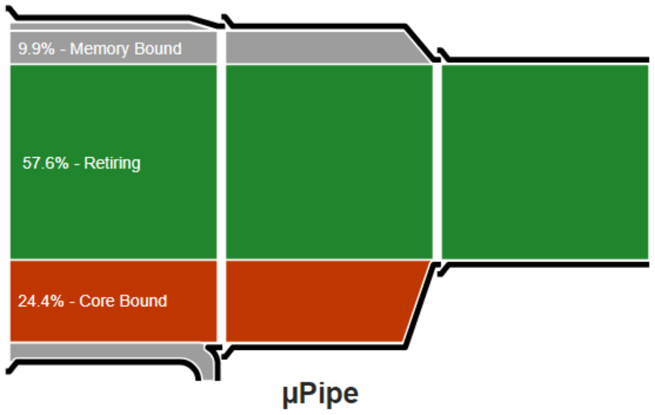
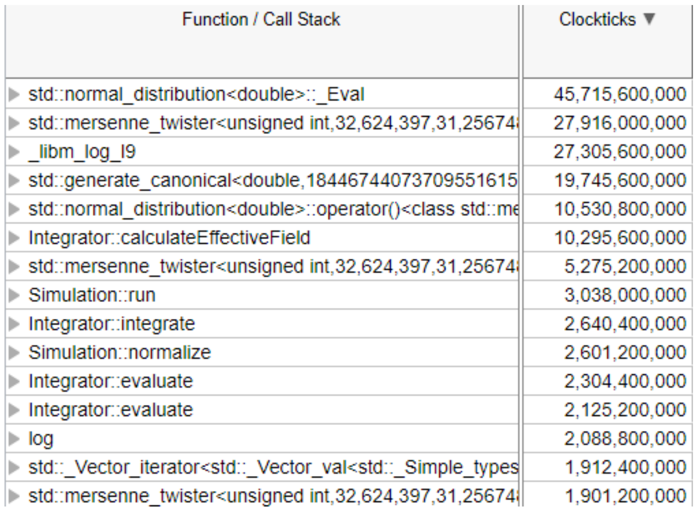


Figure 6.6: A μOps pipeline produced with the Intel compiler.

6.4 Hotspot analysis

Now lets take a look at the hotspots in the code. Figure 6.7 shows the top 15 functions sorted by the number of clock cycles spend in them. The following section will go through each of the functions listed here.



Function / Call Stack	Clockticks ▼
▶ std::normal_distribution<double>::_Eval	45,715,600,000
▶ std::mersenne_twister<unsigned int,32,624,397,31,25674	27,916,000,000
▶ _libm_log_l9	27,305,600,000
▶ std::generate_canonical<double,18446744073709551615	19,745,600,000
▶ std::normal_distribution<double>::operator()<class std::me	10,530,800,000
▶ Integrator::calculateEffectiveField	10,295,600,000
▶ std::mersenne_twister<unsigned int,32,624,397,31,25674	5,275,200,000
▶ Simulation::run	3,038,000,000
▶ Integrator::integrate	2,640,400,000
▶ Simulation::normalize	2,601,200,000
▶ Integrator::evaluate	2,304,400,000
▶ Integrator::evaluate	2,125,200,000
▶ log	2,088,800,000
▶ std::_Vector_iterator<std::_Vector_val<std::_Simple_types	1,912,400,000
▶ std::mersenne_twister<unsigned int,32,624,397,31,25674	1,901,200,000

Figure 6.7: The top 15 functions sorted by the number of CPU cycles.

6.4.1 Temperature analysis

The first thing that sticks out is, that the functions in the top 5 all deal with generating numbers from a Gaussian. This should be obvious from the std::normal_distribution functions but std::mersenne_twister is a random number generator which makes use of std::generate_canonical to produces proper random numbers. The only place where random number generation is used is for the temperature effects. To verify this the program was run with zero temperature which disable the random number generation. The execution time went from 55 seconds to 10 seconds. So the potential gain here is massive.

So where do these issues come from? The biggest issue is back-end core bound. This is partly because std::normal_distribution<double>::operator() hits the division units hard. Over 75% of the cycles is spent division, which is a lot. Then the other are using scalar operations and not vector operations therefore not using the CPU to its full potential. The front-end issues are much smaller. Noteworthy are that std::normal_distribution<double>::_Eval 21% of its branches are mispredicted and std::normal_distribution<double>::operator() is memory bound by the L1 cache.

There are several ways in which this could in theory be optimized but it should be noted that this a library function. Therefore it is quite likely that it has already been optimized to some degree and further optimization will most likely not give big improvements. However there are multiple things to consider. First of all is the choice of algorithm. The current algorithm produces high quality random numbers so choosing a different algorithm could be faster at the cost of quality. Another way to improve in this area is by applying the temperature effects only once every 10 or a 100 iterations. This would massively reduce the time spent to generate numbers from a Gaussian. The variance for this Gaussian contains a factor *dt* and can thus be adjusted accordingly. However in both cases it should be checked that the proper results are still produced.

6.4.2 Code analysis

Next up are the functions that are actually written by myself. However to properly benchmark this a large enough number of clock cycles need to be spend in these functions to give accurate results. In the run shown in the previous section almost all time is spent in the random number generation section and therefore not enough is spent in the rest. So the evaluation was run again but this time with zero temperature and for 10.000.000 times steps. The new pipeline can be seen in figure 6.8 and the hotspots in figure 6.9. A quick look at the pipeline shows that now a bigger percentage of the μOps get retired. There is also lower amount of pipeline slots that are core bound and the amount of bad speculation is significantly reduced (the gray area at the bottom). This is expected since most of these issue stem from the Guassian number generation which is now removed.

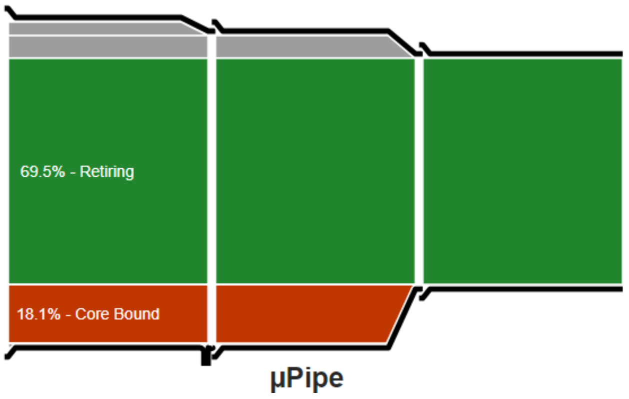
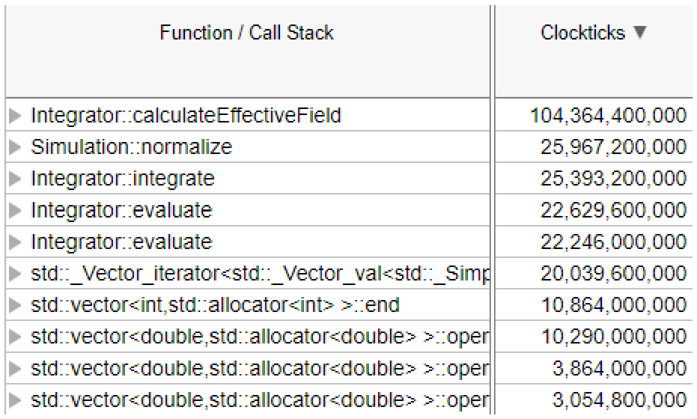


Figure 6.8: The μOps pipeline compiled by the Intel compiler for zero temperature.



Function / Call Stack	Clockticks ▼
▶ Integrator::calculateEffectiveField	104,364,400,000
▶ Simulation::normalize	25,967,200,000
▶ Integrator::integrate	25,393,200,000
▶ Integrator::evaluate	22,629,600,000
▶ Integrator::evaluate	22,246,000,000
▶ std::_Vector_iterator<std::_Vector_val<std::_Simp	20,039,600,000
▶ std::vector<int,std::allocator<int> >::end	10,864,000,000
▶ std::vector<double,std::allocator<double> >::oper	10,290,000,000
▶ std::vector<double,std::allocator<double> >::oper	3,864,000,000
▶ std::vector<double,std::allocator<double> >::oper	3,054,800,000

Figure 6.9: The top 10 functions based on number of CPU cycles. Here the temperature is put to zero.

Looking at the hotspots reveals that Simulation::run is no longer present. This is because in this function the results of the Gaussian number generation are allocated to memory. When the temperature is zero this of course no longer happening.

CalculateEffectiveField

Most of the work is expected to be done in the Integrator class since this is where the bulk of the calculation happens and this reflects relatively well in the number of clock cycles.
At the top is Integrator::calculateEffectiveField. The main point of interest is here is the retirement rate being really high at over 88%. While this means that a lot of useful work is done it means optimization here might be gained by doing the work more efficiently. One of the main ways to do this is by using parallelization and it turns out that this function does mostly perform scalar operations as opposed to vector operations. Looking at the source code shows that the magnetic field and anisotropy use SIMD operations and are thus vectorized. The exchange interaction on the other hand is not vectorized. Thus the low fraction of vectorization indicates that the bottleneck lies in the exchange interaction. The exchange interaction can be vectorized but it has not been done as of yet since it is less trivial than the magnetic field and anisotropy.

Normalize

Next are two functions that are almost equal in time. Also they have switched position when the temperature was turned off. This is mostly likely due to there not being enough clock cycles with temperature to make an accurate prediction. Furthermore the relative position is not actually important since it is simply an indication of where a lot of time is spent.

First Simulation::normalize, here the main issue is division. Since normalization requires dividing each component by its magnitude it is very costly on division which takes a long time compared to multiplication for example. To remedy this the reciprocal magnitude is calculated and this is multiplied with each component. Reducing the number of divisions by three. However there is still one division per atom which is unavoidable.

Another issue here is that it is memory bound, from the L1 cache. This can be potentially be improved however. Currently the magnitude is calculated for each atom which means that it will need to access the x, y and z components. This is slow since they are not next to each other in cache because the memory layout is $\{x_1, x_2, ..., x_n\}, \{y_1, y_2, ..., y_n\}, \{z_1, z_2, ..., z_n\}$. It would therefore be better to calculate 4 atoms at the same time such that everything will be on a single cache line, thus reducing the pressure on the L1 cache. Something else that could be investigated is checking if it is possible to normalize less often, lets say every 10th iteration. This would reduce computation time at the cost of accuracy, the question is if this is accuracy that can be given up.

Integrate

Next is the the function where the actual integration happens, Integrator::integrate. This function is heavily core bound. Looking at the results show that memory is not an issue here and that it is perfectly vectorized. Therefore the issue is most likely in data dependency. It cannot perform calculations out of order since it needs to wait on previous calculations to finish. A way that could potentially be improved is by breaking the chain of calculations and splitting it up in several smaller calculations. This would allow for small parts that are not depended on each other to be calculated separately and combined later.

Evaluate

Finally there are the two Integrator::evaluate functions. It is not completely clear why there are two but the best guess is that since there are two calls to this function with different inputs they are treated separately. Since they show roughly the same characteristics they will be treated together. It should also be noted that if the number of clock cycles are added together they form the second biggest hotspot. In performance it is very similar to that of Integrator::integrate. It is also core bound while being completely vectorized and it also not memory bound. Thus the same possible improvement applies.

std::vector

Then at last there are the std::vector functions. The first two of these have to do with moving the pointer and are very efficient with over 95% retirement. Then there are the std::vector<>operator[] functions. These provide the data stored at the element given in the brackets. Some of these have a very high retirement rate around 90% will other will go as low as 25%. VTune classifies this core bound but it is unclear what causes this. A guess would be that the data is accessed in a different patterns and some cause more troubles than others. The weird thing here is that it is said to be core bound while it is data lookup, so it would be expected to be memory bound.

Chapter 7

A brief tour through the code

7.1 The model

7.1.1 Files

All of the files CLaSSiC produces follow the same naming scheme. The name of the produced data, for example energy, followed by the identifier of the simulation. So if it is the second run of the simulation it will produce energy1.dat.

CLaSSiC produces in total 3 files for every run. The first being data0.dat. This file starts with all of the parameters for that particular run and its followed by the orientations of the spins. They are in a long list which is structured as follows: $x_0, y_0, z_0, x_1, ...$ This file is written as a binary file to reduce file size and increase writing speed.

The next file that is produced is energy0.dat. This file is also a binary file for the same reasons and stores the value of the Hamiltonian for each iteration. This is useful for tracking how the total energy of the system changes.

Lastly there are the positions of the atoms on which the spins are located. This is simply a csv file that lists all of the atoms with the rows being the atom number and the columns the x, y and z coordinates respectively.

The only input is the run.bat file. This file is a batch file and makes it possible to automate command line arguments. This way all of the input parameters are organized neatly and makes it possible to loop over variables.

7.1.2 Structure generation

CLaSSiC generates structures from the unit vectors and basis atoms of the crystal. It will start by putting down all the atoms in the unit cell. Then for each atom it will put another atom 1 unit vector away until the desired number of unit cells is reached. Then if there is 2D component it will take all of the atoms generated so far and adds the second unit vector to each of atoms. The same of course goes for the third unit vector if it is specified. As a note the first unit vector should be along the x axis otherwise undefined behavior might happen.

The next thing is locating the nearest neighbours. This is a relatively easy task which is solved by checking if the distance between two positions is the less than the theoretical distance plus a small error. The small error accounts for any numerical issues.

Lastly there is the issue of applying periodic boundary conditions. A periodic boundary condition should be applied if the position of an atom is translated by a unit vector times the width and is then within nearest neighbour range.

7.1.3 Constants

The constants.cpp file has all the global constants of the code. It is important that they are also defined as extern in constants.hpp such that they can be used in the other files without creating multiple definitions of the same variable. These constants can be divided into three categories. First of all there are the physical constants. Then there are the user defined system parameters such as the step size or magnetic field strength. Lastly there are the derived parameters, these are a function of the other two. This is for example the number of atoms but also prefactors for equation that can be calculated ahead of time.

7.1.4 Integration

As mentioned in section 4.1 the integration is done by the midpoint method which is fairly simple and requires only a couple of steps. The first thing to do is to evaluate the differential equation. To do the evaluation the mean field needs to be calculated. If the evaluation is done the system can be integrated for a single time step.

7.2 Plotting

The CLaSSiC software also provides python scripts to create plots to analyze the results. Here a brief rundown will be given of the different plotting scripts. As a general concepts most files will have two variables called *fNum* and *atom*. *fNum* refers to the file that is being plotted and *atom* specifies a certain atom to analyze.

7.2.1 helper.py

This script deals with all of the automation. It stores all of the constants, and has functions to read the data files. Then there also functions for the physics behind the plotting such as the calculation for the scattering cross section and the theoretical dispersions. In short, anything that can be used by multiple scripts will be here.

7.2.2 Zeeman.py

Here three plots are produced to analyze a single spin. The first one shows the x, y and z spin components as a function of time. The second one is the real part of the Fourier transform and lastly it shows the time evolution of the spin in the x-y plane. An example of this plot is figure 5.1. Apart from the plot it will also calculate if the spin turns clock wise or anti clock wise.

7.2.3 Anisotropy.py

This is a very basic script. It calculates the initial angles from the data and calculates the Fourier transform to determine the energy. Then these are plotted against the theoretical value. This is shown in figure 5.2

7.2.4 Temperature.py

This file shows the average spin orientation as a function of the temperature. So to use this the user is expected to run multiple simulations with various temperatures with the initial condition 0 (see table B.2) where the spins are aligned with z-axis. It also possible to produce multiple curves for different magnetic fields.
The script takes the average over the last 10% of the data changing this value will change the fluctuations of the data unless the simulation has run over a very long time or with a very high relaxation constant. It also calculates the theoretical Langevin curve for reference. An example of this script is figure 5.3.

7.2.5 Rotor.py

To run rotor.py the user is expected to run several simulations with two spins opposing each other in the xy-plane with varying angles. Then the script will determine what these angles were and perform a Fourier transform to get their energy. An example can be seen in figure 5.4

7.2.6 Scatteringpath.py

This script produces the dispersion based on the scattering function $S^(\alpha\beta)(\mathbf{q},\omega)$. The first step is to create a path through reciprocal space. There are some predefined paths for different geometries that go through the high symmetry points. Then for each point on this path the scattering cross section is calculated. This produces an intensity curve on which rudimentary analysis is done to find the peaks. There are two options for this analysis. One is to simply selected the maximum of the curve and the other is the find peaks function provided by the SciPy library. The SciPy option is selected by setting the variable *usePeaks* equal to true. If this option is selected, then there is also the option to select the number of peaks that are plotted in the dispersion by setting *maxPeaks* to that value.

With these values set, two plots will be made. The top plot will show the energy for peaks at the k-point in reciprocal space and the bottom will show the intensity as a function of energy at a the selected point. This point is indicated by a red line in the top plot and can be set in the code by the variable called point. The second plot can be toggled on and off by setting *plotIntensities* to either true of false, respectively. If it is false the result will look like figure 5.6 and if it is true it will look like figure 5.8.

7.2.7 Allspins.py

Allspins.py will create a grid of plots and show the x, y (and z) components of the spins. Here the variable *end* can be changed to shorten or lengthen the time interval and *rows* and *columns* are used to change how many plots the script produces. This script is mostly for debugging purposes and checking if the spins move consistently over the whole time interval.

7.2.8 Animation.py

Like the name suggest this script creates an animation of the movement of the spins in the xy-plane. Some things that are interesting to change are the the time interval and the animation speed. The first is done by changing the value for *end*. The animation speed can be changed by changing two parameters in *animation.FuncAnimation()*. The first is to change the interval between two frames and the other one is by changing the last value in the *range()* function. Changing the frame interval will make the animation speed up or slow down while the second will skip frames if it is increased.

7.2.9 Energy.py

This is a plot which shows the time dependence of the energy. The energy calculated here is not the determined by a Fourier transform but is instead determined by evaluating the Hamiltonian. Like in the other files the time interval can be changed by altering the *end* variable. One thing that should be paid attention to is the y-axis. If the energy only changes slightly the axis will have very small values while if there is actually something physical happening the scale will be on the order of meV.

Chapter 8

Conclusion

In the end the target goal of 400x speedup has been blown out of the water. With the current model running almost 10.000 times faster.
Validation shows that it produces the theory accurately. There are some issues with the integration error and the dispersion anomalies for the antiferromagnetic spin chain. But both of them stem from numerical problems and are small enough to not effect results, especially if temperature is introduced.

The speed was mostly made possible by implementing the simulation in C++. The compiled code was very efficient when all of the optimization were enabled. Also the choice of integration method turned out to be a perfect balance between speed and accuracy. The benchmark section gave a detailed overview of the current bottleneck as well as suggestions on how to solve them. The biggest being the generating of Gaussian numbers for the temperature but solutions are relatively easy to implement.

Development was not without difficulties. The biggest of them was finding a sign error which came from different definitions between sources. Another notable setback was spherical coordinates not being feasible since at face value it seemed very promising.

One of the original goals of the thesis was to use the CLaSSiC model to look at the stability of kagomé lattices at nonzero temperatures. While the Q_0 and the $\sqrt{3} \times \sqrt{3}$ soft modes have been implemented the spin waves are a different story. To simulate these spin waves the initial state has to be known. The calculation of spin waves for kagomé lattices is highly non trivial and no papers which calculate the spin waves were found. Therefore due to time constraints this has not been included but a cursory investigation suggest that it could be calculated from equation 42 from the original spinW paper[20].

After the validation was finished CLaSSiC already went use by two groups of bachelor students. There are Thomas Hansen and Peter Beck who work on the frustrated antiferromagnet hexagonal yttrium manganite (h-$YMnO_3$) where they will compare the simulation results against experimental data from CAMEA. The other group consist of Frederik Philipsen and Simon Ørgaard. They investigate the impact of the circular structures on the magnetic properties of Ytterbium Gallium Garnet ($Yb_3Ga_50_12$) by comparing simulation results against experimental data.

There is still plenty of work left for the CLaSSiC model to be done. The most pressing of these would be to write proper documentation and make the installation process more user friendly. As of now this thesis is the only documentation that exists so getting started and using this simulation suite can be difficult. For someone who is used to writing code, compiling code and using version control it is all relatively straightforward. However this is not true for most people. So it would be very helpful to create an installer that could fetch updates once in a while as well as a decent Graphical User Interface (GUI) to make the experience more streamlined.

Apart from making the package more user friendly there are other additions that can be made to further expand the CLaSSiC suite. One of these would be to take into account all of the prefactors when calculating the scattering function 3.56. This would allow a better comparison between simulation and experiment.
Another option could be to allow for systems with different atoms in the geometry. This would give a much wider spectrum of possible geometries.
Lastly the current geometries are hard coded in. While it is not too difficult too add new ones a proper solution will need to be found if the number gets larger. Also it would be nice if new geometries could be added without changing the source code.

Bibliography

[1]J. Garde, "Numerical simulations of magnetic dynamics in nanoparticles", Masters's thesis (University of Copenhagen, Mar. 2011).

[2]J. P. Hyatt, "General purpose simulations of classical spin dynamics", Bachelor's thesis (University of Copenhagen, June 2019).

[3]E. B. Naver, "Simulations of anisotropy effects of classical spin dynamics in frustrated magnets at finite temperatures", Masters's thesis (University of Copenhagen, June 2021).

[4]S. Blundell, *Magnetism in condensed matter*, Oxford Master Series in Condensed Matter Physics (OUP Oxford, 2014).

[5]V. Berec, "Ion-atom quantum entanglement in a magnetic field based on the superfocusing effect - the spin qubit processing in silicon", PhD thesis (Oct. 2013).

[6]D. J. Griffiths, *Introduction to Quantum Mechanics (3rd Edition)*, 3rd (Cambridge university press, 2018).

[7]V. Sluka, T. Schneider, R. Gallardo, A. Kákay, M. Weigand, T. Warnatz, R. Mattheis, A. Roldán-Molina, P. Landeros, V. Tiberkevich, A. Slavin, A. Erbe, A. Deac, J. Lindner, J. Raabe, J. Fassbender, and S. Wintz, "Emission and propagation of 1d and 2d spin waves with nanoscale wavelengths in anisotropic spin textures", Nature Nanotechnology **14**, `10.1038/s41565-019-0383-4` (2019).

[8]W. F. Brown, "Thermal fluctuations of a single-domain particle", Phys. Rev. **130**, 1677–1686 (1963).

[9]A. Einstein, "Über die von der molekularkinetischen theorie der wärme geforderte bewegung von in ruhenden flüssigkeiten suspendierten teilchen", Annalen der Physik **322**, 549–560 (1905).

[10]T. L. Gilbert, "A phenomenological theory of damping in ferromagnetic materials", IEEE Transactions on Magnetics **40**, 3443–3449 (2004).

[11]E. Merzbacher, *Quantum mechanics* (Wiley, 1998).

[12]G. Squires, *Introduction to the theory of thermal neutron scattering* (Cambridge University Press, 1978).

[13]W. Marshall and S. W. Lovesey, *Theory of thermal neutron scattering* (Clarendon Press Oxford, 1971).

[14] *Numerical Integration - Midpoint, Trapezoid, Simpson's rule*, (July 2021) https://math.libretexts.org/@go/page/10269 (visited on 04/28/2022).

[15] *Illustration of midpoint method*, (May 2006) https://en.wikipedia.org/wiki/Midpoint_method#/media/File:Midpoint_method_illustration.png (visited on 04/28/2022).

[16] *Interpreted vs compiled programming languages: what's the difference?*, (Jan. 2020) https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/ (visited on 04/28/2022).

[17] *Classification of programming languages*, (2021) https://www.javatpoint.com/classification-of-programming-languages (visited on 04/28/2022).

[18] *Intel® vtune™ profiler*, (2022) https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.00q0uy (visited on 05/13/2022).

[19] *Intel® vtune™ profiler performance analysis cookbook*, (Mar. 2022) https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html (visited on 05/15/2022).

[20] S. Toth and B. Lake, "Linear spin wave theory for single-q incommensurate magnetic structures", Journal of Physics: Condensed Matter **27**, 166002 (2015).

Appendix A

Setup Tutorial

All of the code can be found on Github. This section will go over all the tools need to download, compile and run all of the code. This is all given from a windows point of view but the code also runs on Mac OS and Linux but the installation process will be different.

A.1 Github

The first thing to do is install git. Simply follow the download instructions from `https://git-scm.com/downloads` and install git. If you are not familiar with git you can simply use all of the default settings. Now with git installed you are ready to make a setup a remote branch to get the code. Follow the following steps:

1. Create a new folder named CLaSSiC2.0
2. Right-click and press: git bash here. This should open a terminal.
3. Type: ”git init” and press enter.
4. Type: ”git remote add origin https://github.com/timooo0/CLaSSiC2.0” and press enter.
5. Type: ”git pull origin master” and press enter

Now all of the code should be found inside the file CLaSSiC2.0 you just created. Whenever the code is update you can get the newest code by opening the terminal again and running the last command.

A.2 Compile

Now the folder contains all of the .cpp and .hpp files but there is no executable yet since you have to make that yourself. To do this you need a c++ compiler and I have used minGW which can be found here and run the installer with the default options

Then you need to add minGW to your path. You do this by following these steps, note however that theses steps are depended on the language of your computer.

1. type: "environmental variables" in the search bar and click: "Edit the systems environmental variables"
2. Click the button "Environment Variables..." in the bottom right.
3. Select the variable "Path" and click "Edit..."
4. Click "New" in the top right and type "C:\MinGW\bin" (note: if you install minGW somewhere this should be the path to the bin folder of minGW)
5. Click "OK" until all the opened windows are gone

To check if the installation went successful open a terminal by pressing the windows button, typing: "cmd" and press enter. In the terminal that opened type: "g++ –version". The output of this should show the version number of the compiler. Version 9.2.0 was used when compiling this project.

A.3 Python

For Python there are two ways of installing the necessary packages. The first is to install them directly with pip install and the second way is to install them in an environment. This is the preferred way to do it such that the versions and what packages are easier to keep track of and troubleshoot. To do this a package manager called anaconda is used, which can be downloaded from `https://www.anaconda.com/`. Then open anaconda prompt (anaconda3) and type the following commands.

1. Type "conda create –name CLaSSiC" and press enter, then press y to confirm when it prompts you to do so.
2. Type "conda activate CLaSSiC" and press enter.
3. Type "conda install numpy matplotlib scipy" and press enter, then press y to confirm when it prompts you to do so.
4. Type "conda install -c conda-forge pyfftw" and press enter, then press y to confirm when it prompts you to do so.

Appendix B

constants and settings

B.1 Validation settings

	Zeeman	Anisotropy	Temperature	Rotor	F chain	AF chain
step width	1e-15	1e-15	1e-14	1e-15	1e-15	1e-15
step number	1e6	1e6	1e7	1e6	1e6	1e16
Exchange constant	0	0	0	Varies	2	-2
Magnetic field	5	0	Varies	0	Varies	Varies
Axial anisotropy	0	10	0	0	Varies	Varies
Planar anisotropy	0	0	0	0	0	0
Relaxation constant	0	0	1e-3	0	1e-3	1e-5
Temperature	0	0	Varies	0	1	1e-6
Initial state	1	1	0	3	2	2
Mode	0	0	0	0	2	2
Angle	45	Varies	0	Varies	0	0
Structure	Single	Single	Single	Line	Line	Line
Unit cells in $\hat{x}$	1	1	1	2	30	30

B.2 Initial states

Number	Initial state
0	Aligned with z-axis
1	Angle with z-axis
2	Spin wave for chain
3	Rotor mode for 2 spins
4	Spin wave for square lattice
5	Single triangle
6	Kagome $\sqrt{3} \times \sqrt{3}$ soft mode
7	Kagome Q_0 soft mode

B.3 Constants

Symbol	Name	Units	Numerical value
γ	Gyromagnetic ratio	$s^{-1}T^{-1}$	$-1.760859644 \times 10^{11}$
μ_B	Bohr magneton	JT^{-1}	$9.274009994 \times 10^{-24}$
$\hbar$	reduced Planck constant	Js	$1.054571817 \times 10^{-34}$
g	g-factor	-	2.002
s	spin	-	$7/2$
k_b	Boltzmann constant	JK^{-1}	$1.38064852 \times 10^{-23}$

B.4 Simulation variables

Symbol	Name	Units	Typical value
dt	Time step size	s	1×10^{-15}
steps	Number of time steps	-	1×10^{6}
B	Magnetic field strength	T	10
J	Exchange constant	J	$\pm 2k_b$
anisotropyAxis	anisotropy strength	T	T
T	Temperature	K	1×10^{-1}
lambda	relaxation constant	-	1×10^{-3}

Appendix C

Spherical coordinates

$$\begin{aligned}
\frac{d\phi_i}{dt} =&\frac{\gamma}{\sin\theta_i\cos\phi_i}\left(\lambda B_x\cos\phi_i\sin\phi_i\cos^2\theta_i\right.\\
&-\lambda B_y\cos^2\phi_i\cos^2\theta_i\\
&-\lambda B_z\sin\theta_i\sin\phi_i\cos\theta_i\\
&-\lambda B_x\cos\phi_i\sin\phi_i\\
&+\lambda B_y\cos^2\phi_i\\
&+\lambda B_y\cos^2\theta_i\\
&-b\sin\theta_i\cos\phi_i\cos\theta_b\\
&+b\sin\theta_b\cos\phi_b\cos\theta_i\\
&-B_z\sin\theta_i\cos\phi_i\\
&\left.+B_x\cos\theta_i\right)\\
&+\gamma\cot\theta_i\tan\phi_i\left(-b\sin\theta_b\sin\phi_b\cos\phi_i\right.\\
&+B_x\lambda\cos\theta_i\cos\phi_i\\
&-B_y\cos\phi_i\cos\theta_i\sin\phi_i\\
&+b\sin\theta_b\cos\phi_b\sin\phi_i\\
&+B_y\lambda\cos\theta_i\sin\phi_i\\
&+B_x\sin\phi_i\\
&\left.-B_z\sin\theta_i\lambda\right)
\end{aligned}$$