Monte-Carlo Simulation Studies on the Superspin Structure of 3D Nanoparticle Supercrystals

Mauricio Cattaneo

Masterarbeit in Physik
vorgelegt der
Fakultät für Mathematik, Informatik und
Naturwissenschaften der RWTH Aachen
im Juli 2018
angefertigt am
Forschungszentrum Jülich GmbH, Peter Grünberg Institut
bei
Prof. Dr. David DiVincenzo
Prof. Dr. Thomas Brückel

Acknowledgements

I would like to give my sincere thanks to Prof. Dr. Thomas Brückel and PD Dr. Oleg Petracic for providing me the opportunity to work on this thesis in the institute JCNS-2/PGI-4 at Forschungszentrum Jülich GmbH.
I want to thank Prof. Dr. David DiVincenzo who agreed to be the first referee of my thesis. I would like to thank him for his interest in my research and his concise and valuable input.
I owe special thanks to Oleg Petracic, Michael Smik and Xiao Sun for being my supervisors. This thesis would not have been done without their encouragement, guidance and patient support from the beginning till the end.
I found great motivation from discussions with the entire staff at JCNS, special mention to the JCNS Nanoparticle Club. The exchange with excellent researchers and different scientific background than my own has been very inspiring and has shaped this thesis.
Finally, I must offer my heartfelt thanks to my family and my friends who have supported me during this project. I would like to thank them for their support, love and understanding.

Abstract

Nanoparticle (NP) supercrystals constitute a fascinating novel type of material with tunable magnetic, electronic and optical properties [3, 8, 12]. By choosing different NP materials, e.g. ferromagnetic or antiferromagnetic, a variety of magnetic and eventually multifunctional properties might be achieved. Hereby, one major challenge is the deliberate control of the supercrystal structure and of the resulting physical properties. In simulations we are able to model the collective magnetic ground states from microscopic assumptions [3–5]. This thesis aims to develop a proper simulation methodology to deal with lattices of interacting magnetic NP moments. Such a study is a crucial step towards predicting magnetic ground states and energy landscapes as function of the supercrystal lattice type and as function of the individual NP properties. Immediate goals include the study of the influence of dipole-dipole interactions on superparamagnetism and the spin structure of supercrystals at low temperatures.

Contents

1 Introduction **1**

2 Theory of Magnetism **2**

2.1 Origin of Magnetism in Solid State Matter 2

2.1.1 Bohr-van Leeuwen theorem 2

2.2 Magnetic Interactions 4

2.2.1 Dipole-Dipole Interaction 5

2.2.2 Exchange Interaction 5

2.2.3 Anisotropic Exchange or Dzyaloshinky-Moriya Interaction . . 6

2.3 Types of Magnetic Behavior 7

2.3.1 Diamagnetism 7

2.3.2 Paramagnetism 8

2.3.3 Collective Magnetism 8

2.3.3.1 Ferromagnetism 9

2.3.3.2 Ferrimagnetism 9

2.3.3.3 Antiferromagnetism 9

2.3.4 Geometrically Frustrated Systems and Spin Glasses 10

3 Methods **12**

3.1 Why Do Simulations in Physics 12

3.2 Categories of Monte-Carlo Simulations 12

3.2.1 Monte Carlo in Statistical Physics 13

3.2.2 Markov Chain methods 13

3.3 The Metropolis Algorithm 15

3.4 Quality of Pseudo-random Numbers Picked on a 2-Sphere 18

3.4.1 Uniformly Distributed Unit Vectors on a $(d-1)$-Sphere 20

3.4.2 Examples of Vector Distribution 21

3.4.3 MC simulations and Ergodicity 24

4 Ferromagnetic Nanoparticles and Supercrystals in the Non-interacting Limit **26**

4.1 Theory I 26

4.1.1 Stoner-Wohlfarth Model 26

4.1.2 Isotropic Paramagnet 27

4.1.3 Potential Landscape with Non-vanishing Anisotropy 29

4.1.4 ZFC-FC curves 31

4.1.5 ac-Susceptibility 32

4.2 Simulation results 34

4.2.1 Isotropic Superparamagnetism 34

4.2.2 ZFC-FC Curves . . . 35
4.2.2.1 High-Temperature Behavior . . . 35
4.2.3 Influences on the Blocking Temperature . . . 36
4.2.3.1 Particle Number . . . 36
4.2.3.2 Test Vector . . . 37
4.2.3.3 Number of Monte-Carlo Steps per Measurement . . . 38
4.2.3.4 Applied Magnetic Field . . . 39
4.2.4 Hysteresis Plots . . . 40
4.2.5 ac-Susceptibility and Cole-Cole Plot . . . 41
4.3 Summary and Discussion of Results . . . 43

5 Ferromagnetic Nanoparticles with Dipole-Dipole Interaction 44
5.1 Theory II . . . 44
5.1.1 Computational Aspects . . . 44
5.1.2 Mermin-Wagner Theorem . . . 44
5.1.3 Antiferromagnetic Part of the Dipole-Dipole Interaction . . . 45
5.1.4 Finite and Infinite Systems . . . 46
5.1.5 Possible Approximations . . . 47
5.1.6 Onsager Reaction Field Method . . . 47
5.1.6.1 Motivation and Model . . . 48
5.1.6.2 Solution of the Magnetostatic Problem . . . 49
5.1.6.3 Modelling parameters . . . 54
5.1.6.4 Cut-off Radius . . . 55
5.1.6.5 Determination of the relative permeability . . . 56
5.1.6.6 Consistency with non-interacting limit . . . 57
5.2 Simulation Results . . . 59
5.2.1 Groundstates in Periodic, Dipolar 3D Systems . . . 59
5.2.1.1 Results in the Limit of Vanishing Magnetocrystalline Anisotropy . . . 59
5.2.1.2 Series of 'Groundstates' Depending on Different Parameters in the Onsager Approximation . . . 61
5.2.1.3 Reconsidering Non-Vanishing Anisotropy . . . 68

6 Summary and Outlook 70

A Detailed Calculations 71
A.1 Magnetostatic Derivation of the Onsager Reaction Field . . . 71
A.1.1 Laplace Equation in Azimuthal Symmetry . . . 71
A.1.2 Solution to Our Boundary Value Problem . . . 73

B Programming **75**
B.1 C++ implementation of Metropolis algorithm 75
B.1.1 main.cpp 75
B.1.2 parameter.h 80
B.1.3 input-ini.h 81
B.1.4 str.h 85
B.1.5 perAux.h 96
B.1.6 rnd250.c 102
B.1.7 random-spd5.h 102
B.2 domainFinder.cpp 104

C References **113**

1 Introduction

NP supercrystals are regular arrangements of NPs in complete analogy to crystals in condensed matter. An example is given in figure 1.

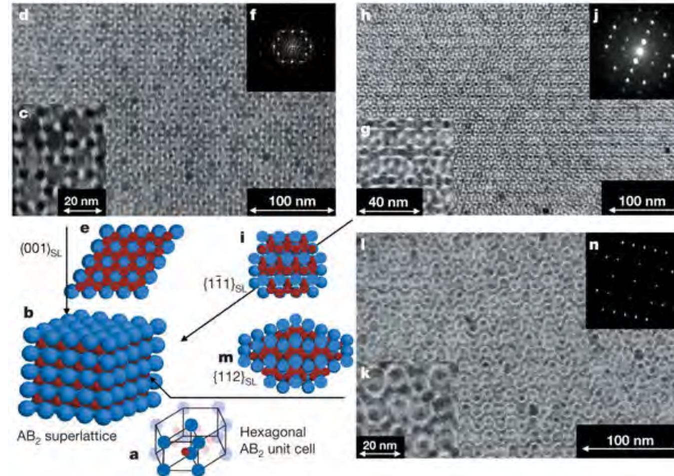


Figure 1: TEM micrographs and sketches of AB_2 superlattices (isostructural with intermetallic phase AlB_2, SG 191) of 11-nm γ-Fe_2O_3 and 6-nm PbSe NCs. [17]

The research on supercrystals started already several decades ago in the context of colloidal crystals on micron sized silica or polystyrene spheres [11]. In recent years, groups have succeeded in fabricating one, two or three dimensional arrangements of e.g. magnetic or semiconducting NPs using various self-organization techniques [3, 4, 8, 12]. The properties of these systems are determined both by the individual NPs and the interactions among them. Provided the composition and interaction of different NPs can be tuned, interesting behavior and novel applications may emerge [18].

But also from a fundamental point of view, such supercrystals of interacting magnetic nanoparticles are a fascinating subject. In particular NPs which consist of one magnetic single-domain per NP represent an interesting model system. The question whether systems with long-range interaction such as dipole-dipole interactions can exhibit long-range ordered ground states is not comprehensively answered in more than two dimensions. In fact, even the Mermin-Wagner theorem, answering the question for systems of up to two dimensions with a general ‘no’, has been challenged by e.g. Kosterlitz-Thouless transitions, via a divergence of the correlation length [9]. Albeit NP supercrystals have much larger length scales than usually considered in many-body theory, mesoscopic systems do provide very similar questions. The different length-scale and the dominance of the dipole-dipole interaction between the NP macromoments potentially leads to new possibilities of magnetic ordering within physical systems. In any case, the NP magnetic dipole-moments are subject to frustrated interactions.

2 Theory of Magnetism

Theoretical expectations of magnetic measurements will be the cornerstone of this chapter.

One can classify the magnetic phenomena into three main groups:

- Diamagnetism
- Paramagnetism
  - Localized Moments
  - Itinerant Moments
- Collective Magnetism
  - Ferromagnetism
  - Ferrimagnetism
  - Antiferromagnetism

2.1 Origin of Magnetism in Solid State Matter

2.1.1 Bohr-van Leeuwen theorem

Often, para- and diamagnetism are explained as an induction effect. It implies a theory of moving charges which can be treated within a classical atom model, and vector analysis as the significant mathematical language to understand magnetism. The general idea might be the following: The Larmor precession of the orbital angular momentum around the direction of the magnetic field induces an extra moment which according to Lenz's law is directed oppositely to the orientation of the applied field. Trying to rigorously calculate any atomic magnetic moment in this setting will inevitably lead to a contradiction as we will see now:
Let a solid consist of identical ions and let it possess translational symmetry. We can then write the magnetization as

$$\mathbf{M} = \frac{N}{V}\langle \mathbf{m} \rangle$$

where $\mathbf{m}$ is the magnetic moment of the individual ion. N is the number of ions in the volume V. If magnetism is a classical phenomenon, then each ion in our solid must offer a classical Hamiltonian function H and we have the following relations from statistical mechanics:

$$\mathbf{m} = -\nabla_{\mathbf{B}_0} H \tag{2.1}$$

$$\langle \mathbf{m} \rangle = \frac{1}{Z} \int d^{3N_e}x \int d^{3N_e}p \, \mathbf{m} \, e^{-\beta H} \tag{2.2}$$

$$Z = \frac{1}{N_e! h^{3N_e}} \int d^{3N_e}x \int d^{3N_e}p \, e^{-\beta H} \tag{2.3}$$

where

- Z is the classical partition function
- N_e is the number of electrons per ion
- $\beta \equiv (k_B T)^{-1}$ is the inverse temperature
- $\frac{1}{N_e! h^{3N_e}} \int d^{3N_e}x \int d^{3N_e}p \equiv \iint\limits_{\Gamma} d\gamma$ is the normalized integration over the complete phase space spanned by N_e electrons (3 dimensions for both position and momentum variables).

We arrive at

$$\begin{aligned}\langle \mathbf{m} \rangle &\overset{(2.2)}{=} \frac{1}{Z} \iint\limits_{\Gamma} d\gamma \, e^{-\beta H} \mathbf{m} \overset{(2.1)}{=} -\frac{1}{Z} \iint\limits_{\Gamma} d\gamma \, \underbrace{e^{-\beta H} \nabla_{\mathbf{B}_0} H}_{= -\frac{1}{\beta} \nabla_{\mathbf{B}_0} e^{-\beta H}} \\ &= \frac{1}{\beta Z} \nabla_{\mathbf{B}_0} \iint\limits_{\Gamma} d\gamma \, e^{-\beta H} \overset{(2.3)}{=} \frac{1}{\beta Z} \nabla_{\mathbf{B}_0} Z \end{aligned} \tag{2.4}$$

We therefore need to investigate the field dependence of the classical partition function. Noting that any magnetic field $\mathbf{B}_0$ can be written as $\mathbf{B}_0 = \nabla \times \mathbf{A}$, we rewrite our Hamilton function as

$$H = \frac{1}{2m} \sum_{\alpha=1}^{3} \sum_{i=1}^{N_e} ((\mathbf{p}_i)_\alpha + e\mathbf{A}_\alpha)^2 + H_{\text{int}}(\mathbf{r}_1, \cdots, \mathbf{r}_{N_e})$$

where H_{int} is the term representing the electron interactions. Thereby, we separated the position- and momentum dependent parts of the total Hamilton function which

lets the partition function take the form

$$Z = \frac{1}{N_e! h^{3N_e}} \int_{\Gamma_{\mathbf{r}}} d^{3N_e}x \, e^{-\beta H_{\mathrm{int}}(\mathbf{r}_1,\cdots,\mathbf{r}_{N_e})} \times$$
$$\times \int_{\Gamma_{\mathbf{p}}} d^{3N_e}p \, \exp\left(-\frac{\beta}{2m}\sum_{\alpha=1}^{3}\sum_{i=1}^{N_e}((\mathbf{p}_i)_\alpha + e\mathbf{A}_\alpha)^2\right)$$

Since the $\Gamma_{\mathbf{p}}$ integration lets every momentum coordinate run from $-\infty$ to ∞, the canonical momentum

$$\tilde{\mathbf{p}}_i := \mathbf{p}_i + e\mathbf{A}$$

can be transformed linearly with arbitrary but constant $\mathbf{A}$, without changing the integration limits. Since any applied magnetic field $\mathbf{B}_0$ would hence not alter the $\mathbf{p}$-integration, we have

$$\nabla_{\mathbf{A}} Z = \nabla_{\mathbf{B}_0} Z = 0 \tag{2.5}$$
$$\overset{(2.4)}{\Rightarrow} \langle \mathbf{m} \rangle = 0 \tag{2.6}$$

for any magnetic field.
We have therefore shown rigorously that, classically, there is no magnetism.
There cannot be any discussion whether magnetism does exit, therefore we have shown that it must be an effect only understandable quantum mechanically. This is the famous Bohr-van Leeuwen theorem:

Bohr-van Leeuwen Theorem

Magnetism is a quantum mechanical effect. Strictly classically, there cannot be either dia-, para- or collective magnetism

For the purposes of this thesis however, this does not mean that we will proceed to argue strictly quantum mechanically.
We can continue using classical or semiclassical models and calculations.

2.2 Magnetic Interactions

Different types of magnetic interactions are discussed in this section. We focus on interactions that are present in systems composed of magnetic moments of constant length.

2.2.1 Dipole-Dipole Interaction

The first interaction which might be expected to play a role is the magnetic dipolar interaction. Two magnetic dipoles $\mathbf{m}_1$ and $\mathbf{m}_2$ separated by $\mathbf{r}$ have an energy equal to

$$\begin{aligned} E_{\text{dip}} &= \frac{\mu_0}{4\pi r^3}\left[\mathbf{m}_1\cdot\mathbf{m}_2 - \frac{3}{r^2}(\mathbf{m}_1\cdot\mathbf{r})(\mathbf{m}_2\cdot\mathbf{r})\right] \\ &= \frac{\mu_0}{4\pi r^3}\left[\mathbf{m}_1\cdot\mathbf{m}_2 - 3(\mathbf{m}_1\cdot\hat{\mathbf{r}})(\mathbf{m}_2\cdot\hat{\mathbf{r}})\right] \\ \text{with} &\qquad \hat{\mathbf{r}} \equiv \frac{\mathbf{r}}{r} \end{aligned}$$

which therefore depends on their separation and their degree of mutual alignment. If the magnetic moments in question are single electrons and we take distances at atomic length scales, the energies were equivalent to roughly 1K in temperature. Therefore, properties of condensed matter are conventionally not overly dependent on the dipole-dipole interaction except for those ordering at mK temperatures. On the other hand, this is a long-range interaction where the complete sample needs to be taken into account. For example, it is conceivable that the specific length scale of this interaction influences the emergence of magnetic domains or other phenomena of magnetic ordering.

2.2.2 Exchange Interaction

In atomic crystals or any conventional magnetic system, exchange interactions are usually responsible for long-range magnetic order. They are purely quantum mechanical in nature, but the underlying principle is electrostatics.
Consider a simple model with just two electrons which have spatial coordinates $\mathbf{r}_1$ and $\mathbf{r}_2$ respectively. The wave function for the joint state can be written as a product of single electron states, so that if the first electron is in state $\psi_a(\mathbf{r}_1)$ and the second electron is in state $\psi_b(\mathbf{r}_2)$, then the joint wave function is in $\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2)$.

Since electrons are fermions, the true wave function must be antisymmetric with respect to particle exchange. So the *spin part* of the wave function must either be an antisymmetric singlet state χ_{S} ($S = 0$) in the case of a symmetric spatial state or a symmetric triplet state χ_{T} ($S = 1$) in the case of an antisymmetric spatial state. Therefore we can write the wave function for the singlet case Ψ_{S} and the triplet case Ψ_{T} as

$$\begin{aligned} \Psi_{\text{S}} &= \frac{1}{\sqrt{2}}\left[\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2) + \psi_a(\mathbf{r}_2)\psi_b(\mathbf{r}_1)\right]\chi_{\text{S}} \\ \Psi_{\text{T}} &= \frac{1}{\sqrt{2}}\left[\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2) - \psi_a(\mathbf{r}_2)\psi_b(\mathbf{r}_1)\right]\chi_{\text{T}} \end{aligned}$$

where both the spatial and spin parts of the wave function are included. If we assume the spin parts to be normalized, the energies of the two possible states are

$$\begin{aligned} E_{\mathrm{S}} &= \int \Psi_{\mathrm{S}}^{*} \mathcal{H} \Psi_{\mathrm{S}} \; d^3 r_1 d^3 r_2 \\ E_{\mathrm{T}} &= \int \Psi_{\mathrm{T}}^{*} \mathcal{H} \Psi_{\mathrm{T}} \; d^3 r_1 d^3 r_2 \\ \Rightarrow E_{\mathrm{S}} - E_{\mathrm{T}} &= 2 \int \psi_a^*(\mathbf{r}_1) \psi_b^*(\mathbf{r}_2) \mathcal{H} \psi_a(\mathbf{r}_2) \psi_b(\mathbf{r}_1) \end{aligned} \tag{2.7}$$

We can construct a new *effective* Hamiltonian by using

$$\mathbf{S}_1 \cdot \mathbf{S}_2 = \begin{cases} -\frac{3}{4} & S = 0 \text{ singlet} \\ \frac{1}{4} & S = 1 \text{ triplet} \end{cases}$$

The new Hamiltonian becomes

$$\mathcal{H}' = \frac{1}{4}(E_{\mathrm{S}} + 3E_{\mathrm{T}}) - (E_{\mathrm{S}} - E_{\mathrm{T}}) \; \mathbf{S}_1 \cdot \mathbf{S}_2$$

The interesting part is the non-constant spin dependent term. Defining the *exchange constant* J by

$$J := \frac{E_{\mathrm{S}} - E_{\mathrm{T}}}{2} \overset{(2.7)}{=} \int \psi_a^*(\mathbf{r}_1) \psi_b^*(\mathbf{r}_2) \mathcal{H} \psi_a(\mathbf{r}_2) \psi_b(\mathbf{r}_1)$$

we define a new effective Hamiltonian

$$\mathcal{H}^{\text{spin}} = -2J \, \mathbf{S}_1 \cdot \mathbf{S}_2 \tag{2.8}$$

If $J > 0$, $E_{\mathrm{S}} > E_{\mathrm{T}}$ and the triplet state $S = 1$ is favoured. If $J < 0$, $E_{\mathrm{T}} > E_{\mathrm{S}}$ and the singlet state $S = 0$ is favoured.

The above derivation only holds for exactly 2 electrons and the generalization to many-body systems is not trivial. Nevertheless, the Hamiltonian (2.8) motivates the Heisenberg model

$$\mathcal{H} = -\sum_{ij} J_{ij} \mathbf{S}_i \cdot \mathbf{S}_j \tag{2.9}$$

with the factor 2 omitted to prevent counting pairs twice.

2.2.3 Anisotropic Exchange or Dzyaloshinky-Moriya Interaction

This interaction originates from spin-orbit interactions within one magnetic ion. It can be understood as the exchange interaction between the excited state of one ion

and the ground state of the other.

$$\mathcal{H}_{\mathrm{DM}} = \mathbf{D} \cdot (\mathbf{S}_1 \times \mathbf{S}_2) \tag{2.10}$$

The vector $\mathbf{D}$ vanishes when the crystal field has an inversion symmetry with respect to the centre between the two magnetic ions. However, in general $\mathbf{D}$ may not vanish and then will lie parallel or perpendicular to the line connecting the two spins, depending on the symmetry. The form of the interaction is such that it tries to force $\mathbf{S}_1$ and $\mathbf{S}_2$ to be at right angles in a plane perpendicular to the vector $\mathbf{D}$ in such an orientation as to ensure that the energy is negative. Its effect is therefore very often to cant the spins by a small angle. It commonly occurs in antiferromagnetics and results in a small ferromagnetic component of the moments which is produced perpendicular to the spin axis of the antiferromagnet. The effect is known as *weak ferromagnetism*. It is found in, for example, α-Fe_2O_3.

2.3 Types of Magnetic Behavior

We discuss magnetic behavior via the classification of magnetic materials. Specifically, we utilize the characteristic dependence of the magnetic susceptibility χ on temperature, applied magnetic field and history:

2.3.1 Diamagnetism

Diamagnetism is defined by

$$\chi^{\mathrm{dia}} < 0 \qquad \chi^{\mathrm{dia}} = \mathrm{const.}$$

The classical picture of diamagnetism being an induction effect has already been discussed in 2.1.1. Without discussing proper quantum mechanical treatments like Landau-Diamagnetism in crystals, we focus on phenomenology:
Diamagnetism is a property displayed by *all* materials. However, we only speak of diamagnetism if no other form of magnetism (para- or collective magnetism) is present since it will in general be the weaker effect.
Examples include

- Organic molecules
- Few metals like bismut, zinc, mercury
- Nonmetals like sulfur, iodine, silicon
- *Meissner-Ochsenfeld* effect: Superconductors at $T < T_c$ are perfect diamagnets: $\chi^{\mathrm{dia}} = -1$

As noted, in the presence of e.g. permanent magnetic dipoles, diamagnetism will be a negligible effect and it will not feature throughout this thesis.

2.3.2 Paramagnetism

Typically, one has

$$\chi^{\text{para}} > 0 \quad \chi^{\text{para}} = \chi^{\text{para}}(T)$$

Essentially, paramagnetism is connected to the existence of permanent magnetic dipoles which try to, more or less, orient themselves along an applied auxiliary field $\mathbf{H}$. This competes with thermal motion, hence the temperature dependence of χ^{para}.

In materials, there are two possible origins of such permanent dipoles:

- Localized moments $\rightarrow$ *Langevin paramagnetism*
- Itinerant moments ($\sim$ quasi-free conduction electrons) $\rightarrow$ *Pauli paramagnetism*

Generally, one has

$$\chi^{\text{Langevin}} \gg \chi^{\text{Pauli}} \approx \text{const.}$$

and χ^{Pauli} temperature independent to a first order approximation.
Therefore, only Langevin paramagnetism will feature in this thesis, this however extensively.

2.3.3 Collective Magnetism

The susceptibility in this case is in general a complicated function of the applied field, temperature and the magnetic history of the sample:

$$\chi^{\text{coll.}} = \chi^{\text{coll.}}(T, H, \text{history})$$

Collective magnetism arises due to interactions between the permanent magnetic dipoles. Again, these permanent magnetic dipoles can be either *localized* or *itinerant*
In the context of more standard solid state physics, the interaction in question is the quantum mechanical *exchange interaction* which does not have a classical analogy. As we will discuss later, this view is not sufficient in case of supercrystals comprised of magnetic nanoparticles. Instead, the origin for collective magnetism in our systems, will be the magnetic dipole-dipole interaction between the permanent magnetic moments of the nanoparticles.
Nevertheless, we recall other characteristic features of standard systems with collective magnetism: The exchange interaction leads to a critical temperature T^* below which there exists a *spontaneous magnetization*, i.e. a spontaneous ordering

of dipoles that is not forced externally.
Collective magnetism due to exchange interaction is conventionally divided into three sub-classes:

- Ferromagnetism
- Ferrimagnetism
- Antiferromagnetism

Above the critical temperature T^*, collective magnetic transitions into paramagnetism with the characteristic temperature dependence of the inverse susceptibility sketched in figure 2

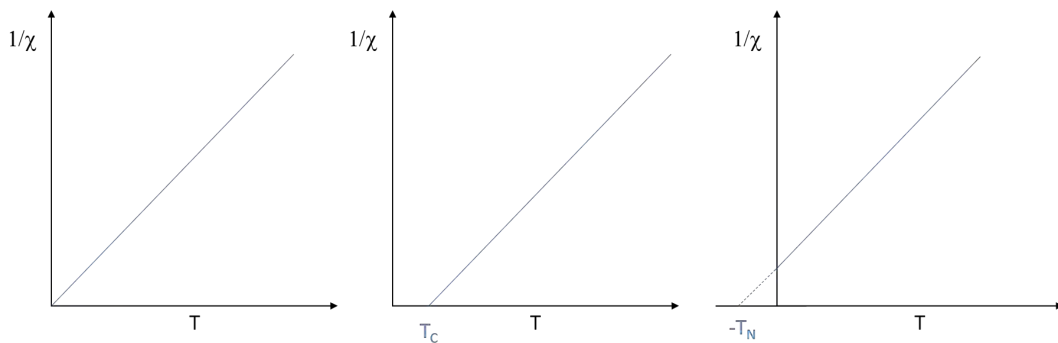


Figure 2: Temperature dependence of the inverse magnetic susceptibility according to the Curie-Weiss law. *Left:* Paramagnetism, *middle:* Ferromagnetism, *right:* Antiferromagnetism

2.3.3.1 Ferromagnetism

In this case the critical temperature is called the *Curie temperature*:

$$T^* = T_C$$

For temperatures $0 < T < T_C$ the permanent moments have a preferential orientation. At $T = 0$, all moments are oriented parallel to each other although $\mu_0 \mathbf{H}_{\text{ext}} = 0$.

2.3.3.2 Ferrimagnetism

In this case the lattice of the system is divided into at least two sublattices A, B with different absolute values of sublattice magnetizations $\mathbf{M}_A$, $\mathbf{M}_B$ such that

$$\mathbf{M}_A \neq \mathbf{M}_B \qquad \text{and} \qquad \mathbf{M}_A + \mathbf{M}_B \neq 0 \quad \text{for } T < T_C$$

2.3.3.3 Antiferromagnetism

Here, the criticial temperature is called *Néel temperature*:

$$T^* = T_N$$

It is a special case of ferrimagnetism:

$$|\mathbf{M}_A| = |\mathbf{M}_B| \neq 0 \qquad \text{and} \qquad \mathbf{M}_A + \mathbf{M}_B = 0 \quad \text{for} \;\; T < T_C$$

The total magnetization $\mathbf{M} = \mathbf{M}_A + \mathbf{M}_B$ is therefore always zero in the absence of externally applied fields.

2.3.4 Geometrically Frustrated Systems and Spin Glasses

In realistic systems, more than one interaction or energy contribution determines the total energy of the system. The subsequent competition between interactions can lead to complex magnetic behavior that is not trivially following from the constituent interactions.
Even before considering the competition between independent interactions, 2-particle interactions in realistic (super)crystal structures will induce geometric frustration: In many lattices it is not possible to satisfy all the interactions in the system to find *the* ground state. Often this leads to the absence of a single unique ground state but a variety of low energy states.
We refer to this property that the system has no good way to choose which low energy configuration it must adopt as *frustration*. It is important to note that this phenomenon emerges with pretty much any magnetic interaction, including nearest-neighbour interactions such as the antiferromagnetic interaction:

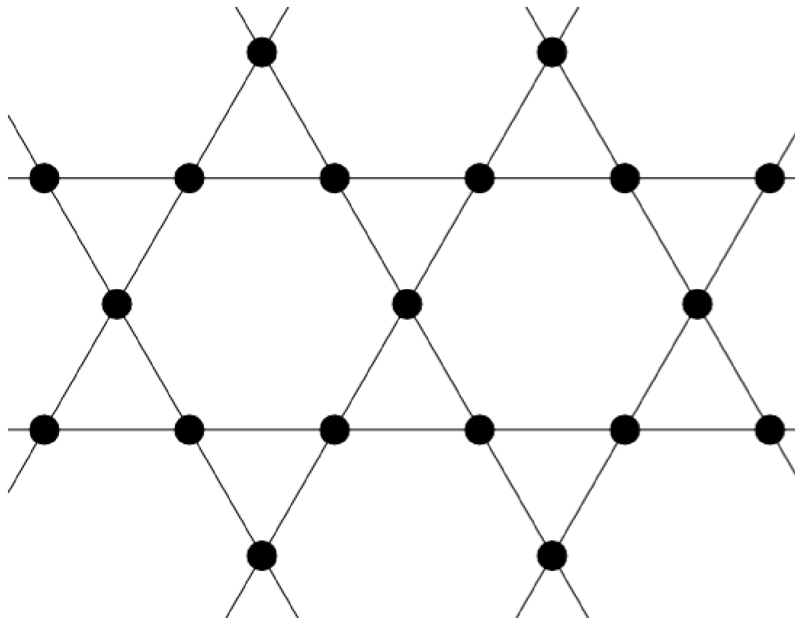


Figure 3: Kagome lattice as an example for a 2D geometry which leads to a frustrated system.

On a square lattice it is easily possible to satisfy the requirement that nearest-neighbour spins must be antiparallel. However on a triangular lattice, things differ: If two adjacent spins are placed antiparallel, the third spin has no good choice

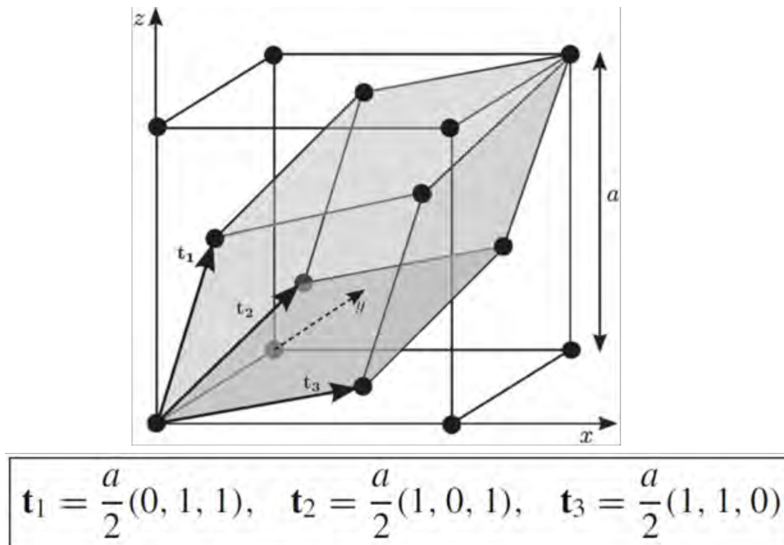


Figure 4: The face-centered-cubic (*fcc*) crystal structure also accounts for geometric frustration and will be heavily featured in future chapters. [7]

between aligning with one or the other spin. Locally, there exists no unique lowest-energy state, but only two low-energy states that are equally unsatisfied.
This minimal frustrated system already shows metastability, hysteresis effects and time-dependent relaxation towards equilibrium, all of which are phenomena absent from the square lattice.
This example relies heavily on the fact that we considered models with low dimensionality, i.e. $dim < 3$. In three dimensions, frustration would not emerge from a triangular of e.g. *Kagome* lattice, but from a pyrochlore structure in which the magnetic ions occupy a lattice of corner sharing tetrahedra. Here, there is no spin order observable for any temperature, only a classical groundstate with macroscopic degeneracy, sometimes described as *cooperative paramagnetism.*

We can define a spin glass as a random, magnetic system with mixed interactions characterized by a random yet cooperative freezing of spins at a well defined freezing temperature T_f below which a metastable frozen state appears without the usual magnetic long-range ordering.
The decisive term in this definition is *random.* Different types of randomness can be imagined to transform a non-spin glass into a spin glass:

- Site-randomness
- Bond-randomness

Equally important is the presence of competing interactions as previously described. Contributing features include magnetic anisotropy e.g. in amorphous magnets where a random distribution of the *easy-axes* implies random anisotropy.

3 Methods

3.1 Why Do Simulations in Physics

In many cases, models of ideal systems can be explored by theoretical methods, but they do not offer any physical realization so that no comparison to experiment is available. In many other cases, experimental realizations are too complex to be modelled by theoretical methods. In this situation the only possible test for an approximate theoretical solution is to compare with 'data' generated from a computer simulation.
Nuclear reactor meltdowns are a dramatic example: Although we want to know what the results of such events would be, we do not want to carry out experiments. There are also real physical systems which are sufficiently complex that they are not presently amenable to theoretical treatment. An example is the problem of understanding the specific behaviour of a system with many competing interactions and which is undergoing a phase transition. A model Hamiltonian/Hamilton function which is believed to contain all the essential features of the physics may be proposed, and its properties may then be determined from simulations. If the simulation disagrees with experiment, then a new Hamiltonian must be found. An important advantage of simulations is that different physical effects which are simultaneously present in real systems may be isolated and, through separate consideration by simulation, may provide a much better understanding.
The Monte Carlo method has had a considerable history in physics. As far back as 1949 a review of the use of Monte-Carlo simulations using 'modern computing machines' was presented by Metropolis and Ulam [15]. In addition to giving examples they also emphasized the advantages of the method. Of course, in the following decades the kinds of problems they discussed could be treated with far greater sophistication that was possible in the first half of the twentieth century. Nowadays, Monte-Carlo simulation methods have spread into different disciplines that have barely any connection to physics.

3.2 Categories of Monte-Carlo Simulations

A brief overview about Monte-Carlo methods is given. I present additional details about Markov-Chain methods because the research which is presented in this thesis was exclusively done via the Metropolis-Algorithm, the most famous representative of Markov-Chain Monte-Carlo methods.

3.2.1 Monte Carlo in Statistical Physics

Monte-Carlo methods are used throughout many physical and non-physical science branches. In physics, especially statistical mechanics, the following branches are of special interest:

- Monte-Carlo integration
- Importance sampling techniques, specifically Markov Chain methods
  - Local algorithms
  - Non-local algorithms

3.2.2 Markov Chain methods

The concept of Markov chains is central to those Monte-Carlo methods that are the most prominent in physics, especially solid state physics [10].

We define a stochastic process at discrete times labeled consecutively $t_1, t_2, t_3 \dots$ for a system with a finite set of possible states $S_1, S_2, S_3, \dots$, and we denote by X_t the state the system is in a time t. We consider the conditional probability that $X_{t_n} = S_{i_n}$,

$$P\left(X_{t_n} = S_{i_n} | X_{t_{n-1}} = S_{i_{n-1}}, X_{t_{n-1}} = S_{i_{n-2}}, \dots X_{t_2} = S_{i_1}\right)$$

given that at the preceding time the system state $X_{t_{n-1}}$ was in state $S_{i_{n-1}}$, etc. Such a process is called a Markov process if this conditional probability is in fact independent of all states but the immediate predecessor, i.e.

$$P\left(X_{t_n} = S_{i_n} | X_{t_{n-1}} = S_{i_{n-1}}\right)$$

The corresponding sequence of states $\{X_t\}$ is called a Markov chain, and the above conditional probability can be interpreted as the transition probability to move from the state i to state j,

$$W_{ij} = W(S_i \to S_j) = P(X_{t_n} = S_j | X_{t_{n-1}} = S_i)$$

We further require that

$$W_{ij} \geq 0 \qquad \sum_j W_{ij} = 1$$

as usual for transition probabilities. We may then construct the total probability $P(X_{t_n} = S_j)$ that at time t_n the system is in state S_j as

$$\begin{aligned} P(X_{t_n} = S_j) &= P(X_{t_n} = S_j | X_{t_{n-1}} = S_i) \cdot P(X_{t_{n-1}} = S_i) \\ &= W_{ij} P(X_{t_{n-1}} = S_i) \end{aligned}$$

The master equation conserves the change of this probability with time t (treating time as a continuous rather than discrete variable and writing then $P(X_{t_n} = S_j) =: P(S_j, t)$)

$$\frac{dP(S_j,t)}{dt} = -\sum_i W_{ji} P(S_j,t) + \sum_i WijP(S_i,t) \tag{3.1}$$

Equation (3.1) can be considered as a 'continuity equation': The total probability is conserved at all times because

$$\sum_j P(S_j,t) \equiv 1 \qquad \forall t \in \mathbb{R}$$

Furthermore, all probability of a state i that is 'lost' by transition to state j is gained in the probability of that state, and vice versa.
The Master equation therefore describes the balance of gain and loss processes:

The processes

$$\begin{aligned} S_j &\to S_{i_1} \\ S_j &\to S_{i_2} \\ S_j &\to S_{i_3} \\ &\cdots \end{aligned}$$

are mutually exclusive. Hence the total probability for a move away from the state j is simply the sum $\sum_i W_{ij} P(S_j, t)$.
We stress that equation (3.1) brings out the basic property of Markov processes:

Basic Property of Markov Processes

The knowledge of the state at time t completely determines the future time evolution.

The main significance is that the importance sampling Monte Carlo process that will feature throughout this thesis via the Metropolis algorithm, can be interpreted as a Markov process if the following is true about the transition probabilities W_{ij}: From now on we require that the transition probabilities satisfy the principle of detailed balance with the equilibrium probability $P_{\text{eq}}(S_j)$:

$$W_{ji} P_{\text{eq}}(S_j) = W_{ij} P_{\text{eq}}(S_i) \tag{3.2}$$

which will be fundamental for the Metropolis algorithm presented in section 3.3.
We already note that (3.2) implies

$$\frac{dP_{\text{eq}}(S_j,t)}{dt} \equiv 0 \qquad \forall t \in \mathbb{R}$$

when put into (3.1) because all gain and loss terms cancel exactly. This is elementary for what one would understand by the term 'equilibrium' in context of a system transitioning between states [10].

3.3 The Metropolis Algorithm

From now on, we are only concerned with Monte-Carlo techniques as applied in statistical physics, specifically on-lattice models of systems offering permanent magnetic dipoles that display collective magnetism due to interactions.
In order to illustrate our discussion, we consider the Ising model.
The simple Ising model in zero applied field consists of spins which are confined to the sites of a lattice and which may have only the values $+1$ or -1. These spins interact with their nearest neighbors on the lattice with interaction constant J; the Hamiltonian for this model is given by

$$H = -J \sum_{i,j} \sigma_i \sigma_j \qquad \sigma_i = \pm 1$$

The Ising model has been solved exactly in one and two dimensions so that Monte-Carlo results in these cases can be directly compared to theoretical expectations.
Next I present the classic Metropolis method.
Configurations are generated from a previous state using a transition probability which depends on the energy difference between the initial and final states. The sequence of states produced follows a time-ordered path, but the time in this case is referred to as 'Monte Carlo time'. For relaxation models, such as we will assume are viable models for magnetization curves of nanoparticle supercrystals, the time-dependent behavior is described by a master equation like (3.1):

$$\frac{\partial P_n(t)}{\partial t} = -\sum_{n \neq m} \left[P_n(t) W_{n \to m} - P_m(t) W_{m \to n} \right] \tag{3.3}$$

where $P_n(t)$ is the probability of the system being in state n at time t, and $W_{n\to m}$ is the transition rate for the process $n \to m$. We again identify the *detailed balance* from (3.2)

$$P_n(t) W_{n \to m} = P_m(t) W_{m \to n}$$

as a simple constraint that guarantees an equilibrium being realized as

$$\frac{\partial P_n(t)}{\partial t} \overset{\text{equil.}}{=} 0$$

The probability of the nth state occurring in a classical system is given by

$$P_n(t) = \frac{1}{Z} \exp\left(-\frac{E_n}{k_B T} \right) \tag{3.4}$$

where Z is the partition function. Outside of very simple cases like the Ising model discussed here, this expression is very difficult to evaluate, mostly because the partition function, i.e. knowledge about *every possible state and its energy*, is rarely ever known. However, one can avoid this difficulty by generating a Markov chain of states, i.e. generate each new state directly from the preceding state. If we produce the nth state from the mth state, the relative probability is the ratio of the individual probabilities and the denominator, the largely unknown Z cancels. As a result, only the energy difference between the two states is needed, e.g.

$$\Delta E = E_n - E_m$$

The previous idea is possibly the most significant reason why Markov-Chain methods have been so successful in statistical physics because one can circumvent the arduous and ultimately not as interesting task of evaluating the partition function of a large, interacting system.
For the transition rates, any choice that satisfies detailed balance (3.2) is acceptable[1]. The first choice of rate which was used in statistical physics is the Metropolis form [15]

$$W_{m \to n} = \begin{cases} \tau_0^{-1} \exp(-\Delta E / k_B T) & \Delta E > 0 \\ \tau_0^{-1} & \Delta E < 0 \end{cases} \tag{3.5}$$

where τ_0 is the time required to attempt a spin-flip. The way the Metropolis algorithm is implemented can be described by a simple recipe, illustrated in figure 5

Metropolis importance sampling Monte-Carlo scheme (3.6)

1. Choose an initial state
2. Choose a site i
3. Calculate the energy change ΔE which results if the spin at site i is flipped
4. Generate a random number r such that $0 < r < 1$
5. If $r < \exp(-\Delta E / k_B T)$, flip the spin
6. Go to the next site and go to 3.

[1] From the derivation from the Master equation (3.1), detailed balance is sufficient but not necessary. It turns out however, that in practice only transition rates that do satisfy detailed balance are regularly used.

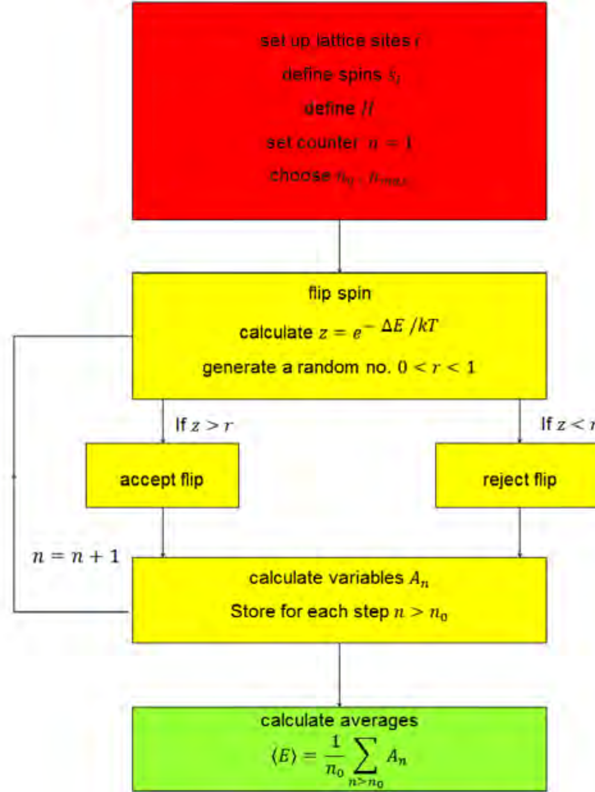


Figure 5: Metropolis Algorithm recipe. The red part is done once in the beginning of a simulation. the yellow part is done in loops, the number of which linearly determines the computation effort and time. The green part is done once after the yellow part has been iterated sufficiently, meaning that one assumes an equilibrium has been reached.

After a certain number of spins have been considered, the properties of the system are determined and added to the statistical average which is being kept. Note that the random number r must be chosen *uniformly* in the interval $[0, 1]$, and successive random numbers should be uncorrelated.

Obviously, this algorithm can be easily modified for use on different on-site models where the total energy of any given configuration can be calculated. Both the required high quality of random numbers and the precise nature of *'flipping'* in the context of the more involved model we consider in the actual research part of this thesis will be thoroughly discussed in section 3.4.

'Monte-Carlo time' is usually measured in terms of Monte Carlo steps per site (MCS/site) which corresponds to the consideration of every spin in the system once. With the algorithm from figure 5 states are generated with a probability proportional to (3.4) once the number of states is sufficiently large such that any initial transients from the early stages of the iterative loop are negligible. Then the desired averages

$$\langle A \rangle = \sum_n P_n A_n$$

of variables or observables A simply become arithmetic averages over the entire sam-

ple of states which is kept. Note that if an attempted spin-flip is rejected, the old state is counted again for the averaging.

We reiterate that the hallmark of the *Metropolis algorithm* is the specific choice of transition rates given in (3.5). Additional transitions can be imagined like Parallel Tempering which are used in order to accelerate convergence speeds of equilibrium averages for observables or the spin-configuration itself. Also, the choice how to flip spins or, more generally, choose a new configuration for consideration during a subsequent MCS, can be heavily altered compared to figure 5 where one just picks a site randomly.

Both types of modifications lead to algorithms that are not strictly *Metropolis* algorithms, but are still Markov-chain methods if they offer constraints ensuring possible equilibrium, most frequently via enforcing detailed balance [10].

3.4 Quality of Pseudo-random Numbers Picked on a 2-Sphere

The Metropolis algorithm as described in section 3.3 has been used extensively for Ising models where spins are restricted to exactly two states, *up* and *down*. In particular, this implies that it is clear how a local update to achieve a new configuration has to be carried out: Flipping one site to the other state.

In a Heisenberg-like model where each site carries a (super)spin that can assume every position on the 2 dimensional surface of a 3D unit sphere, it is far less clear how an update should be performed. Broadly speaking, 2 types of local updates in a Heisenberg-like on-lattice spin model are possible:

- The possible new states are uniformly distributed on the entire sphere without any bias from position of the original state.
- There is a probabilistic bias which part of the sphere is reachable within one Monte Carlo update.

In this thesis, we will exclusively utilize the second type of spin update. The implementation is depicted in figure 6:

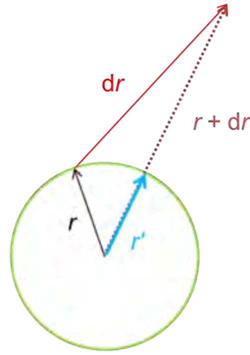


Figure 6: How spins are transformed or 'flipped' during one Monte Carlo step. A random vector $d\mathbf{r}$ (red) is obtained via the Marsaglia method and added to the original $\mathbf{r}$ (black). The result is normalized, yielding $\mathbf{r}'$ (blue). In the end, one has both $\mathbf{r}$ and $\mathbf{r}'$ within a spherical shell with radius 1 (green).

Incremental Transformation

1. Generate an *unbiased*, random vector $d\tilde{\mathbf{r}}$ that is uniformly distributed on a 3D unit sphere and stretch it if necessary with a scalar d_m such that $d\mathbf{r} = d_m\, d\tilde{\mathbf{r}}$
2. Add this random shift vector to the original $\mathbf{r}$
3. Normalize the result so that $\mathbf{r}'$ is a unit vector.

The generation of a uniform distribution of random vectors on a spherical shell is discussed in section 3.4.1, we can assume for now that we have access to such random vectors of sufficient quality.
This construction ensures that as long as d_m is not too large, $\mathbf{r}'$ is biased towards not deviating too much from $\mathbf{r}$. For example:

$$d_m \lnapprox 2 \quad \Rightarrow \quad \mathbf{r}' \neq -\mathbf{r} \quad \forall\, d\mathbf{r}$$

which means that not every orientation is obtainable within one MCS. The d_m dependence of the $\mathbf{r}'$ distribution is discussed in 3.4.2. In most cases[2], we choose $d_m = 1$ which means that within 1 MCS there is a heavy bias for new orientations towards the original position. Figuratively, if the original position $\mathbf{r}$ represents the north pole, then $\mathbf{r}'$ is mathematically confined in the northern hemisphere while the equator is only asymptotically obtainable.

[2]Justification and exemptions are given when needed

3.4.1 Uniformly Distributed Unit Vectors on a $(d-1)$-Sphere

For arbitrary dimensions $d \geq 1$ one can generate uniformly distributed vectors on the surface of the corresponding $(d-1)$-sphere:

- for $d = 1$: set $\{-1, 1\}$ as boundary of the interval $[-1, 1]$
- for $d = 2$: full circle (as boundary, not area) with radius 1 and origin $(0, 0)$ in cartesian coordinates
- for $d = 3$: spherical shell of the conventional 3-sphere with radius 1 and origin $(0, 0, 0)$ in cartesian coordinates

We can always do this with d random variables which have a Gaussian distribution in an arbitrary interval:

$$\text{Gaussian} \quad (x_1, x_2, \ldots, x_d) \quad \mapsto \quad \frac{1}{\sqrt{x_1^2 + x_2^2 + \cdots + x_d^2}} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix}$$

This d-vector will be uniformly distributed on the surface of a d-sphere.

For the special case $d = 3$ that we are interested in however, there is a more elegant method by Marsaglia [13] which requires only 2 instead of 3 random numbers:

Marsaglia Method

1. Pick a and b from independent uniform distributions on $(-1, 1)$
2. Reject points for which $a^2 + b^2 \geq 1$
3. From the remaining points
   - $x = 2a\sqrt{1 - a^2 - b^2}$
   - $y = 2b\sqrt{1 - a^2 - b^2}$
   - $z = 1 - 2(a^2 + b^2)$

The vectors $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ then have a uniform distribution on the surface of a unit sphere as needed in our spin-update scheme.

3.4.2 Examples of Vector Distribution

Next we demonstrate the influence of the length of the test vector d_m onto the vector distribution of a modified vector after a certain number of steps according to our recipe in figure 6.
The following graphs correspond to this scenario:

1. 10^6 unit vectors are initialized identically with orientation $(1, 0, 0)$
2. Choose $d_m = 2$.
3. With independent Marsaglia random vectors, perform the transformation.
   - once → graphs in figures 7, 8, 9.
   - 1000 times → graphs in figures 10, 11, 12.
4. Record the component distribution of the resulting vectors after all transformations are done.
5. Fill histograms with bins of width 0.002.

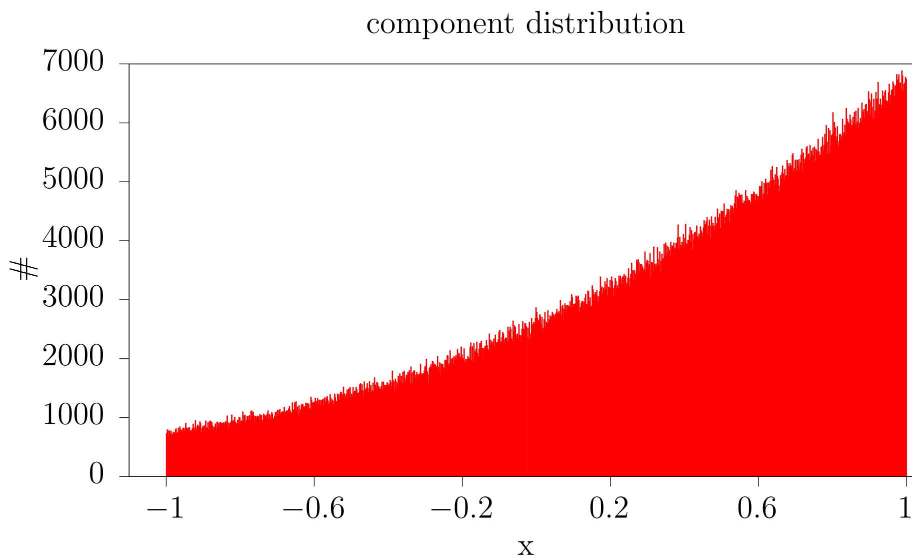


Figure 7: Distribution of x-components after 1 loop

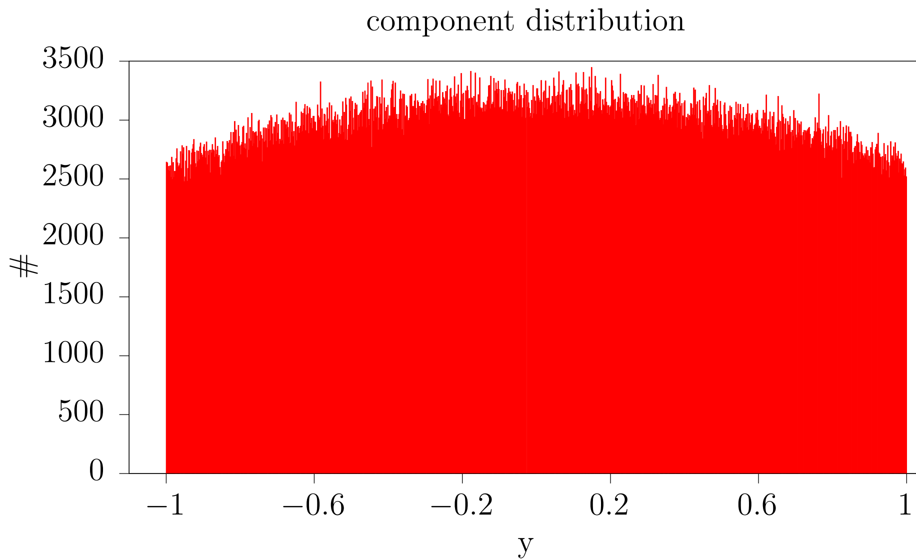


Figure 8: Distribution of y-components after 1 loop

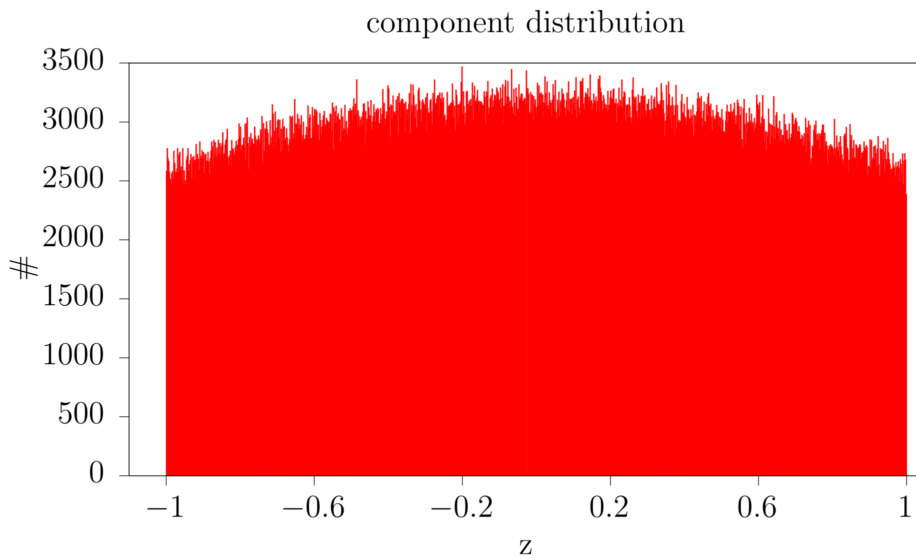


Figure 9: Distribution of z-components after 1 loop

We observe that the x-component covers the complete interval $(-1, 1)$, but the distribution is skewed in favor of the original orientation $x \to 1$. The y- and z-components follow identical distributions, centered around 0. Note that the integrated y- and z-components are less than the x-component because one 'flip' cannot undo the non-uniform start distribution around $(1, 0, 0)$.

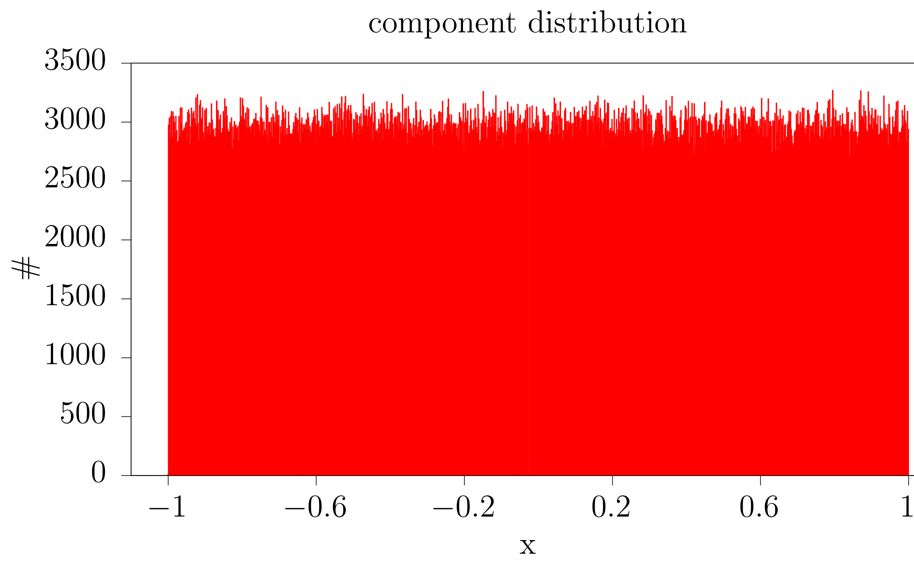


Figure 10: Distribution of x-components after 1000 loops

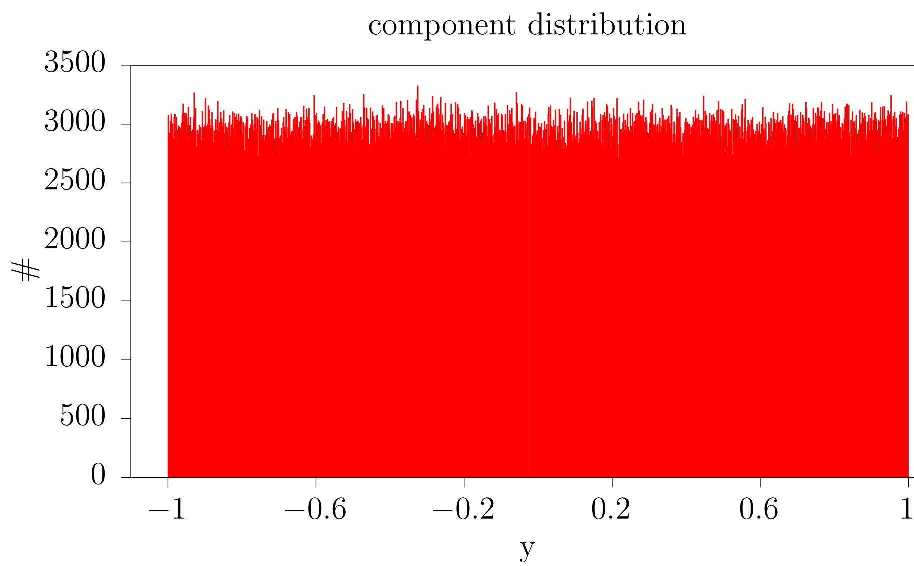


Figure 11: Distribution of y-components after 1000 loops

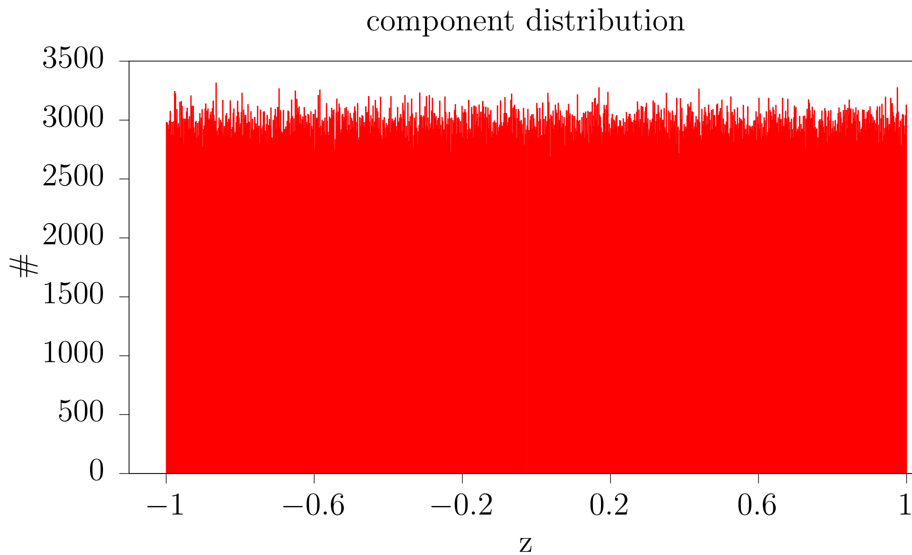


Figure 12: Distribution of z-components after 1000 loops

In this second set of graphs, we see that a sufficient number of transformation steps produces a distribution that is indistinguishable from a uniform distribution.

3.4.3 MC simulations and Ergodicity

We give a definition of ergodicity in the language of Monte-Carlo simulations on realistic machines with finite precision:

> **Definition: Ergodicity**
> In the context of a MC simulation, ergodicity means that the implementation of the algorithm ensures that all points in the simulated phase space of the system are eventually visited after a finite number of simulation steps.

Because of the finite nature of the precision of the machine, the phase space is also finite, albeit quite large. This is a subtle difference to the original definition of ergodicity in an uncountable phase space where each point merely is arbitrarily closely matched after sufficient, but also finite time.
In this thesis, we are interested in measurements taken at thermal equilibrium and therefore ergodicity is generally *assumed* on the level of microstates. This assumption is also referred to as *ergodic hypothesis*

> **Ergodic Hypothesis**
> Thermodynamic systems evolve in a way that all energetically allowed regions in phase space are covered. The time that the trajectory stays in a particular region of phase space is proportional to the phase-space volume of this region.

The consequence of the above is that a necessary condition for our implementations is that one must choose a combination of d_m and number of MCS that allows any microstate evolving to any other microstate. Additionally, on average there must not be any bias left in favor of phase space regions 'closer' to the original region. The last point implies that a distribution like in figure 10 is fine while a distribution like figure 7 is in violation of the ergodic hypothesis and thus cannot correspond to a system at thermal equilibrium.

4 Ferromagnetic Nanoparticles and Supercrystals in the Non-interacting Limit

The main goal of this thesis is to further the knowledge of dipolar magnetic systems. Because the interactions are long-range, a theoretical treatment is presently impossible. Comparing the simulation results with experiments will at least provide answers to questions like 'Do we know which energies are important' and 'Is the single-domain approximation valid'?

4.1 Theory I

Throughout section 4 we will not need to consider any interactions between the magnetic moments of the nanoparticles. The physical properties we simulate here are therefore comparatively easy to treat rigorously. This will be discussed next.

4.1.1 Stoner-Wohlfarth Model

Two energies will be considered in this chapter.

- The magnetostatic energy of one nanoparticle magnetic moment ('superspin') **m** in an external magnetic field
  $E_m = -\,\mathbf{m} \cdot \mathbf{B} = -m\,B\,\cos(\theta)$
- The magnetocrystalline anisotropy energy $E_a = K\;V \sin^2(\delta)$

which is quantified according to the Stoner-Wohlfarth model for uniaxial anisotropy constants.

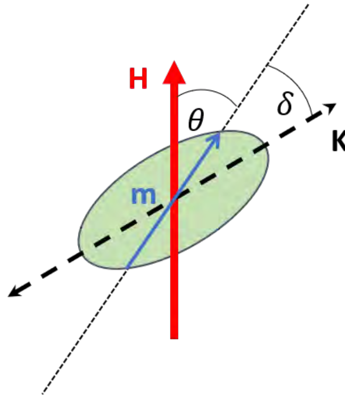


Figure 13: Geometry of the uniaxial Stoner-Wohlfarth model

$$E = E(\theta, \delta) = -mB\cos(\theta) + KV\sin^2(\delta)$$

or, in order to use the experimentally more common applied field strength **H**:

$$E = -\mu_0 H m\cos(\theta) + KV\sin^2(\delta)$$

4.1.2 Isotropic Paramagnet

First, we will ignore the second energy and try to recover the both analytically and experimentally well-known result for the magnetization curve of such a paramagnet in an external field. In nature, the vanishing of any magnetocrystalline anisotropy is not usual. The best analogy would be a material where the saturation magnetization is rather high compared to the anisotropy constant.
We define the total energy of the system being given as

$$E = -\mathbf{m} \cdot \mathbf{B}$$

where the modulus $|\mathbf{m}| = \mathbf{m}$ is constant and $\mathbf{B} = B\hat{\mathbf{e}_z}$. When calculating the expectation value of

$$m_z = m \cos\theta$$

in the canonical ensemble, we therefore have to consider all possible orientations of $\mathbf{m}$ on $\mathcal{S}^2$, the surface of the 3-sphere, according to:

$$\langle m_z \rangle = \frac{1}{Z} \int\limits_{\mathcal{S}^2} m_z \ \exp\left(\frac{\mathbf{m}\cdot\mathbf{B}}{k_B T}\right) \ d^2 r$$

$$\text{with} \quad Z = \int\limits_{\mathcal{S}^2} \exp\left(\frac{\mathbf{m}\cdot\mathbf{B}}{k_B T}\right) \ d^2 r$$

$$\varphi\text{-Symmetry} \ \Rightarrow \langle m_z \rangle = m \ \frac{\int\limits_0^\pi \exp\left(\frac{mB\cos\theta}{k_B T}\right) \sin\theta\cos\theta \ d\theta}{\int\limits_0^\pi \exp\left(\frac{mB\cos\theta}{k_B T}\right) \sin\theta \ d\theta}$$

where we used standard spherical coordinates (r, θ, φ) and $z = r\cos\theta$. Defining:

$$x := \frac{mB}{k_B T} \qquad v := \cos\theta$$

we get the easily solvable

$$\langle m_z \rangle = m \ \frac{\int_{-1}^{1} v e^{xv} dv}{\int_{-1}^{1} e^{xv} dv} \tag{4.1}$$

The magnetic moments are indistinguishable. We therefore get for the total magnetization M if n is the number *density* of magnetic moments, i.e. the number of magnetic moments, each with modulus m, per Volume

$$M(T, B) = n\langle m_z \rangle = M_{\max} \ L\left(\frac{mB}{k_B T}\right) \tag{4.2}$$

with the solution of (4.1), the *Langevin function* L

$$L(x) = \frac{1}{\tanh(x)} - \frac{1}{x} = \frac{x}{3} + O(x^3)$$

and the saturation magnetization, read maximum achievable magnetization

$$M_{\text{max}} = n\, m$$

For small fields B, the magnetic susceptibility can be written as

$$\chi = \frac{M}{H} \sim \frac{\mu_0 M}{B} = \frac{n \mu_0 m^2}{3 k_B T} \tag{4.3}$$

In particular, we get $\chi \propto 1/T$ which is known as Curie's law and is an important hallmark of paramagnetic systems or their analogies. We can rewrite the Curie law as

$$\chi = \frac{C}{T} \tag{4.4}$$

with the Curie constant C which we here expect to be

$$C = \frac{n \mu_0 m^2}{3 k_B} \tag{4.5}$$

or, in case that we have particles of finite volume V_{part} and saturation magnetization *per particle volume* M_s :

$$\begin{aligned} M_{\text{max}} &= n\, M_s\, V_{\text{part.}} \\ C &= \frac{n \mu_0 (M_s V_{\text{part.}})^2}{3 k_B} \end{aligned} \tag{4.6}$$

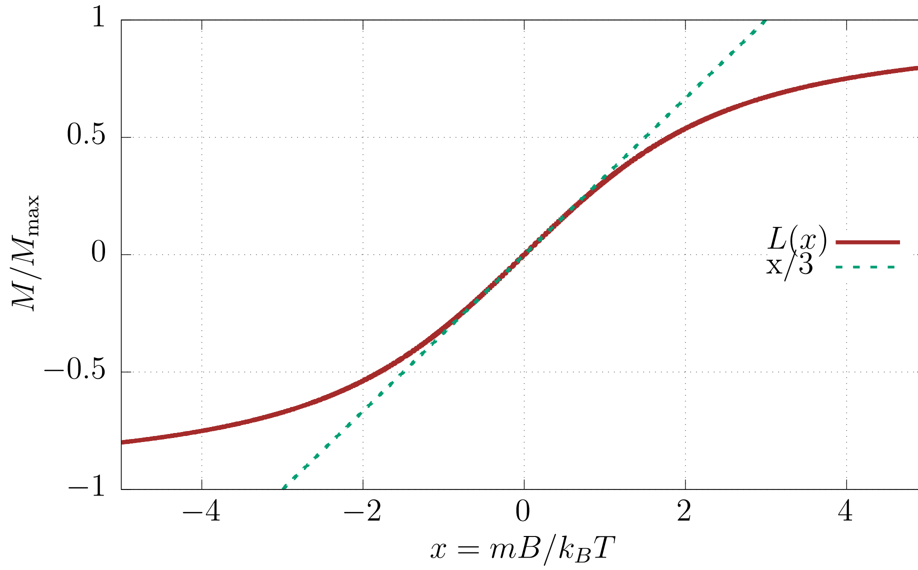


Figure 14: Langevin function and small-field approximation

4.1.3 Potential Landscape with Non-vanishing Anisotropy

We will now also include the magnetocrystalline anisotropy energy. It is important to note that we thereby have introduced at least two additional parameters: The magnitude of the anisotropy constant K and the distribution of the easy axes $\mathbf{k}$ where E_a is minimal if $\pm\mathbf{m} \parallel \mathbf{k}$.

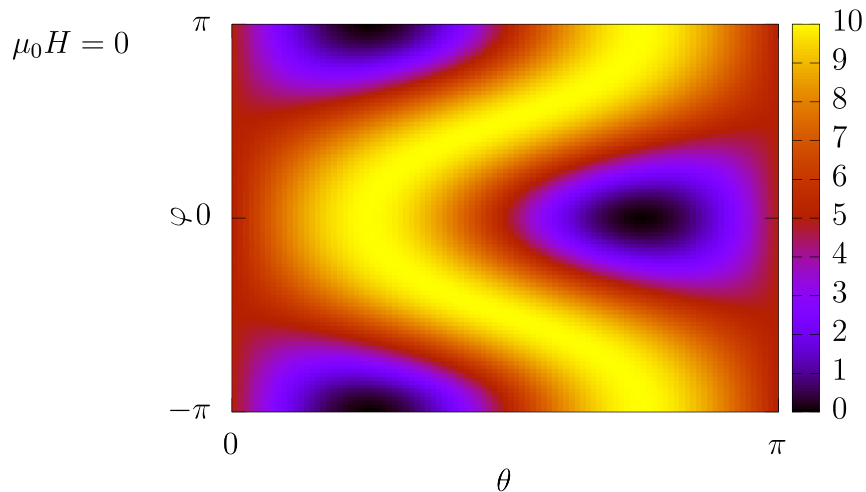


Figure 15: Potential landscape in spherical coordinates without external field. Energy in arbitrary units, following the given parameter set.

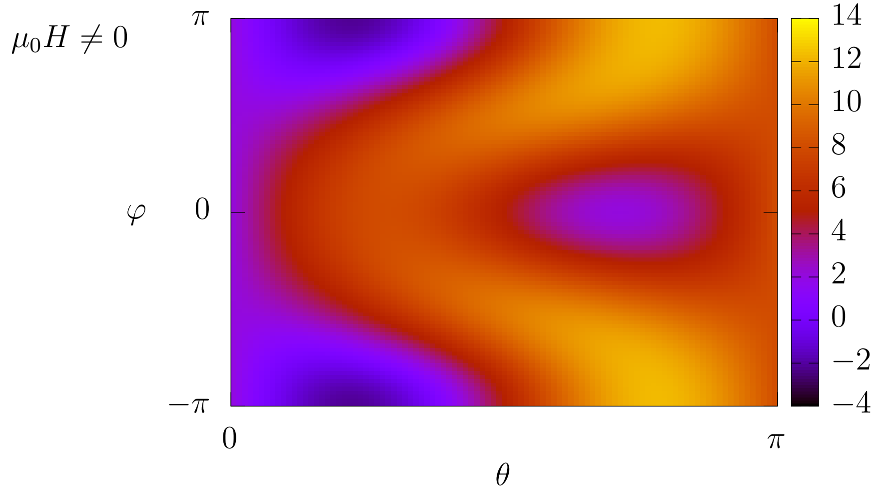


Figure 16: Potential landscape in spherical coordinates with external field

Figures 15 and 16 show the potential landscape for one superspin orientation in one setting of parameters in spherical coordinates $(\theta\,\varphi)$:

- $K\,V = 5 \qquad m\,B = 1.5 \qquad$ (arbitrary, non-physical units)
- $\hat{\mathbf{B}} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \qquad \hat{\mathbf{k}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$
- $E = E(\theta,\,\varphi) = K\,V\,\{1 - \cos^2[\delta(\theta,\varphi)]\} - m\,B\,\cos\theta)$
- $\cos[\delta(\theta,\varphi)] = \frac{1}{\sqrt{2}}(\sin\theta\cos\varphi - \cos\theta)$

Figures 17 and 18 show profiles for specific choices of φ. Note that any external field breaks the π-periodicity in θ into a 2π-periodicity.

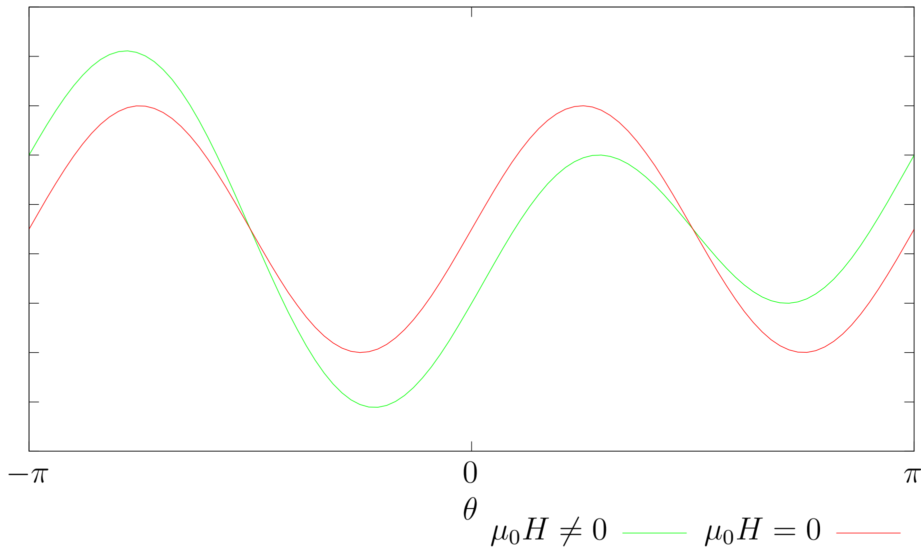


Figure 17: Profile of potential in figures 15, 16 for $\varphi = 0$

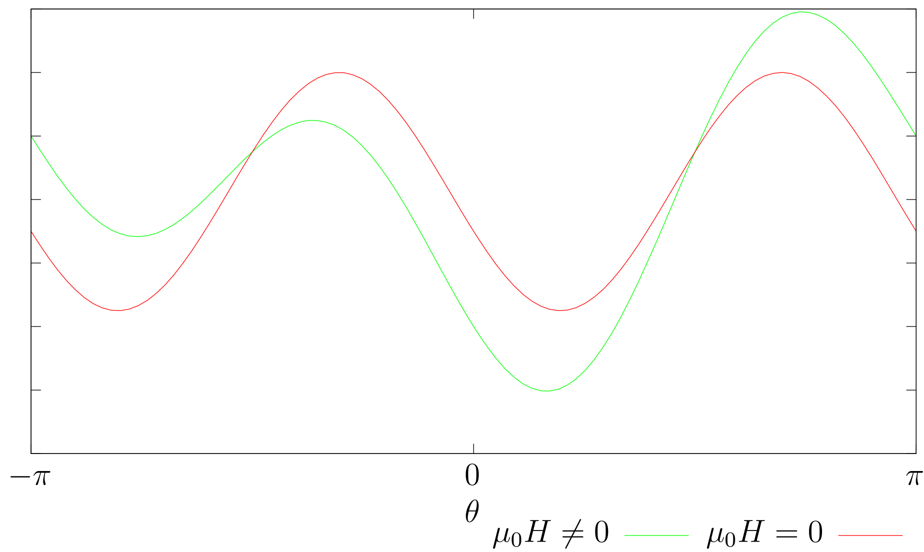


Figure 18: Profile of potential in figures 15, 16 for $\varphi = 3\pi/4$

4.1.4 ZFC-FC curves

A very common experimental procedure is recording the *Zero-field-cooled* (ZFC) and *Field-cooling* (FC) curves. The sample undergoes (at least partly) the following path and typically shows a magnetization curve depending on T as given in figure 19.

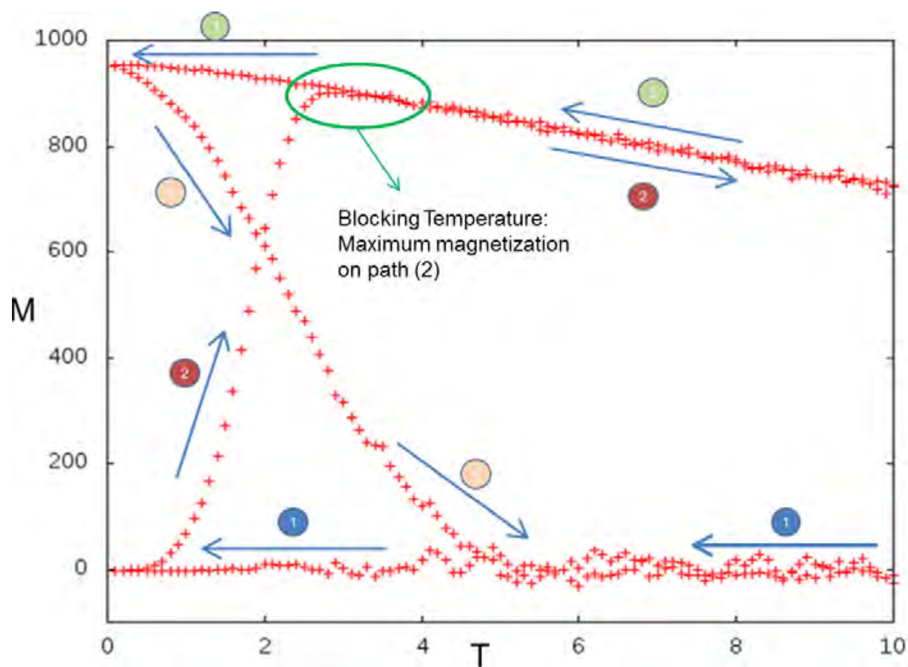


Figure 19: Idealized $M(T)$ curve with arbitrary units. Path 2, 3 are the ZFC, FC branches

Standard Temperature-Sweep Magnetometric Experiment

1. The system is randomly generated at a sufficiently large starting temperature with $\mu_0 H = 0$ and cooled down
2. System is heated with a constant $\mu_0 H$ applied (*ZFC curve*)
3. without change in the applied field, system is cooled again (*FC curve*)
4. at $H = 0$, is now heated again

The fourth branch is most often omitted whereas the first may not have a name but is still essential for branches 2 and 3.
One of the characteristic quantities of these magnetization curves is the *Blocking temperature* T_B which can (roughly) be defined as the temperature where during the ZFC-curve the magnetization reaches its maximum. The term blocking refers to the fact that above T_B, the system is expected to behave like a paramagnet. Since the involved magnetic moments are the nanoparticle superspins, this behavior is refered to as *Superparamagnetism*.

4.1.5 ac-Susceptibility

In case of ideal monodisperse nanoparticles as we mostly consider them, the relaxation time of particles obeys the Néel formula

$$\tau \simeq \tau_0 \exp\left(\frac{\theta}{T}\right) \quad \frac{\theta}{T} \equiv \frac{KV}{k_B T}$$

with volume V and uniaxial anisotropy constant K
we furthermore recall the static susceptibility (Langevin susceptibility) χ_0

$$\chi_0 = \frac{(M_s V)^2}{3 k_B T}$$

with saturation magnetization M_s. Then, for an applied ac field, one finds

$$M(t) = H_0 \left(\chi' \cos \omega t + \chi'' \sin \omega t\right)$$

This implies that the magnetization will try to follow the applied field but would do so with a phase lag.
We present two cases of an ac-susceptibility experiment:

1. Apply a saturating field onto the system and have it relax before applying the sinusoidal H_{ac}
   $\rightarrow \chi_{\mathrm{SAT}}$

2. Have the sample cooled in zero-field and then switch on H_{ac}
   $\rightarrow \chi_{\text{ZFC}}$

In the first case, one assumes that after switching off the saturating field, any single particle will undergo the following exponential decay (Néel relaxation)

$$M(t) = M_{eq} e^{-t/\tau}$$

This leads to the following real and imaginary part of the susceptibility in the model of Néel relaxation after saturation [1]:

$$\chi'_{\text{SAT}}(\omega) = \chi_0 \, \frac{1}{1 + (\omega\tau)^2}$$
$$\chi''_{\text{SAT}}(\omega) = \chi_0 \, \frac{\omega\tau}{1 + (\omega\tau)^2}$$

The second case also exhibits Néel relaxation as basis for the magnetization dynamics. However, an important difference is that as soon as any field is applied to the zero-field cooled sample, a net magnetization occurs instantaneously (as described in 4.1.4) which then leads to a temperature-independent nonzero contribution to χ_{ZFC}.
The real-time result for the ZFC susceptibility χ becomes [1]

$$\chi_{\text{ZFC}}(t) = \frac{M_s^2}{3K} \left[1 + \frac{KV}{k_B T} \left(1 - e^{-t/\tau} \right) \right]$$

and a Fourier transformation yields

$$\chi'_{\text{ZFC}}(\omega) = \frac{M_s^2}{3K} \left[1 + \frac{KV}{k_B T} \frac{1}{1 + (\omega\tau)^2} \right] \tag{4.7}$$
$$\chi''_{\text{ZFC}}(\omega) = \frac{M_s^2}{3} \frac{V}{k_B T} \frac{\omega\tau}{1 + (\omega\tau)^2} \tag{4.8}$$

Only simulation results for the ZFC-ac susceptibility and the first setup will be presented. In future research however, the alternative setup may be of particular interest if samples with non-vanishing interactions are considered. Via eliminating ωt we obtain the following relationship between χ', χ'':

$$\left(\chi' - \frac{\alpha(2+\sigma)}{2} \right)^2 + \chi''^{\,2} = \left(\frac{\alpha\sigma}{2} \right)^2 \tag{4.9}$$

where

$$\alpha \equiv \frac{M_s^2}{3K} \qquad \sigma \equiv \frac{KV}{k_B T}$$

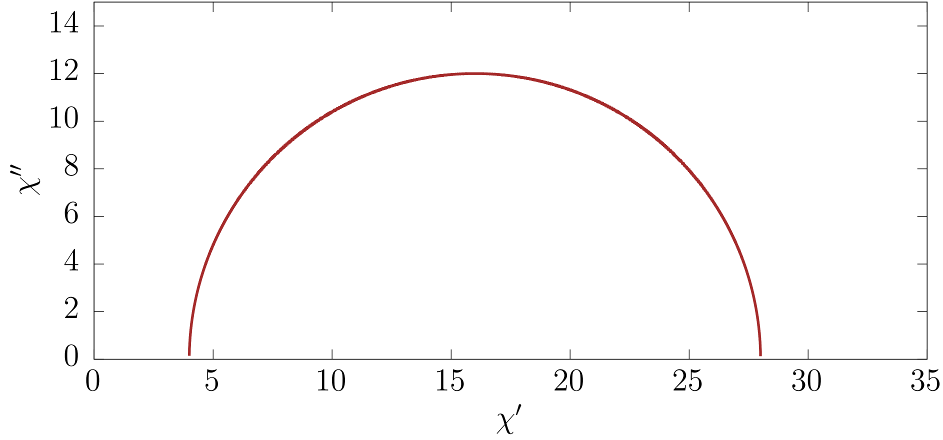


Figure 20: Theoretical result for non-interacting Cole-Cole plot with $c = 16$, $r = 12$

we obtain the formula of a semi-circle.

This means that the so-called Cole-Cole plot of real vs. imaginary part of χ_{ac} is a perfect semi-circle in the first quadrant with its center coordinates

$$(c,\ 0) \equiv (\alpha(2+\sigma)/2,\ 0)$$

and radius

$$r = \alpha\sigma/2$$

Furthermore, both the anisotropy constant and saturation magnetization can be extracted from such a circle plot

$$KV = 2k_B T\ \frac{r}{c-r}$$
$$M_s = \sqrt{6\frac{k_B T}{V}}\ \sqrt{r}$$

4.2 Simulation results

Throughout the following chapters, we will almost exclusively consider *maghemite* (γ-Fe_2O_3) [2] nanoparticles with uniformly distributed easy axes distribution and realistic parameters as they are given in nature.

4.2.1 Isotropic Superparamagnetism

We study the limit of vanishing magnetocrystalline anisotropy for magnetic nanoparticles and aim to recover the behavior of an isotropic paramagnet.

To this end, at constant temperatures several magnetic fields are applied to the nanoparticles and the magnetization is recorded. Results are given in figure 21.

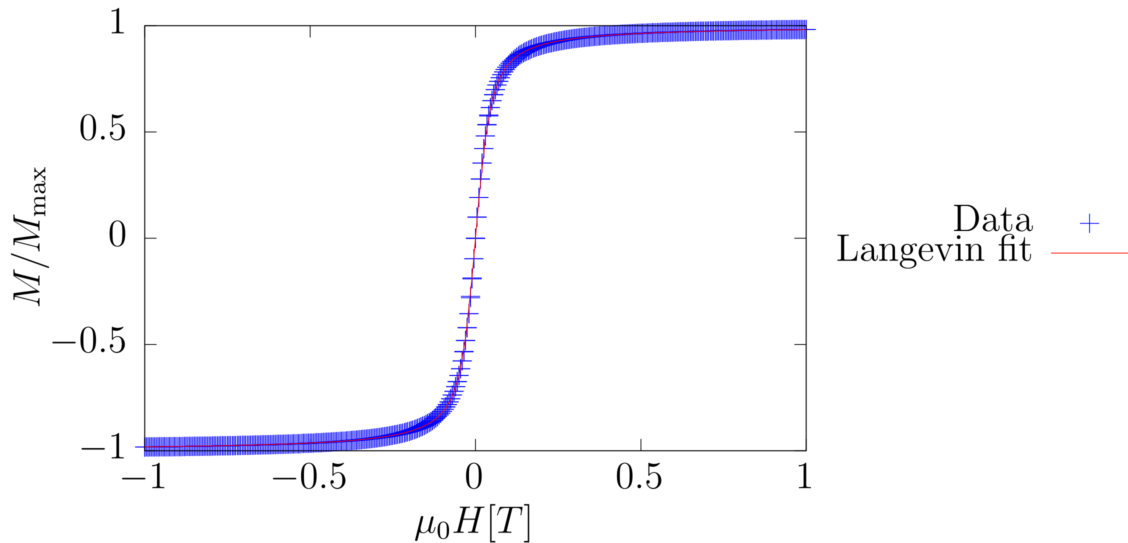


Figure 21: Simulation results for the isotropic paramagnet and the fit of the Langevin expectation

Fitting the theoretically expected Langevin function lets us recover the saturation magnetization that we implemented perfectly. We have therefore shown that our Monte-Carlo approach is in accordance with both theory and experiment.

4.2.2 ZFC-FC Curves

Several magnetometric simulations were carried out where appropriate temperature intervals were sweeped in the presence of static magnetic fields.

4.2.2.1 High-Temperature Behavior

We are interested in the behavior of the ZFC curve well above T_B where theory predicts the sample to behave like an isotropic paramagnet.
In particular, one expects the magnetization to follow a Curie-Weiss law

$$M(T) = M_\infty + \frac{C}{T - T_C}$$

with both $M_\infty = 0$ and $T_C = 0$ because anything other would indicate a non-vanishing permanent magnetization after the zero-field cooling or (anti-)ferromagnetic behavior. Also, the Curie constant C can be compared to equation (4.6)
There is nothing in our model allowing for either. Specifically, any $T_C \neq 0$ would indicate interactions among our superspins that would have to originate from errors in the implementation or a poor quality in the utilized pseudo-RNG.

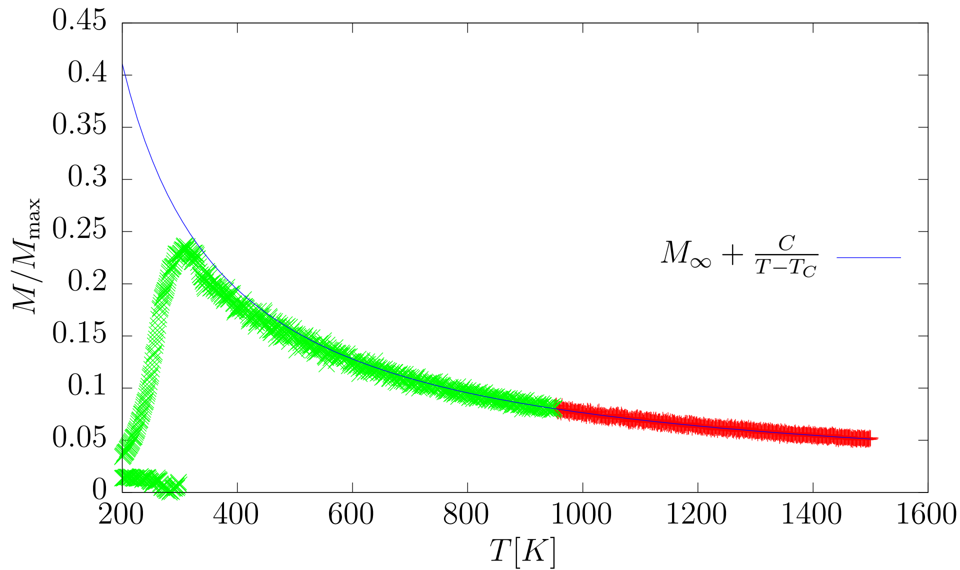


Figure 22: Fitting Curie-Weiss law onto the high-temperature tail of the ZFC-curve at $\mu_0 H = 2\mathrm{mT}$

We find that our computer experiment is in accordance with the expectation.

4.2.3 Influences on the Blocking Temperature

When doing a simulation, we still have to specify the following parameters:

- number of particles
- modulus of the test vector (see figure 6)
- number of Monte-Carlo steps (distributed among relaxation- and averaging-loops) per measurement point
- magnitude and direction of the applied magnetic field $\mu_0 H$
- distribution and magnitude of uniaxial anisotropy constants

Each of these parameters may influence the observed blocking temperature in our computer experiment. (As it turns out, all of the above do so, at least in the case of interacting, periodic systems which are discussed later).

4.2.3.1 Particle Number

Figure 23 shows simulations of the ZFC-curve for a different number of superspins in the sample but otherwise same parameters:

- Test vector modulus $|\mathbf{d}_m| = 1$
- Monte-Carlo steps per point: $2000^{(\mathrm{relaxation})} + 10^{(\mathrm{averaging})}$

- applied field $\mu_0 H = 15$ mT

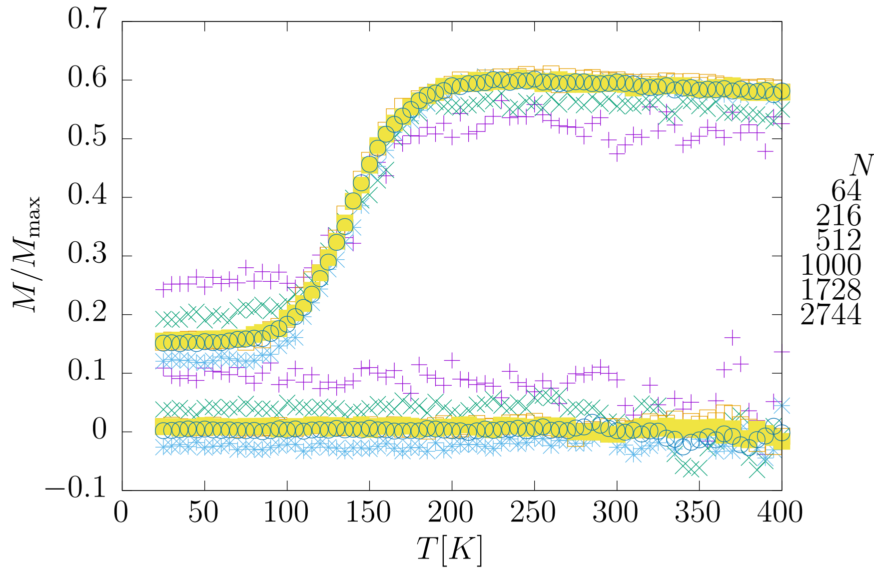


Figure 23: ZFC curves with varying number of particles

We see that for low N, the statistical quality of the simulation is poor since for example no T_B can be extracted well. Other than that, we do not observe different dynamics for greater N, but less noisy data. This is to be expected because we do not consider any interaction between particles in this chapter.

4.2.3.2 Test Vector

Figure 24 shows simulations of the ZFC-curve for different test vectors moduli in the sample but otherwise same parameters:

- number of magnetic moments: 1000
- Monte-Carlo steps per point: $2000^{(\text{relaxation})} + 10^{(\text{averaging})}$
- applied field $\mu_0 H = 15$ mT

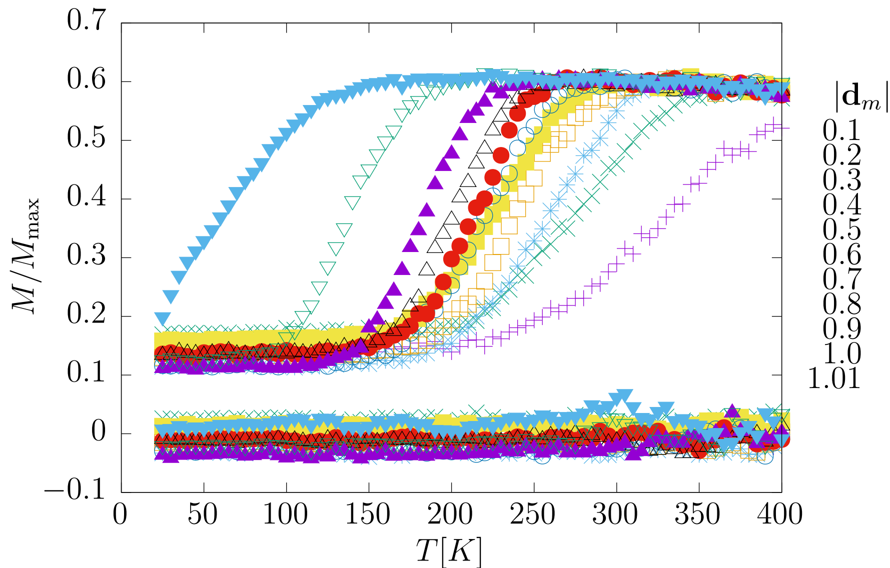


Figure 24: ZFC curves with varying modulus of test vector $|\mathbf{d}_m|$

This is probably the most involved parameter study in 4.2.2. The analysis of the component distribution for the transformed vectors as demonstrated in 3.4.2 shows for all d_m that were considered here, that after 2000 MCS the vectors would be uniformly distributed *in the absence of any potential.* Yet we see dramatically different behaviour in terms of observed T_B. We see an increase of T_B for decreasing d_m, which is intuitive because a smaller d_m corresponds to a slower dynamic of any individual nanoparticle whereas the speed of temperature change is constant in all 11 plots shown. Especially the non-linear behaviour around $d_m = 1$ however is only plausible if we recall this geometric insight: For this test-vector length, a jump 'from one pole to the equator' is barely possible. This is precisely the distance between the minima of the Stoner-Wohlfarth model. This means that for $d_m > 1$, a jump from the energetically unfavourable equilibrium state to the favourable one is possible (though still not very likely) within one Monte-Carlo update, which seems counterintuitive for systems at low temperature. For smaller d_m where this quick channel is completely forbidden, which leads to significantly slower dynamics.
The geometric significance of the test-vector length must thus be taken into account when designing simulation parameters for other energy landscapes. A balance must be found between the economy of wasting too much computation time in uneventful parts of phase space ($\leftrightarrow d_m$ chosen too low) and the phenomenon of spins leap-frogging potential landscapes at low temperatures ($\leftrightarrow d_m$ chosen too big).

4.2.3.3 Number of Monte-Carlo Steps per Measurement

Figure 25 shows simulations of the ZFC-curve for different numbers of *relaxation*-Monte-Carlo steps per measurement point but otherwise same parameters, including *averaging* Monte-Carlo steps:

- number of magnetic moments: 1000
- Test vector modulus $|\mathbf{d}_m| = 1$
- Monte-Carlo *averaging* steps per point: 10
- applied field $\mu_0 H = 15$ mT

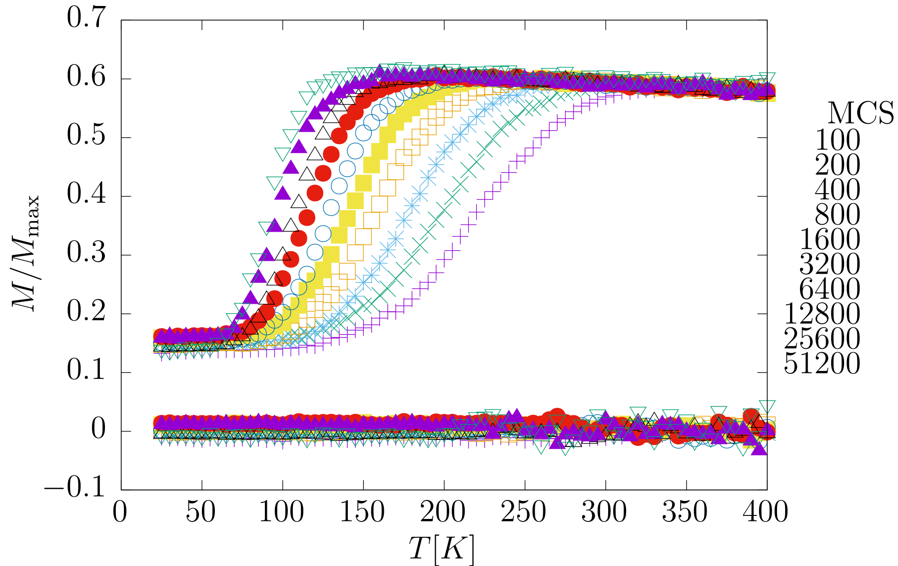


Figure 25: ZFC curves with varying number of Monte-Carlo steps (MCS)

In contrast to the previous case, the number of MCS at fixed test-vector length is a well-behaving parameter. An increase of MCS yields lower T_B which is the expected behaviour because any magnetic moment is allowed more time per temperature step to find its equilibrium. However, an increase of MCS leads to a convergent series of possible observations, precisely because our algorithm is based on Markov-Chains.
In contrast to the previous example, the parameter change here does not constitute a different physical system being studied. For higher MCS, we enhance the statistical quality of our simulation.

4.2.3.4 Applied Magnetic Field

Figure 26 shows simulations of the ZFC-curve for different applied magnetic fields after cooling in zero-field, but otherwise same parameters.

- number of magnetic moments: 1000
- Test vector modulus $|\mathbf{d}_m| = 1$
- Monte-Carlo steps per point: $2000^{(\text{relaxation})} + 10^{(\text{averaging})}$

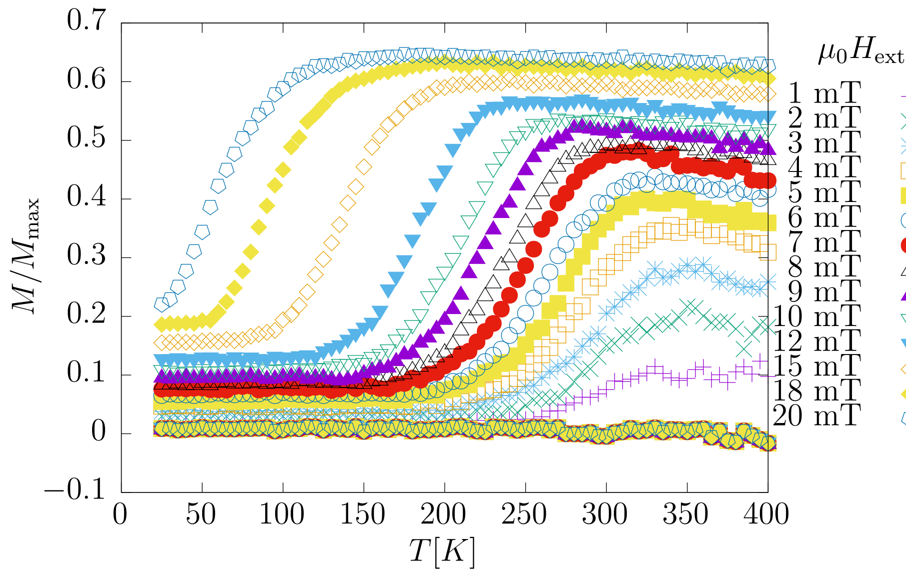


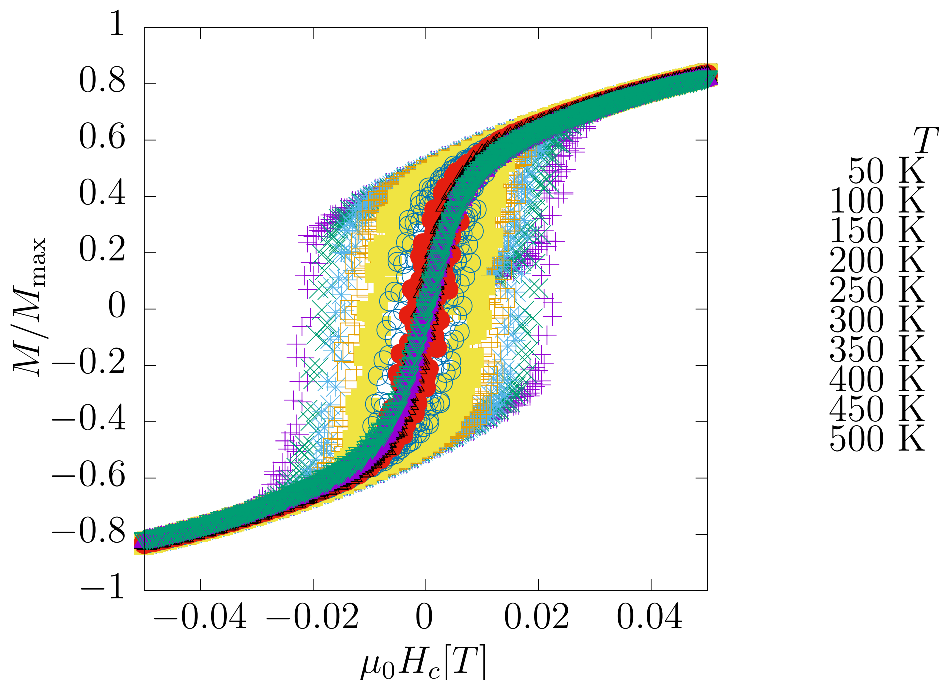
Figure 26: ZFC curves with varying applied field $\mu_0 H_{ext}$

The resulting series of ZFC curves shows, as expected, that T_B decreases when the applied field is increased.

4.2.4 Hysteresis Plots

Alternatively to cooling and heating the sample at discrete applied magnetic fields, it is insightful to apply varying external fields at constant temperature on the sample after it has been cooled down at zero external field.

In the limit of vanishing anisotropy constants, this procedure will reproduce our previous results for the isotropic paramagnet. For the more realistic case of finite KV, we will encounter the well-known phenomenon of open hystereses.



Comparing hysteresis loops at different temperatures will give a relationship between coercivity and temperature. The coercive field is defined as the strength of the applied magnetic field required to reduce the magnetization of the sample after it had been driven to saturation.

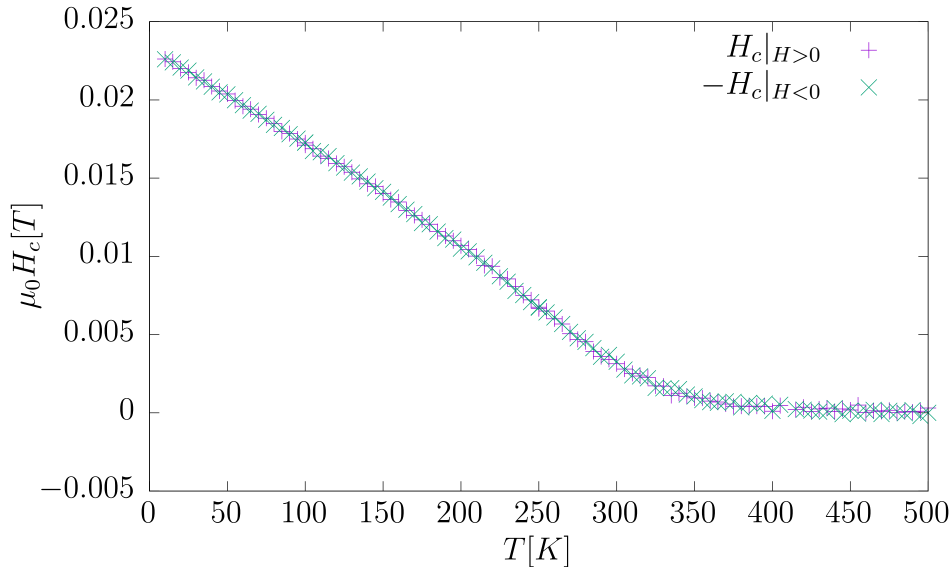


Figure 27: Temperature dependence of observed coercivity. The hysteresis is symmetric as shown by the overlapping of the curves

This behavior is also in accordance with experimental results. However, a theory of the precise analytical nature of the function $H_c = H_c(T)$ would have to involve the nature of modeled/implemented relaxation times τ as discussed in the context of Neél-Brown theory.
Another (technical) aspect is that the presented simulation results were achievable with far lesser computation time that comparable T-sweep simulations but with same statistical quality. Quite often it is observed that Metropolis-algorithm simulations at constant temperature yield much faster convergence.
Unfortunately, in the context of systems that feature frustration, such as most realistic magnetic systems do, experiments where the temperature is not held constant offer more insight.

4.2.5 ac-Susceptibility and Cole-Cole Plot

Finally, before moving on to interacting systems, we cover a third kind of sweep through parameter space. We want to investigate the AC-, or complex susceptibility χ_{ac} in the environment of a sinusoidal applied field at frequency ω

$$\chi_{ac}(\omega) = \chi'(\omega) - i\chi''(\omega) \quad \chi', \chi'' \in \mathbb{R}$$

Studying an explicitly time-dependent quantity is interesting from a technical point of view because we will directly make use of the interpretation of '1 Monte-Carlo step' as representative of a finite if extremely small period of time.

$$\mathbf{H}_{\mathbf{ext}} \equiv \mathbf{H}(t_{MC}) = \mathbf{H}_0 \sin(\omega t_{MC})$$

is the physical field we want to simulate. We introduced a 'Monte-Carlo time' t_{MC} which we can implement in a straightforward fashion in a computer experiment by a certain number of Monte-Carlo steps. For χ', χ'' we then have

$$\chi'(\omega)\mathbf{H}_0 = \frac{1}{N_{MC}} \sum_{t_{MC}=1}^{N_{MC}} \mathbf{M}(t_{MC}) \sin(\omega t_{MC})$$

$$\chi''(\omega)\mathbf{H}_0 = \frac{1}{N_{MC}} \sum_{t_{MC}=1}^{N_{MC}} \mathbf{M}(t_{MC}) \cos(\omega t_{MC})$$

Throughout this thesis we always consider magnetic samples that are considered linear and isotropic media. Thus, we can omit the vectors and χ is scalar instead of a tensor of degree ≥ 2.

As one can plainly see from the definition above, high frequencies are implemented via a larger number of MC steps. This approach works fine if the computational effort behind one MCS is small, but it will be a great obstacle when treating interacting systems.

Figure 28 shows the results for the temperature dependence of the real and imaginary part of χ at fixed frequency.

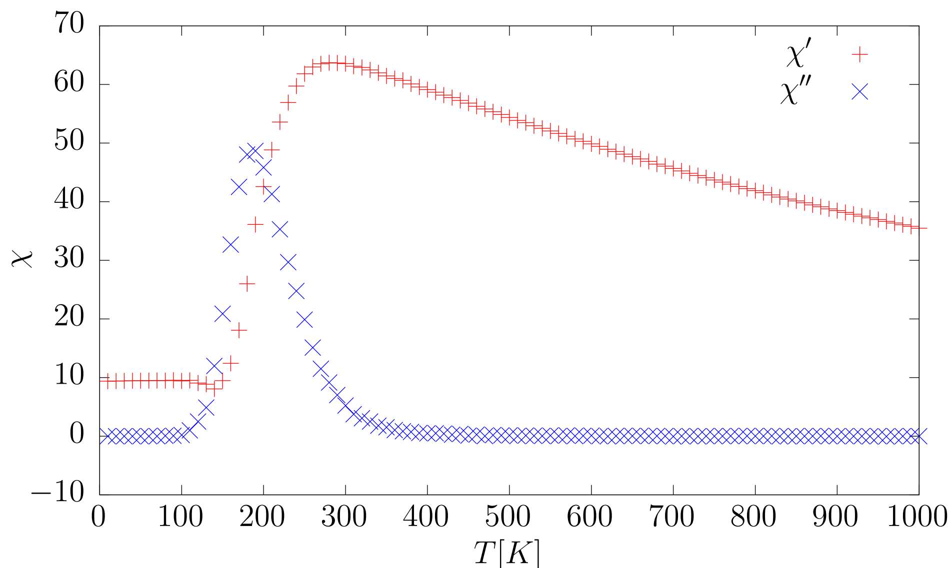


Figure 28: Real and imaginary part of the ac-susceptibility at $N_{\mathrm{MCS}} = 10^6$

A frequency sweep at fixed temperature, conversely yields the Cole-Cole plot in Fig. 29:

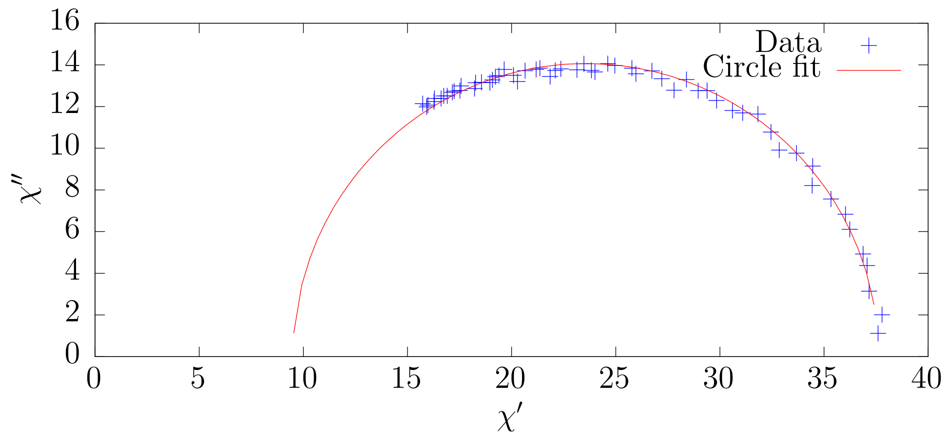


Figure 29: Cole-Cole plot and with fitting of a circle function from (4.9)

We find that our result confirms the expectation of a non-interacting system because a perfect semi-circle is observed. In contrast, an interacting system would show a strongly flattened circle.

4.3 Summary and Discussion of Results

Non-interacting systems have been comprehensively studied and we have collected sufficient evidence that our simulation approach is correctly implemented. We have found that simulation parameters such as the number of MCS per measurement have significant influence on any quantifyable observation.

5 Ferromagnetic Nanoparticles with Dipole-Dipole Interaction

From now on, we will consider nanoparticles forming supercrystals at realistic length scales (10^{-8} to 10^{-7} m). Here, it is no longer a realistic assumption to consider the nanoparticles as non-interacting. Instead, our model of supercrystals will include magnetic dipole-dipole interactions as additional energy contribution.
Theoretical aspects of this model are discussed first, especially computational challenges arising from any 2-particle interaction and the comparatively sparse theoretical knowledge regarding 3D models with a rather complicated interaction like the dipole-dipole interaction.
Motivated by this, we discuss widely used approximative approaches towards these models and re-introduce a method originally used in molecular physics in order to apply a customized version for our models.

5.1 Theory II

As noted, the only difference compared to the previous section is the introduction of realistic 2-particle interactions in our models. However, this small change has major implications for code efficiency and other aspects which were rather simple in the case of non-interacting nanoparticles.

5.1.1 Computational Aspects

Recalling the Metropolis-Hastings algorithm from (3.6) we see that an obvious bottleneck for code efficiency is the estimation of the total energy change of the system. This effect is far larger in case of 2-particle interactions because it will imply a $\mathcal{O}(N^2)$-scaling with particle number, or correspondingly $\mathcal{O}(L^6)$-scaling with edge length L.
On the other hand, we note that this energy estimation is entirely deterministic. This is important because we can then at least utilize multi-threaded computations to speed this step up. This will be done and discussed in the sections which do not employ different approximative theories. We note however that, purely from a computer efficiency point of view, it will always pay off to reduce the amount of 2-particle interaction computations.

5.1.2 Mermin-Wagner Theorem

Another challenge in this thesis is the absence of comprehensive theoretical knowledge what a dipolar 3D system is expected to behave like.
Although it is not the focus of this research, we recall an important result for sys-

tems of lower dimensionality, the Mermin-Wagner theorem [14]:

Mermin-Wagner Theorem

In 1D and 2D systems with sufficiently short-range interactions, no continuous symmetry can be broken spontaneously. This means that any thermal fluctuation offers sufficient perturbation destroying a possible ordering.

This result rigorously only applies to an isotropic Heisenberg ferromagnet, but generalizations towards various many-body systems exist [6]. This possesses rotational symmetry so that all the spin directions can be globally rotated without any additional energy cost. This means that long wavelength excitations, in which the spin state may deviate from its ground state value over a considerable distance, cost very little energy. Thus a fluctuation of the spins can be excited with very little energy cost. In one and two dimensions, they destroy the long range order. If, however, there is significant anisotropy there will be an energy cost associated with rotating the spins from their ground state value.
It turns out that the anisotropy energy penalty incurred by allowing these fluctuations increases with the square of R, the radius of the excitation, and hence the anisotropy energy will suppress all but the smallest of these non-linear fluctuations. It is the presence of such symmetry breaking fields which can stabilize long range order in two-dimensional systems. There is also a dipolar interaction between spins in real systems which, although much weaker than the exchange interaction, is anisotropic and can act in a similiar way to suppress the growth of fluctuations.

5.1.3 Antiferromagnetic Part of the Dipole-Dipole Interaction

When writing down the full dipole-dipole energy between two magnetic moments $\mathbf{m}_1$,$\mathbf{m}_2$

$$\begin{aligned} E_{\mathrm{dip}} &= \frac{\mu_0}{4\pi r^3}\left[\mathbf{m}_1\cdot\mathbf{m}_2 - 3(\mathbf{m}_1\cdot\hat{\mathbf{r}})(\mathbf{m}_2\cdot\hat{\mathbf{r}})\right] \\ &= \underbrace{\frac{\mu_0}{4\pi r^3}\left(\mathbf{m}_1\cdot\mathbf{m}_2\right)}_{=:E_{\mathrm{afm}}} \underbrace{-\frac{3\mu_0}{4\pi r^3}\left(\mathbf{m}_1\cdot\hat{\mathbf{r}})(\mathbf{m}_2\cdot\hat{\mathbf{r}}\right)}_{=:\tilde{E}_{\mathrm{dip}}} \end{aligned} \tag{5.1}$$

we can identify two contributions which we call the *antiferromagnetic* part E_{afm} and residual

$$\tilde{E}_{\mathrm{dip}} \equiv E_{\mathrm{dip}} - E_{\mathrm{afm}}$$

We can now imagine a hypothetical lattice model where the Hamilton function or Hamiltonian is completely defined by the 2-particle interaction given by $E_{\mathrm{afm}} =$

$\frac{\mu_0}{4\pi r^3}\left(\mathbf{m}_1 \cdot \mathbf{m}_2\right)$. We observe that this resembles the conventional Heisenberg model Hamiltonian

$$\mathcal{H}_{\text{Heis}} = -\sum_{i,j} J_{ij}\, \mathbf{S}_1 \cdot \mathbf{S}_2 \tag{5.2}$$

where J_{ij} is the exchange integral between two spins $\mathbf{S}_1$, $\mathbf{S}_2$. The analogy we propose is therefore

$$E_{\text{afm}} \leftrightarrow \mathcal{H}_{\text{Heis}}$$
$$\mathbf{m}_i \leftrightarrow \mathbf{S}_i$$
$$\frac{\mu_0}{4\pi r^3} \leftrightarrow -J_{ij}$$

Most importantly, we see that our 'exchange integral'

$$-\frac{\mu_0}{4\pi r^3} \mathrel{\hat{=}} J_{ij} < 0 \quad \forall\,\text{pairs}\,(i,j)$$

which implies that antiferromagnetism is the expected type of collective magnetism if the system has the appropriate translational symmetry.

5.1.4 Finite and Infinite Systems

It is possible to study finite systems non-approximatively. By this we mean that one can in principal evaluate the complete energy of any interacting system as long as the number of considered magnetic moments is finite. Computational constraints will however restrict such simulations in terms of maximum number of moments in the system:
As discussed in section 5.1.1, the computational effort necessary to exactly calculate the necessary energy terms for each MCS scales with $\mathcal{O}(N^2)$. The simulation of one measurement (i.e. one instance of the parameters T and $\mu_0 H_{\text{ext}}$) for a system of $N \sim 10^3$ particles may take as long as 1 hour on machines that are available to me. Realistically, any obtainable sample of self-assembled nanoparticles in a superlattice will consist of no fewer than $N \sim 10^9$ contributing superspins which will already imply an estimated computation time factor of 10^{12}. Therefore, what took 1 h before, would require several orders of magnitude longer.
This estimate ignores the fact that in the implementation used in this thesis (see appendix B.1.4) invariant parameters like site distances d and displacement vectors $\hat{\mathbf{r}}$ are calculated once and kept in working storage. For $N \sim 10^3$, this requires ~ 100 MB of RAM in the implementation given in the appendix. This could of course be cut down drastically if one sacrifices the flexibility of our code in terms of which superlattices can be put in, but it will still require $\sim 10^3$ KB of RAM and the same multiplication factor as before yields $\sim 10^9$ GB of required RAM. One could of

course perform simulations without putting these recurring parameters into storage, but this will in turn slow down the performance even further.
Because of these harsh limitations, we will focus on ensembles of $N < 10^5$ superspins. These are either considered

1. finite systems where surface effects will prohibit any meaningful insight into bulk behavior

2. supercells that are continued indefinitely with periodic boundary conditions in order to simulate bulk behavior

This thesis puts more emphasis on the second branch. Additional assumptions and approximations are required to effectively study interacting systems to a similar degree as was done for non-interacting systems in section 4.2.

5.1.5 Possible Approximations

We present a selection of possible approaches to simulate interacting lattice models effectively. We focus on methods that are still within the Metropolis scheme (3.6). This means that we focus on approximations that only affect the calculation of energies.

- **Approximations for finite systems**
  - Cut-off
- **Approximations for infinite systems with PBC**
  - Ewald method
  - Cut-off
  - Cut-off + Mean Field (Onsager)

The idea of introducing a cut-off length beyond which interactions are neglected is that the loss in precision of the calculated energy is far outweighed by the gain in computation efficiency.

5.1.6 Onsager Reaction Field Method

We will now cover in detail the adaptation of a mean-field theory applied mostly in molecular physics, for the purpose of getting reasonable energy estimations. It will offer a vast reduction of the needed computation time. A derivation from a general idea which implies a classical magnetostatic problem will be presented.

The mathematical problem itself is exactly solvable and will thus be presented thoroughly. Emerging parameters and their physical significance will be discussed.
Finally, both the achieved accuracy in simulations and the performance from an efficiency standpoint will be discussed.

5.1.6.1 Motivation and Model

In molecular physics, research on electric dipolar particles in fluids is an important topic with a long history. With electric dipolar fluids it is important to develop approximate methods to estimate energies stemming from two-particle interaction such as the electric dipole energy, similar to our simulation studies.
The dipole-dipole interaction has the same form whether one talks about the energy of electric permanent dipoles $\mathbf{p}$ within an external electric field $\mathbf{E}$ or magnetic dipoles $\mathbf{m}$ in an applied $\mu_0\mathbf{H}$. Therefore, it appears reasonable to take previous work on approximative methods there as motivation for our work on magnetostatics.
Starting from the complete model with long-range interactions, the first step in the Onsager-Reaction-field method [16] is to introduce a cutoff radius R which limits the spatial extension of any electric field $\mathbf{E}_d$ produced by an electric dipole

$$\mathbf{E}_d(\mathbf{r}) = \frac{1}{4\pi\epsilon_0}\frac{1}{r^3}\left(3(\mathbf{p}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}} - \mathbf{p}\right) \quad \mapsto \quad \mathbf{E}_d^{\mathrm{ORF}}(\mathbf{r}) = \Theta(R-r)\ \mathbf{E}_d(\mathbf{r}) \tag{5.3}$$

with the Heaviside function Θ

$$\Theta(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 1 \end{cases}$$

In the context of computer simulations, this is already the most important step of the entire approximation because it implies that only a fraction of interaction terms have to be computed with $\mathbf{E}_d^{\mathrm{ORF}}$ instead of $\mathbf{E}_d$.
On the other hand, taking this new model-interaction on its own is obviously too naive because one would quickly come to the conclusion that there were no qualitative difference between bulk behavior or the behavior of a tiny set of interacting dipoles.
The 2nd part of the ORF-method therefore provides an idea how to model the effect of the environment outside the cut-off sphere on the dipole in the center. The idea is to model these magnetic moments as a continuous, linear, isotropic, polarized medium and imagine it reacting towards a superdipole which sits at the exact center of the sphere which otherwise is comprised of vacuum. This medium will have a relative permittivity[3] $\epsilon_r > 1$ which is a real scalar and is a quantity which can also be calculated when measuring the total polarization of the entire set of electric

[3] More generally, $\epsilon_r \neq 1$ would suffice, but metamaterials with negative susceptibility $\chi_e \equiv \epsilon_r - 1$ are rarely considered

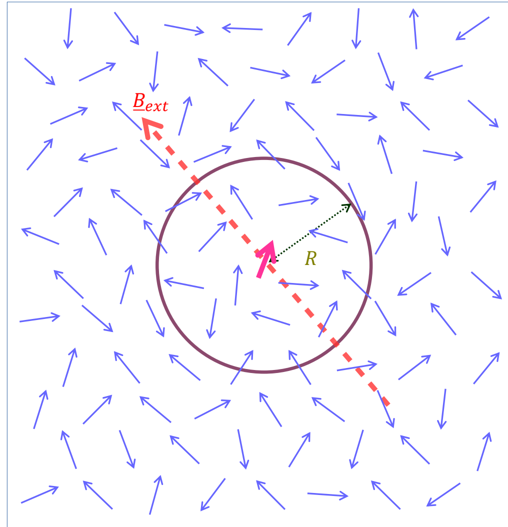


Figure 30: First step of the ORF method: Introduce cut-off R beyond which dipole-dipole interactions are not considered

dipoles in an external electric field.
From there, classical electrostatics will give an expectation for the *reaction field* $\mathbf{E}_{\mathrm{RF}}$, an additional electric field which is the response of the surrounding medium towards the presence of the dipole.
As stated, the magnetic field $\mathbf{B}_d$ of a magnetic dipole has the same form as in the electric case, which is why we can just propose the following $\mathbf{B}_d^{\mathrm{ORF}}$ providing a cut-off for the magnetic field analogously to (5.3)

$$\mathbf{B}_d(\mathbf{r}) = \frac{\mu_0}{4\pi}\frac{1}{r^3}\left(3(\mathbf{m}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}} - \mathbf{m}\right) \quad \mapsto \quad \mathbf{B}_d^{\mathrm{ORF}}(\mathbf{r}) = \Theta(R-r)\;\mathbf{B}_d(\mathbf{r}) \tag{5.4}$$

As in the original case of electric dipolar molecules, we now have to make an estimate for any environment of the magnetic dipole beyond the cut-off radius R: We imagine a continuous, linear, isotropic, magnetized medium with the relative permeability $\mu_r > 1$ surrounding the cut-off sphere which is comprised of vacuum ($\mu_r = 1$) and a magnetic super-dipole at its center. This super-dipole again is the direct sum of all magnetic moments contained in the cut-off sphere. The model is illustrated in figures 30 and 31.

5.1.6.2 Solution of the Magnetostatic Problem

Vector calculus states that any curl-free vector field is the gradient of a scalar field[4].

[4]Provided that mathematical conditions like continuity, differentiability and correct asymptotic behavior for $r \to \infty$ are met.

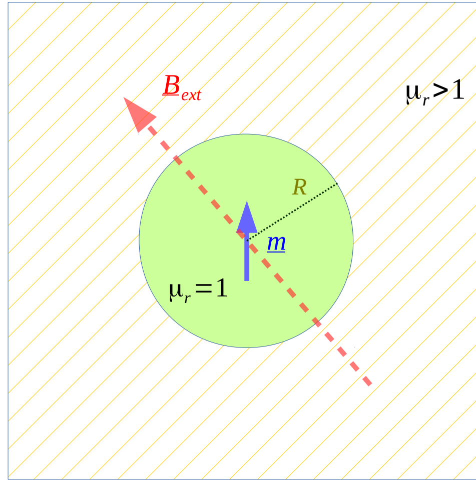


Figure 31: Second step of the ORF method: Onsager Reaction Field model as a problem in classical magnetostatics

This means

$$\nabla \times \mathbf{E} = 0 \quad \Rightarrow \quad \exists\ \phi(\mathbf{r}) : -\nabla\ \phi = \mathbf{E} \tag{5.5}$$

Therefore, the solution of boundary problems for any stationary *electric* field $\mathbf{E}$ and its potential ϕ correspond to a given charge density $\rho_e = \rho_e(\mathbf{r})$ via

$$\begin{aligned} \nabla\phi &= -\mathbf{E} \\ \text{div } \mathbf{E} &= \frac{\rho_e}{\epsilon_0} \\ \Rightarrow \quad \Delta\phi &= -\frac{\rho_e}{\epsilon_0} \end{aligned} \tag{5.6}$$

which is the well-known Poisson equation. Any $\phi(r)$ that solves the Poisson equation and also satisfies boundary conditions given by materials etc. is the unique solution to such a problem.
Finding a solution to (5.6) is relatively easy if the problem has any symmetries. Because of the Maxwell equation $\nabla \cdot \mathbf{B} = 0$ there is no analogous scalar potential for magnetic fields. Instead, one has a not easily found *vector* potential $\mathbf{A}$ such that $\mathbf{B} = \nabla \times \mathbf{A}$. This is given via the Biot-Savart law

$$\mathbf{B}(\mathbf{r}) = \frac{\mu_0}{4\pi} \int_V \frac{\mathbf{j}\ dV \times \mathbf{r}'}{|\mathbf{r}'|^3}$$

but cannot be evaluated easily since we first would have to find the correct static current density $\mathbf{j}$ which represents our permanent magnetic dipole $\mathbf{m}$.

It is therefore not immediately obvious that the derivation of an appropriate reaction field $\mathbf{B}_{RF}$ will be as easy as in the dielectric case.
However, we realize that there is no *free* current density in our problem, only *bound* currents which induce magnetization, i.e. a dipole-moment density. This will facilitate the mathematics immensely in the following way:
We use the auxiliary field $\mathbf{H}$ and a decomposition of the total charge current $\mathbf{j}$ into bound and free part:

$$\mathbf{H} = \frac{1}{\mu_0}\mathbf{B} - \mathbf{M} \tag{5.7}$$

$$\mathbf{j} = \mathbf{j}_{\text{free}} + \mathbf{j}_{\text{bound}} \tag{5.8}$$

$$\mathbf{j}_{\text{bound}} = \nabla \times \mathbf{M} \tag{5.9}$$

We study the derivatives of H:

$$\nabla \cdot \mathbf{H} = \frac{1}{\mu_0}\nabla \cdot \mathbf{B} - \nabla \cdot \mathbf{M} = -\nabla \cdot \mathbf{M} \tag{5.10}$$

$$\nabla \times \mathbf{H} = \frac{1}{\mu_0}\nabla \times \mathbf{B} - \nabla \times \mathbf{M} = \mathbf{j} - \mathbf{j}_{\text{bound}} = \mathbf{j}_{\text{free}}$$

Since there is no free current, we have found analogously to (5.5)

$$\nabla \times \mathbf{H} = 0 \quad \Rightarrow \quad \exists\, W(\mathbf{r}) : -\nabla\, W = \mathbf{H} \tag{5.11}$$

Combining (5.11) and (5.10) we have found the

Poisson equation of the magnetic potential

$$\Delta W = \nabla \cdot \mathbf{M} \tag{5.12}$$

just like (5.6) for electric fields. Now we can also make explicit use of our assumption that the outside medium is linear and isotropic:

$$\mathbf{M}_i \overset{\text{linear}}{=} \sum_{j=1}^{3} \chi_{ij}^{(m)} \mathbf{H}_j + O\left(\mathbf{H}^2\right)$$

$$\chi_{ij}^{(m)} \overset{\text{isotropic}}{=} \chi_m \delta_{ij}$$

$$\Rightarrow \quad \mathbf{M} = \chi_m\, \mathbf{H}$$

Most importantly for our calculations below, this implies that $\mathbf{B}$ and $\mathbf{H}$ are related to each other simply by

$$\mathbf{B} \equiv \mu_0(\mathbf{H}+\mathbf{M}) = \mu_0\mathbf{H} + \mu_0\chi_m\mathbf{H} = \mu\mathbf{H} \tag{5.13}$$

where

$$\mu = \mu_0(1+\chi_m) \equiv \mu_0\ \mu_r \tag{5.14}$$

If we assume that the surrounding medium with magnetization M is 'roughly paramagnetic'

$$1\ < \mu_r\ \lesssim 100$$

this assumption seems reasonable.
Very importantly, we hereby have a $\mathbf{H}_{ext}$ which is *not* constant throughout $\mathbb{R}^3$ like $\mathbf{B}_{ext}$ but piecewise defined as

$$\mathbf{H}_{ext} = \mathbf{B}_{ext} \begin{cases} \frac{1}{\mu_0} & r < R \\ \frac{1}{\mu_0\ \mu_r} & r > R \end{cases} \tag{5.15}$$

In this very special case of magnetostatic problems, the boundary conditions given by the Maxwell equations look completely analogous to the electrostatic case via

$$W \leftrightarrow \phi$$
$$\mathbf{H} \leftrightarrow \mathbf{E}$$
$$\mu_r \leftrightarrow \epsilon_r$$

Finally, we arrive at the following set of differential equations and boundary conditions.

Boundary conditions for the magnetic Poisson equation (5.16)

$$W(r\to 0) \to \frac{\mathbf{m}\cdot\mathbf{r}}{4\pi r^3}$$
$$W(r\to\infty) \to -\mathbf{H}_{ext}\cdot\mathbf{r}$$
$$W(r=R) = \text{continuous}$$
$$\mathbf{H}_{in}^{\perp} = \mu_r\mathbf{H}_{out}^{\perp}$$
$$\mathbf{H}_{in}^{\parallel} = \mathbf{H}_{out}^{\parallel}$$

Again, note that $\mathbf{m}$ now represents the total sum of dipole moments within V_c:

$$\mathbf{m} \equiv \sum_{V_c} \mathbf{m}_i \tag{5.17}$$

With (5.16), the problem is unfortunately very unsymmetric because the angle between $\mathbf{m}$ and $\mathbf{H}_{ext}$ will take arbitrary values.
In order to be able to use techniques for problems with azimuthal boundary conditions, we split the problem in two parts where either the cumulated magnetic moment $\mathbf{m}$ or the external field $\mathbf{B}_{ext} = \mu \mathbf{H}_{ext}$ are a symmetry axis:

$$\begin{aligned}
W^{\mathrm{I}}(r \to 0) &\to \frac{\mathbf{m} \cdot \mathbf{r}}{4\pi r^3} \\
W^{\mathrm{I}}(r \to \infty) &\to 0 \\
W^{\mathrm{I}}(r = R) &= \text{continuous} \\
\mathbf{H}_{in}^{\mathrm{I},\perp} &= \mu_r \mathbf{H}_{out}^{\mathrm{I},\perp} \\
\mathbf{H}_{in}^{\mathrm{I},\parallel} &= \mathbf{H}_{out}^{\mathrm{I},\parallel}
\end{aligned}$$

$$\begin{aligned}
W^{\mathrm{II}}(r \to 0) &\to 0 \\
W^{\mathrm{II}}(r \to \infty) &\to -\mathbf{H}_{ext} \cdot \mathbf{r} \\
W^{\mathrm{II}}(r = R) &= \text{continuous} \\
\mathbf{H}_{in}^{\mathrm{II},\perp} &= \mu_r \mathbf{H}_{out}^{\mathrm{II},\perp} \\
\mathbf{H}_{in}^{\mathrm{II},\parallel} &= \mathbf{H}_{out}^{\mathrm{II},\parallel}
\end{aligned}$$

The explicit derivation makes use of techniques to solve the homogenuous Laplace equation

$$\nabla^2 W = 0$$

and is given in the appendix A.1.
The coordinate-free representation for the solution of the two problems is:

$$\begin{aligned}
W_{\mathrm{in}}^{\mathrm{I}}(\mathbf{r}) &= \frac{\mathbf{m} \cdot \mathbf{r}}{4\pi} \left(\frac{1}{r^3} + \frac{1}{R^3} \frac{1 - \mu_r}{1 + 2\mu_r} \right) \\
W_{\mathrm{out}}^{\mathrm{I}}(\mathbf{r}) &= \frac{\mathbf{m} \cdot \mathbf{r}}{4\pi r^3} \, \frac{3}{1 + 2\mu_r} \\
W_{\mathrm{in}}^{\mathrm{II}}(\mathbf{r}) &= -\mathbf{H}_{ext} \cdot \mathbf{r} \, \frac{3\mu_r}{1 + 2\mu_r} \\
W_{\mathrm{out}}^{\mathrm{II}}(\mathbf{r}) &= \mathbf{H}_{ext} \cdot \mathbf{r} \left(\frac{R^3}{r^3} \frac{1 - \mu_r}{1 + 2\mu_r} - 1 \right)
\end{aligned}$$

and direct summation yields the solution of the original problem:

$$W(\mathbf{r}) = \begin{cases} \mathbf{m}\cdot\mathbf{r}\ \frac{1}{4\pi}\left(\frac{1}{r^3} + \frac{1}{R^3}\frac{1-\mu_r}{1+2\mu_r}\right) - \mathbf{H}_{ext}\cdot\mathbf{r}\ \frac{3\mu_r}{1+2\mu_r} & r < R \\ \mathbf{m}\cdot\mathbf{r}\ \frac{1}{4\pi r^3}\ \frac{3}{1+2\mu_r} + \mathbf{H}_{ext}\cdot\mathbf{r}\left(\frac{R^3}{r^3}\frac{1-\mu_r}{1+2\mu_r} - 1\right) & r > R \end{cases} \tag{5.18}$$

We can now identify the so-called reaction field that has a finite value at $r = 0$:

$$\mathbf{H}_{RF} = -\left(\mathbf{m}\ \frac{1}{2\pi R^3}\frac{1-\mu_r}{1+2\mu_r} - \mathbf{H}_{ext}\ \frac{3\mu_r}{1+2\mu_r}\right) \tag{5.19}$$

Introducing μ_r-depending constants

$$\gamma_{\mathbf{m}} := \frac{1-\mu_r}{1+2\mu_r} \qquad \gamma_{\mathbf{h}} := -\frac{3\mu_r}{1+2\mu_r} \tag{5.20}$$

our estimate for the energy of one superspin $\mathbf{s}$ due to the external field and the dipolar environment is therefore:

$$E_{RF} = -\ \mathbf{s}\cdot\mathbf{B}_{RF} = -\mu_0\mathbf{s}\cdot\mathbf{H}_{RF}$$
$$E_{RF} = \mathbf{s}\cdot\left(\mathbf{m}\ \frac{\mu_0\ \gamma_{\mathbf{m}}}{2\pi R^3} + \mathbf{H}_{ext}\ \mu_0\gamma_{\mathbf{h}}\right)$$

$$E_{RF} = \mathbf{s}\cdot\left(\mathbf{m}\ \frac{\mu_0}{2\pi R^3}\frac{1-\mu_r}{1+2\mu_r} - \mathbf{B}_{ext}\ \frac{3\mu_r}{1+2\mu_r}\right) \tag{5.21}$$

Both the $\mathbf{m}$- and $\mathbf{B}_{ext}$ term have properties that should be discussed further.
We calculate μ_r via

$$\mu_r = 1 + \chi_m = 1 + \frac{M}{H_{ext}} = 1 + \frac{\mu_0\ M}{B_{ext}} \tag{5.22}$$

where $M = |\mathbf{M}|$ is the magnetization, i.e. the total magnetic dipole density which is assumed to be linearly dependent of $B_{ext} = |\mathbf{B}_{ext}|$ throughout our approximation.

5.1.6.3 Modelling parameters

As a general rule, the reaction field encourages configurations within the cut-off sphere with large cumulative magnetic moment via a parallel auxiliary field because

$$-1/2 < \frac{1-\mu_r}{1+2\mu_r} < 0 \qquad \mu_r > 1 \tag{5.23}$$

This term therefore indeed deserves the name 'reaction field' because this auxiliary field is the response of the imagined medium towards any net magnetic moment in the sphere.
The second term means that applied magnetic field acting on any single magnetic

moment is amplified because of the surrounding medium since

$$-3/2 < \frac{-3\mu_r}{1+2\mu_r} < -1 \qquad \mu_r > 1 \tag{5.24}$$

At first look, this seems like the Onsager approach leads to inconsistencies in the limit of non-interacting systems. This is ultimately not the case, as shown in 5.1.6.6.

5.1.6.4 Cut-off Radius

The radius R largely determines the computational effort that an energy estimate with this approximation still requires. Intuitively one would expect that a smaller cutoff radius implies an increase of the necessary correction, which is indeed the case. The fact that the correction is inversly proportional to the volume of the cut-off sphere means that it is also inversely proportional to the number of superspins that are treated exactly within our energy-estimation scheme which seems reasonable.

Because we are always dealing with a discrete dipole distribution, we must however specify how R is implemented. There will always be an interval of possible R values whose corresponding cut-off spheres would contain the same magnetic moments. Without additional criteria, R could thus be chosen arbitrary which would however limit comparability between simulations.

We found that a consistent and reasonable estimate would be choosing *the greatest possible R that does not span additional sites outside the cut-off sphere.* This implies that no internal site touches the boundary of the sphere as seen in figure 32, and the reaction field takes on the lowest possible absolute value In any case, R has to

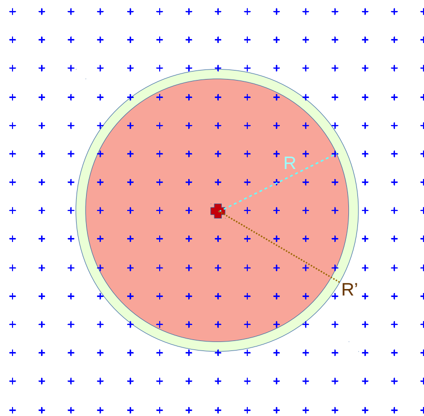


Figure 32: Choice of R for reaction field estimation. Blue crosses signify magnetic sites. Both the cyan and salmon colored circles contain the exact same sites. They represent the smallest and biggest possible choices for R

be chosen consistently in order to have comparable simulation results since e.g. a 10% increase of R already means that the **m**-factor is cut in half.
Simulations were mostly done on fcc-super lattices which have a very high degree of spatial symmetry so that the computation of the best choice for R needs to be carried out only once for any choice of 'modelling' cut-off.

5.1.6.5 Determination of the relative permeability

The recipe (5.22) given above has the advantage of being a self-consistent procedure because each magnetic moment will be updated regularly during the simulation of one measurement point and all under the same conditions. This means that the chosen μ_r indeed becomes the best choice self-fulfillingly.
This however holds only true if the global susceptibility remains roughly constant throughout these measurements. This is obviously not true e.g. during the process of a ZFC-FC curve where the applied magnetic field has only one value, yet the magnetization is dependent on the recent history and temperature of the sample.
The hitherto implemented scheme thus suffers at least from temporal retardation, which becomes an issue if the goal of the simulation is to pin down an exact value for the blocking temperature.
Even worse, in the absence of applied fields the recipe (5.22) becomes ill-defined and one has to either retroactively impose values from e.g. high-temperature values in the ZFC-curve or impose a theoretical expectation like the Langevin value for paramagnets

$$\mu_r^{\text{Langevin}} = 1 + \chi^{\text{Langevin}} = 1 + n\mu_0 \frac{(M_s V)^2}{3k_B T}$$

which we only used as a very first guess. In subsequent simulations, the value was indeed changed to be the value found for the tail-end of ZFC curves which let our μ_r converge towards $\mu_r \sim 15$ in zero-field.

The exact value of μ_r would of course be an interesting property/result in itself, however its influence of our reaction field is limited. The two prefactors in (5.20) are shown in figure 33.

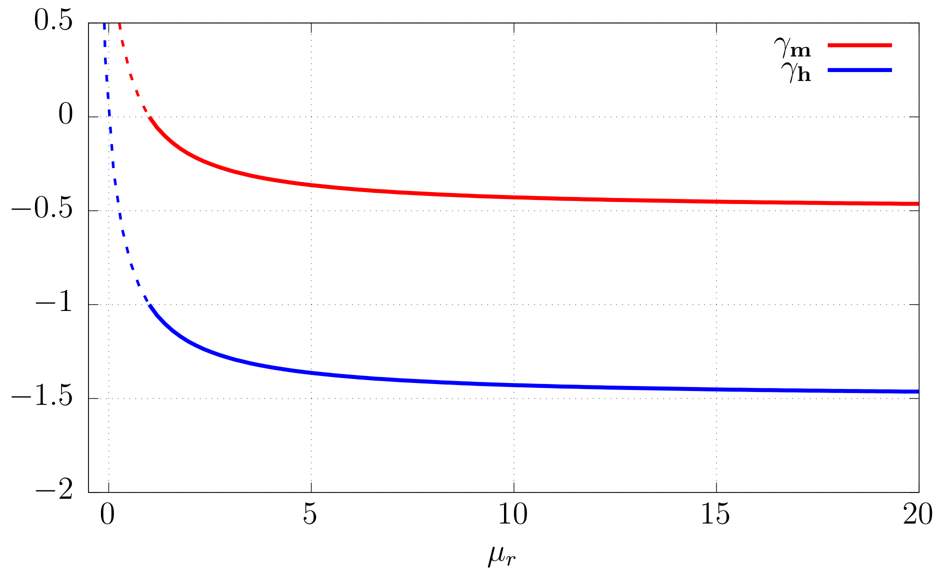


Figure 33: Dependence of correction factors on relative permeability. Only $\mu_r > 1$ (solid) applies in our simulations

The prefactors vary only very slightly for $\mu_r \gtrsim 10$, and the applied-field correction term does not even apply in zero-field. Therefore, we can at least have confidence that the precise value of our assumed μ_r is of less importance than e.g. the validity of the assumptions regarding the material in (5.13)

5.1.6.6 Consistency with non-interacting limit

It is expected that the influence of the dipole-dipole interaction on the sample diminishes quickly with an increase of the lattice constant of the supercrystal a_0.
This will be reflected automatically for the interactions that are considered in an exact matter within V_c. The **m**-factor in the reaction-field term also reflects an increase of the lattice constant automatically because R per design scales automatically with any change of the lattice constant.
The $\mathbf{B}_{ext}$-factor does not scale directly with the spatial dimensions of the sample. The scale of the system is instead only influential via μ_r because the magnetization M in (5.22) is the dipole moment *density* which means that if only the lattice constant increases without adding further magnetic dipoles, we have for μ_r, $\gamma_\mathbf{m}$, $\gamma_\mathbf{h}$

$$\left.\begin{aligned} &\mu_r && \to 1 \\ &\gamma_\mathbf{m} = \tfrac{1-\mu_r}{1+2\mu_r} && \to 0 \\ &\gamma_\mathbf{h} = \tfrac{3\mu_r}{1+2\mu_r} && \to 1 \end{aligned}\right\} \quad \text{for} \quad a_0 \to \infty$$

This means that the **m**-term vanishes even faster than $\propto 1/R^3$.
$\gamma_\mathbf{h} \to 1$ means that there is no amplification via the magnetic medium anymore, which is also to be expected.
In total, we arrive for large lattice constants at a vanishing reaction field so that

only the superspins within V_c will interact via dipole-dipole interaction and the unaltered externally applied field. As shown in figure 34, we get a smooth transition of the ZFC/FC curves towards the non-interacting case as soon as $\frac{a}{a_0} \sim 5$ where a_0 is chosen such that for $a \leq a_0$ the behavior is significantly different due to 2-particle interactions: $a_0 \simeq 25$ nm.
This means that the behavior of our approximative theory is largely correct in the non-interacting limit.

To understand what happens at $a = a_0$, we calculate the dipole-dipole energy of one spin in the environment of ≥ 100 randomly oriented neighbouring spins. This component of the total potential that is independent from all anisotroy axes is given in figure 35.

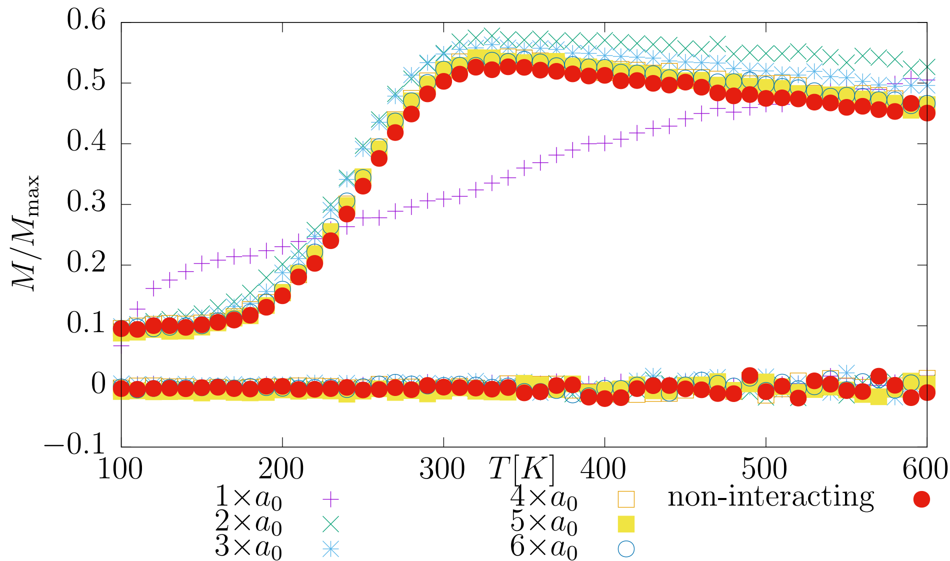


Figure 34: Convergence of ORF simulations of the ZFC curve towards the non-interacting limit

The relevant information of this is that there is always exactly one global minimum produced by the dipole fields. Even local minima are only rarely observed. If one adds this potential to the known Stoner-Wohlfarth results in 4.1.3, one observes in most cases that additionally to a change in position of the entire potential landscape, that the potential wall between the two local minima increases. Since the blocking temperature is predominantly associated with this potential wall, one can already assume that the dipole-dipole interaction will increase the blocking temperature.

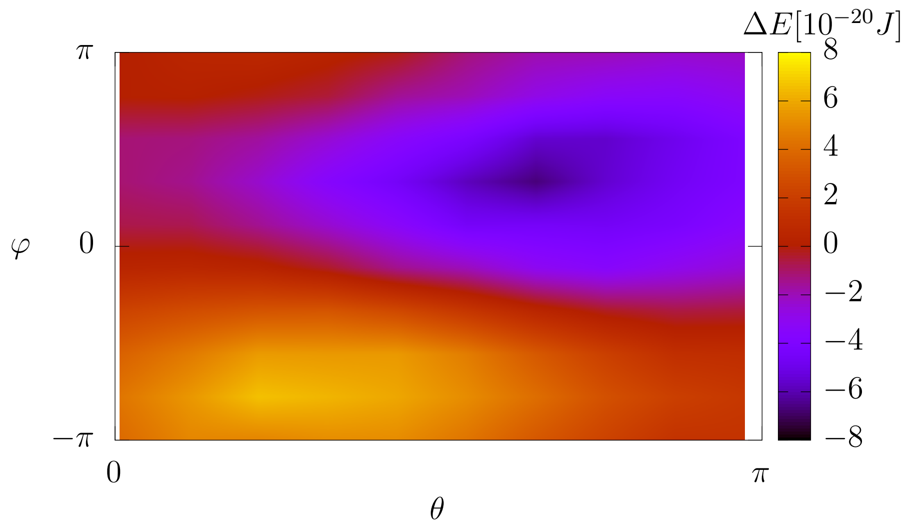


Figure 35: Potential landscape for a superspin in a 8x8x8 magnetic supercell (fcc). Plotted are energy differences compared to the energy for orientation $\theta = \pi/2$, $\varphi = 0$. Any cut-off of more than 30 contributing neighbours leads to an image that is indistinguishable at this resolution

5.2 Simulation Results

We now present simulation results obtained by studying systems with interacting nanoparticles.

5.2.1 Groundstates in Periodic, Dipolar 3D Systems

We discuss a topic that is motivated more from pure statistical physics point of view, than in the context of self-assembled magnetic nanoparticles. We consider 3D crystal systems whose total energy is exclusively determined by the magnetic dipole-dipole interaction of magnetic moments positioned on the lattice sites. The aim is an investigation of a purely dipolar ground state in certain lattice structures.

5.2.1.1 Results in the Limit of Vanishing Magnetocrystalline Anisotropy

We realize that the question above is equivalent to the physically not viable case of self-assembled magnetic nanoparticles in the limit of vanishing magnetocrystalline anisotropy at each lattice site. Simulations can be therefore carried out in a straightforward fashion with our existing methods. We consider systems that are initiated at very high temperatures and then cooled down with zero applied field. We pay particular attention to the energy of the system and discuss the superspin landscape at $T \to 0$.

With the help of the self-developed program which is documented in B.2, we will produce figures like figure 36. In all of these figures, all spins are grouped into

sublattices that in most cases span the magnetic super cell and require all of their member sites to be parallel aligned within a certain margin of error.

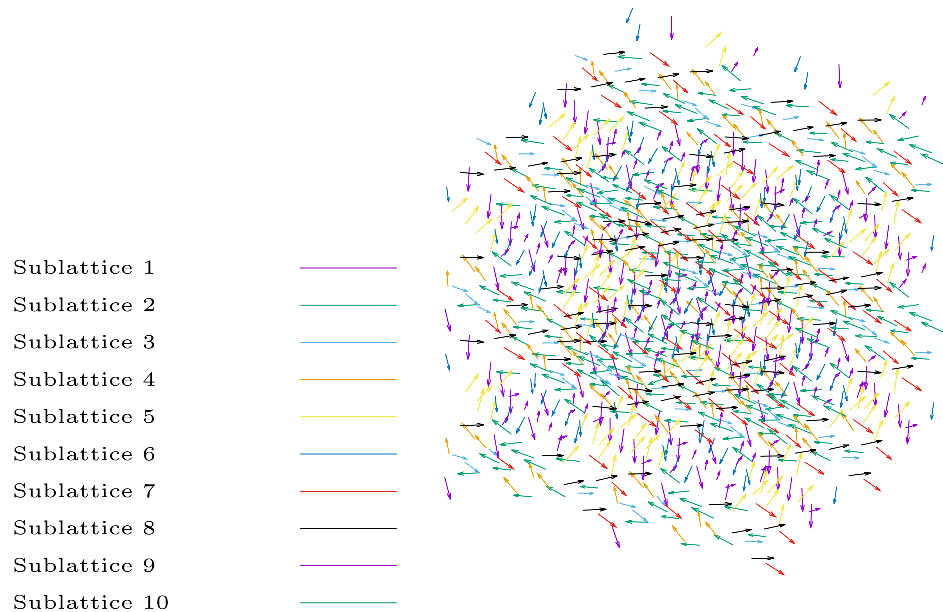


Figure 36: 8x8x8 magnetic super cell (*fcc*) at $T = 1$. Spin distribution of one start configuration

In this thesis, I mostly show instances where these sublattices have the peculiar property of being essentially 2-dimensional so that one can find a perspective that makes the spin structure readable by a projection on a plane, as seen in figure 37.

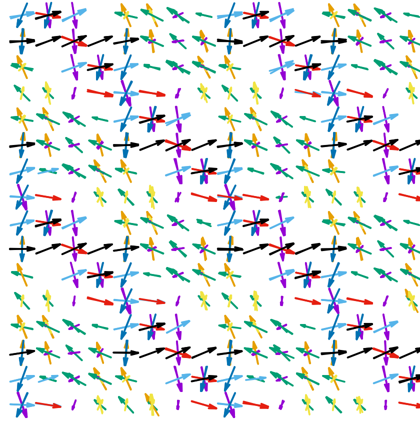


Figure 37: Rotated view of figure 36

5.2.1.2 Series of 'Groundstates' Depending on Different Parameters in the Onsager Approximation

We present several low energy states and their spin structure that were obtained with varying spin initializations. Although different cut-off radii were employed to further increase the variety of simulated configurations, the following selection of low-energy structures were found independently from these respective choices.

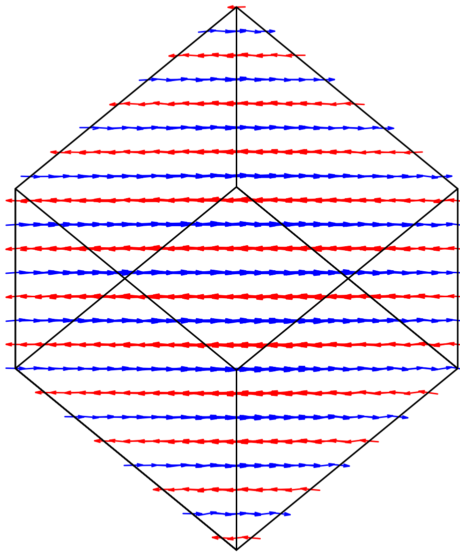


Figure 38: 2 sublattices, inspecific view

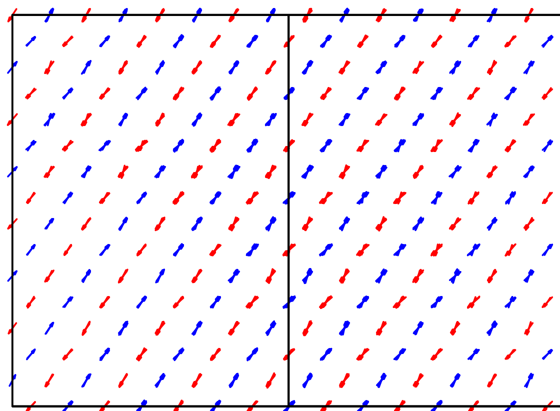


Figure 39: Rotated view of figure 38

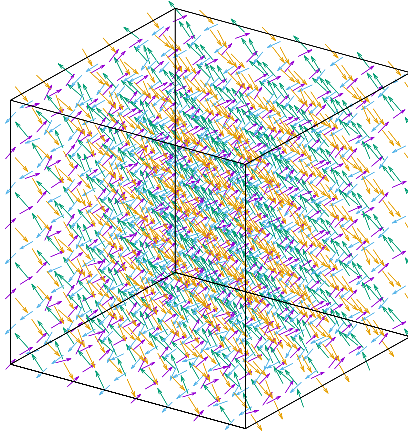


Figure 40: 4 sublattices, inspecific view

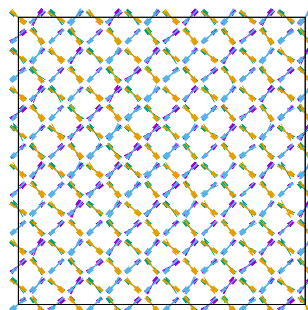


Figure 41: Rotated view of figure 40

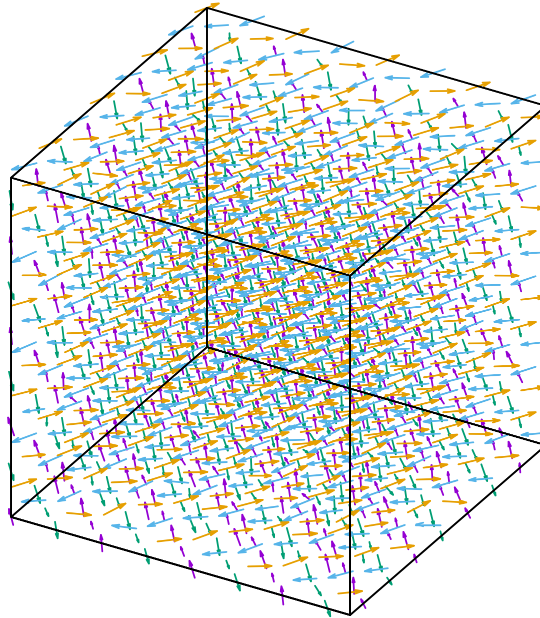


Figure 42: 4 sublattices, inspecific view

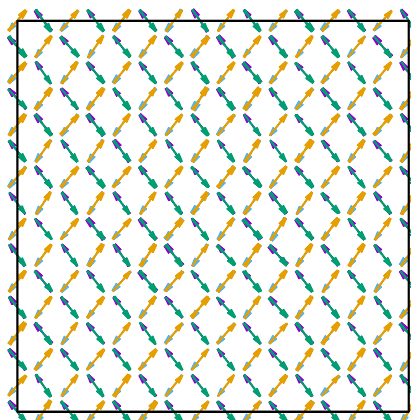


Figure 43: Rotated view of figure 42

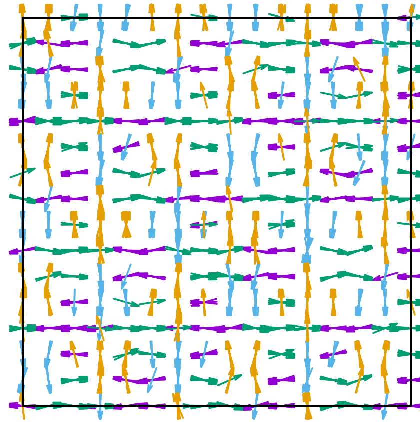


Figure 44: 4-6 sublattices as a more complex configuration, rotated view

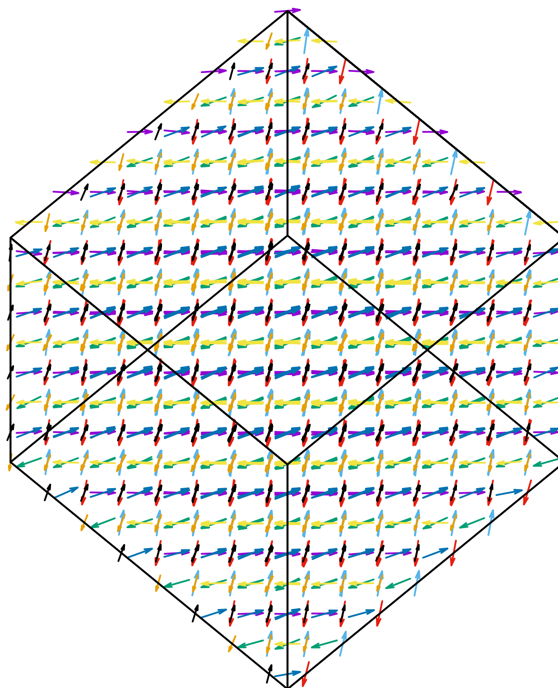


Figure 45: 8 sublattices, inspecific view

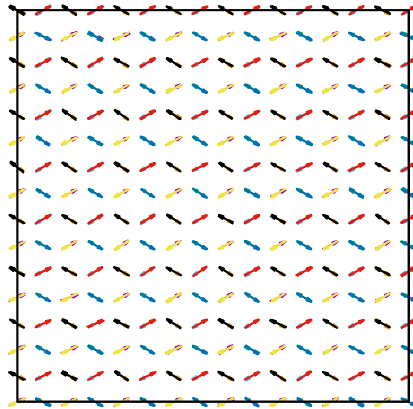


Figure 46: Rotated view of figure 45, yz-plane

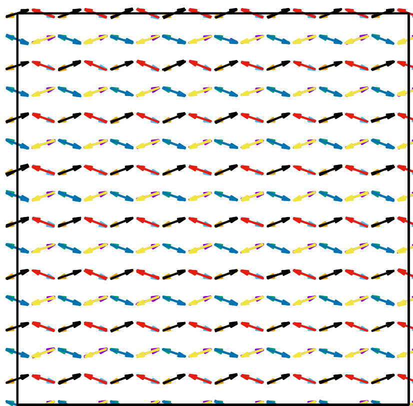


Figure 47: Rotated view of figure 45, xz-plane

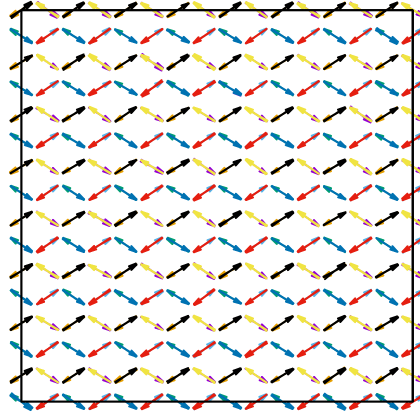


Figure 48: Rotated view of figure 45, xy-plane

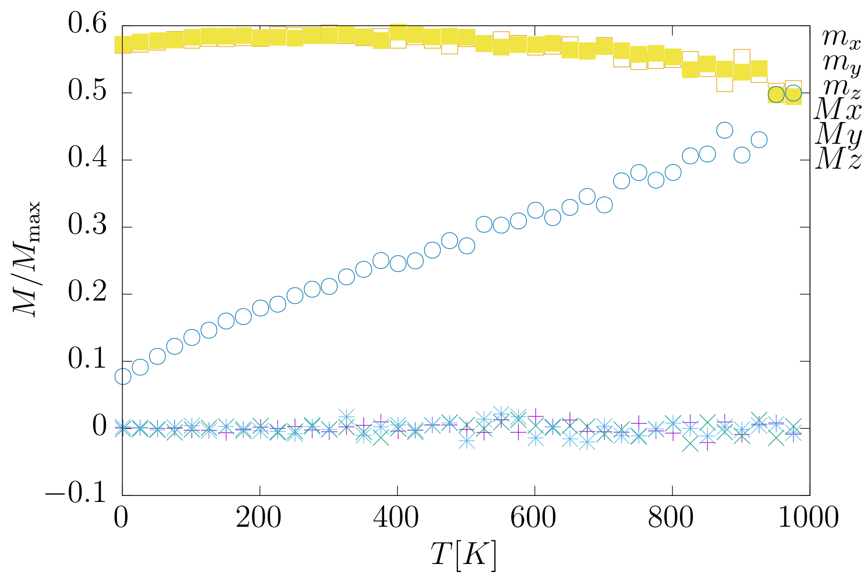


Figure 49: Temperature dependence of the magnetization components for the sample as shown in both figures 42 and 43. Mx, My, Mz are defined in (5.25). By these 'components', the 2-dimensional character of the low-energy state corresponds to the vanishing of 'MZ'

It is useful to define the following quantities:

$$M\alpha := \sum_{i=1}^{N} |m_\alpha^i| \qquad \alpha = x, y, z \tag{5.25}$$

This means that e.g. $Mx = 0$ if the x-component of the magnetization of each site is zero and *not* just randomly distributed in $(-1, 1)$. Therefore, Mx measures if the x-component of the magnetization vanishes completely, and the system therefore does not have a 3-dimensional, but a layered 2-dimensional magnetization landscape.
Apart from this often occurring reduction of dimensionality, it has always been observed that an even number of antiparallel sublattices emerges at low temperatures. However, we cannot call this behaviour antiferromagnetic because no spontaneous symmetry breaking is observed. Conversely, this 'phase transition' happens steadily but slowly in an arbitrarily large temperature region.

5.2.1.3 Reconsidering Non-Vanishing Anisotropy

We connect the previous chapter back to the more realistic setting of magnetic nanoparticles such as maghemite nanoparticles. We discuss how stable the findings of chapter 5.2.1.2 are with regards to non-zero anisotropy energies.

In short, employing realistic parameters of the maghemite nanoparticles in order to study the number and character of sublattices at low temperatures yields a clear but disappointing result: No long range order can be detected, meaning that the random anisotropy changes the local potential landscapes too strongly.

On a more positive note, we can therefore study ZFC curves quite well with the help of the ORF method because it is even more viable if there is no (short-range) magnetic order at the temperatures we consider. Simulation results are given in figures 50 and 51. Because of the more complicated potential landscape, it is necessary to make the simulations with a smaller test-vector length than in the non-interacting case. Otherwise, local potential minima maybe insufficiently scanned by our algorithm and we would find non-physical dynamics.
Both simulations are performed with $d_m = 0.1 \ll 1$, but the first plot shows that one must then also increase the number of Monte-Carlo steps per measurement: It has only been increased to 2000 from 1000 in the non-interacting examples earlier. The different shape and unrealistically high blocking temperature show that this simulation does not represent nature, and one has to increase the number of MCS even further. Of course, the real computation time also increases linearly.

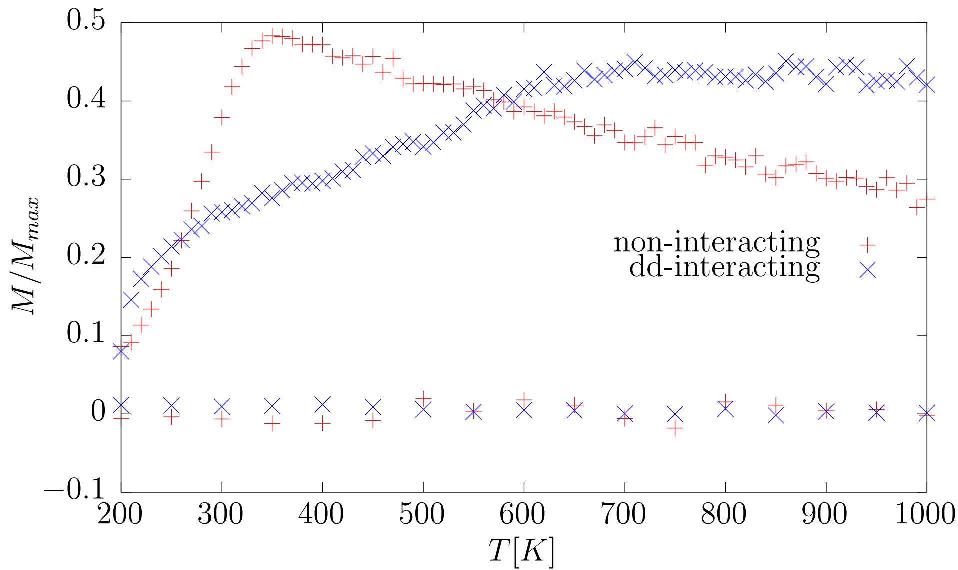


Figure 50: Simulated ZFC curve for maghemite nanoparticles at $\mu_0 H = 8mT$ with and without dipole-dipole interaction. The observed difference in T_B is highly exaggerated, as is the difference in shape. This is because the number of MCS is chosen too low and the blue curve does not represent a system that has found its thermal equilibrium at any point

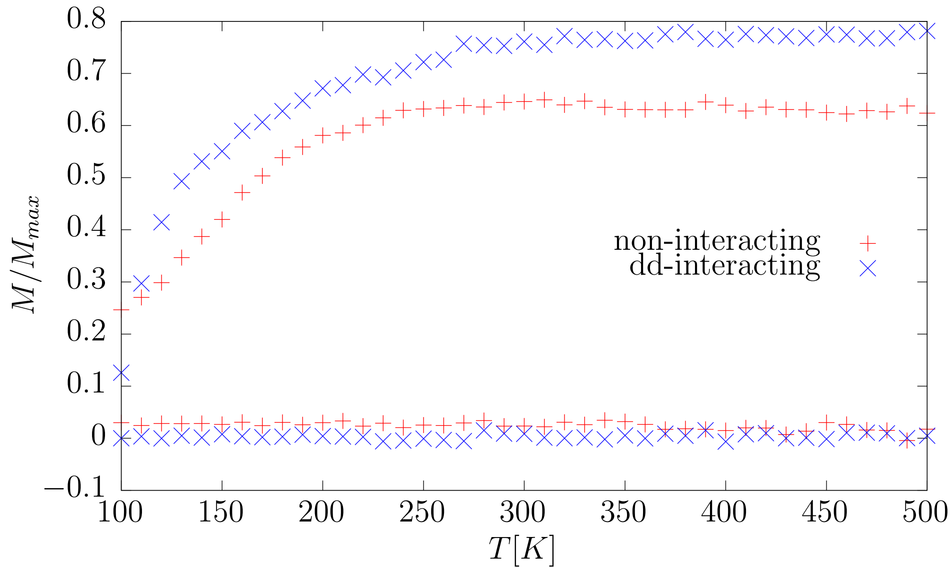


Figure 51: Simulated ZFC curve for maghemite nanoparticles at $\mu_0 H = 20mT$ with and without dipole-dipole interaction. In contrast to figure 50, the number of MCS per temperature step is increased by a factor of 10 with unaltered $d_m = 0.1$. The result is in far better accordance with experiments because the shape is better preserved and the difference in T_B is only at about $100K$

6 Summary and Outlook

From the study of interacting particles, we have learned about the significance of simulation parameters like the test-vector length. The introduction of the Onsager Reaction Field method has been shown to be a valid approximation that can be used for magnetometric simulations like ZFC/FC curves in order to drastically reduce computational effort.

The presented research on purely dipolar systems in *fcc*-geometry has shown peculiar magnetic behavior that is very much distinct from magnetic phenomena associated with short-range exchange interactions. Our research was very much restricted only towards pure bulk behaviour. Certainly surface effects are extremely important in conjunction with dipole-dipole interaction and need to be addressed in a next step. Also it appears quite likely that the magnetocrystalline anisotropy has a large impact when studying systems with long-range magnetic order. Therefore, the limit of vanishing anisotropy is not a satisfactory basis for understanding realistic behaviour of assemblies of nanoparticles.

Because our current algorithm and implementation does allow for any choice in lattice constant, saturation magnetization, anisotropy distribution etc, future work should be done by studying larger regions of parameter space.

A Detailed Calculations

A.1 Magnetostatic Derivation of the Onsager Reaction Field

The ORF is a consequence of the magnetic Poisson equation that was derived, including suitable boundary conditions, in 5.1.6.2. Our 'magnetic sources' $\nabla \cdot \mathbf{M}$ are zero almost everywhere:

- Except for the magnetic dipole in the center, the inside of our cut-off sphere $\{\mathbf{r} \in \mathbb{R}^3 \,|\, 0 < r < R\}$ is modeled as vacuum, i.e. $\mathbf{M} = 0$.
- The outside $\{\mathbf{r} \in \mathbb{R}^3 \,|\, r > R\}$ is homogenuously magnetized, i.e. $\nabla \cdot \mathbf{M} = 0$.

Therefore, it is more elegant to use the general solution of the homogenuous Laplace equation which is easier to find, and account for everything else by finding special solutions at $r = 0$ and $r = R$ via the boundary conditions. Since the solution to any boundary value problem as stated is unique, any solution we find, will automatically be the correct solution.

A.1.1 Laplace Equation in Azimuthal Symmetry

We are interested in the solution to the homogenous *Laplace equation* in spherical coordinates:

$$\Delta W(\mathbf{r}) = 0; \qquad W(\mathbf{r}) = W(r, \theta, \varphi)$$

$$\frac{1}{r}\frac{\partial^2}{\partial r^2}(rW) + \frac{1}{r^2 \sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial W}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2 W}{\partial\varphi^2} = 0 \tag{A.1}$$

We will immediately ignore the φ-part because we can always reduce our systems of interest to such with azimuthal symmetry. Furthermore, we make a separation ansatz:

$$W(r, \theta, \varphi) = W(r, \theta) = \frac{u(r)}{r}\, P(\theta)$$

Plugging this into (A.1) and multiplying by $r^2/(uP)$ yields

$$\underbrace{\frac{r^2}{u}\frac{d^2u}{dr^2}}_{\text{independent from } \theta} + \underbrace{\frac{1}{P\sin\theta}\frac{d}{d\theta}\left(\sin\theta\frac{dP}{d\theta}\right)}_{\text{independent from } r} = 0 \tag{A.2}$$

Therefore, both summands have to be simultaneously constant and opposite in sign. We therefore can define

$$\frac{r^2}{u}\frac{d^2u}{dr^2} =: \lambda \in \mathbb{R} \qquad \Rightarrow \qquad \frac{d^2u}{dr^2} - \frac{\lambda}{r^2}u = 0 \tag{A.3}$$

and

$$x := \cos\theta \qquad \Rightarrow \qquad \frac{d}{dx}\left(\bullet\right) = -\frac{1}{\sin\theta}\frac{d}{d\theta}\left(\bullet\right) \tag{A.4}$$

With these definitions we obtain from the second summand in (A.2) the ordinary differential equation

$$\frac{d}{dx}\left[(1-x^2)\frac{dP}{dx}\right] + \lambda P = 0$$
$$(1-x^2)\frac{d^2P}{dx^2} - 2x\frac{dP}{dx} + \lambda P = 0 \qquad -1 \leq x \leq 1 \tag{A.5}$$

At this point, we must stress that the solutions need to cover the complete domain $x \in [-1,1]$. We now define a function $w_n(x)$

$$w_n(x) := -(x^2-1)^n \tag{A.6}$$
$$\Rightarrow \qquad (1-x^2)w' + 2nxw = 0$$
$$\text{Differentiate n+1 times} \Rightarrow \qquad (1-x^2)w_n^{(2+n)} - 2xw_n^{(n+1)} + n(n+1)w_n^{(n)} = 0 \tag{A.7}$$

This means that we have found functions w_n' that solve our 2nd order ordinary differential equation (A.5) including the necessary condition that it is defined $\forall x \in [-1,1]$ These functions are therefore the only possible and unique solutions and we found that λ is not an arbitrary real number, but has to have the form

$$\lambda = n(n+1)$$

so that our differential equation is the *Legendre Differential Equation*

$$\frac{d}{dz}\left[(1-z^2)\frac{d}{dz}P_n(z)\right] + n(n+1)P_n(z) = 0$$

The so-called *Rodrigo form*

$$P_n := w_n' = \frac{d^n}{dx^n}(x^2-1)^n$$

that we recovered up to a normalization factor produces the *Legendre Polynomials* $P_n(x)$

$$P_0(x) = 1$$
$$P_1(x) = x$$
$$P_2(x) = \frac{1}{2}(3x^2-1)$$
$$P_3(x) = \frac{1}{2}(5x^3-3x)$$
$$\vdots$$

Since we have also found what λ is, we can now investigate (A.3):

$$\frac{d^2u}{dr^2} - \frac{n(n+1)}{r^2}u = 0;$$

The general solution to this for any given $n \in \mathbb{N}_0$ is

$$u(r) \equiv u_n(r) = a_n r^n + \frac{b_n}{r^{n+1}} \tag{A.8}$$

Together, the general solution of the Laplace equation for problems with azimuthal symmetry is

$$\boxed{W(\mathbf{r}) = W(r,\theta) = \sum_{l=0}^{\infty} \left(a_l r^l + \frac{b_l}{r^{l+1}} \right) P_l(\cos\theta)} \tag{A.9}$$

$$W(r) \text{ finite for } r = 0 \Rightarrow b_l = 0 \quad \forall l \geq 0$$
$$W(r) \to 0 \text{ for } r \to \infty \Rightarrow a_l = 0 \quad \forall l \geq 0$$

$$W(r) = A(r)\ \mathbf{a}\cdot\mathbf{r} \Rightarrow \nabla W(r) = \frac{d}{dr}A(r)\ \hat{\mathbf{r}}\ (\mathbf{a}\cdot\mathbf{r}) + \mathbf{a}\ A(r)$$

A.1.2 Solution to Our Boundary Value Problem

We restate the two distinct boundary value problems from section 5.1.6.2 with azimuthal symmetry that we want to solve with the above.

$$\Delta W^{\mathrm{I/II}} = \nabla\cdot\mathbf{M}^{\mathrm{I/II}}$$

$$\begin{aligned}
W^{\mathrm{I}}(r\to 0) &\to \frac{\mathbf{m}\cdot\mathbf{r}}{4\pi r^3}\\
W^{\mathrm{I}}(r\to\infty) &\to 0\\
W^{\mathrm{I}}(r=R) &= \text{continuous}\\
\mathbf{H}_{in}^{\mathrm{I},\perp} &= \mu_r \mathbf{H}_{out}^{\mathrm{I},\perp}\\
\mathbf{H}_{in}^{\mathrm{I},\parallel} &= \mathbf{H}_{out}^{\mathrm{I},\parallel}
\end{aligned}$$

$$\begin{aligned}
W^{\mathrm{II}}(r\to 0) &\to 0\\
W^{\mathrm{II}}(r\to\infty) &\to -\mathbf{H}_{ext}\cdot\mathbf{r}\\
W^{\mathrm{II}}(r=R) &= \text{continuous}\\
\mathbf{H}_{in}^{\mathrm{II},\perp} &= \mu_r \mathbf{H}_{out}^{\mathrm{II},\perp}\\
\mathbf{H}_{in}^{\mathrm{II},\parallel} &= \mathbf{H}_{out}^{\mathrm{II},\parallel}
\end{aligned}$$

Because the Legendre polynomials form an orthonormal set, one only needs to look at the powers or $\cos\theta$ that show up in the equations. This simplifies the problem to the following set of equations.

$$W^{\mathrm{I}}_{\mathrm{in}}(\mathbf{r}) = \alpha\,\frac{\cos\theta}{r^2} + \beta\,r\,\cos\theta$$
$$W^{\mathrm{I}}_{\mathrm{out}}(\mathbf{r}) = \gamma\,\frac{\cos\theta}{r^2} + \delta\,r\,\cos\theta$$

with $\alpha, \beta, \gamma, \delta \in \mathbb{R}$ to be determined from the boundary conditions.

B Programming

B.1 C++ implementation of Metropolis algorithm

Here we give the C++ implementation that has given all simulation results that were presented in this thesis, except for the dynamic susceptibility measurements. To achieve this versatility of one programme, the user must set logical parameters declaring what part of the programme will be relevant when executed. Also, the user can still declare things when running the executable. It is written in the C++11 standard.
The basic structure of this implementation is taken from an older C programme with permission by Oleg Petracic. Both the transition to the more modern C++ realization and the treatment of 2-particle interactions, especially within ORF were written by the author.

B.1.1 main.cpp

```
/*
MonteCarlo.cpp - Main file

M vs T / M vs B - simulation

!!!!
This programme is written in C++11 standard and utilizes openMP for easily implemented multithreaded computations
!!!!

The following is based on work leading to MnO Bulk, see its history below

Modification: 23/01/03, 24/01/03: main loop structure
Modification: 27/02/03: show+save Dip.- and Anis.-Energies -> MCS_one()
Modification: Feb/2009: rename to 5, with and without periodic boundary cond.
Modification: Nov/2012: MnO Bulk
START OF HEAVY MODIFICATION/REDUCTION since Nov/2016
Modification: Nov/2016: Transition to C++, intended to be executed on Linux -> e.g. substituting Visual C elements like '_s' functions.
Modification: Dec/2016: Heavy reduction leading to Heisenberg Paramagnetic simulation. Spin data as output for visualization in gnuplot or similar. Necessary dipole-dipole functions added
Modification: Jan/2017: sc lattice generator for easiest super-crystal structure. Optional cut-off implementation to speed up computation.
Modification: Feb/2017: include anisotropy energy. implement random distribution easy axes at every site
Modification: Sep/2017: Inclusion of perAux.h and introduction of ORF method for speedy simulations of interacting, periodic systems

*/

#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <array>
#include <string>
#include <algorithm>
#include <thread>

#include <stdlib.h>
#include <omp.h>

#include "parameter.h" // parameter file
#include "rnd250.c" // randum number generation
#include "random_spd5.h" // rng addition
#include "input_ini.h" // necessary programming stuff like arrays, file save
#include "perAux.h" // contains functions for periodic boundary conditions and ORF method
#include "ORF0_str.h" // functions with physical meaning

using namespace std;
```

```
45
46  int main()
47  {
48          cout << "NN estimate: " << NN << endl;
49          cout << "Include d-d interaction? ";
50          cin >> DIPOLAR;
51          int edition;
52          cout << "Edition: ";
53          cin >> edition;
54          cout << endl;
55          latt_const = latt_const_0;
56
57          int xx, StepNr, lx, lm, c;
58          xx = 1;
59          xxmax = 1;
60
61          reset_files(edition);
62
63          if (PERIODIC)
64          {
65                  if (type == "sc")
66                  {
67                          sc_generator_new(edge_N);
68                          max_N = pow(edge_N,3);
69                  }
70                  if (type == "bcc")
71                  {
72                          bcc_generator_new(edge_N);
73                          max_N = 2*pow(edge_N,3);
74                  }
75                  if (type == "fcc")
76                  {
77                          fcc_generator_new(edge_N);
78                          max_N = 4*pow(edge_N,3);
79                  }
80
81                  if (type == "hcp")
82                  {
83                          hcp_generator_new(edge_N);
84                          max_N = 1*pow(edge_N,3);
85                  }
86
87                  makePeriodic();
88
89          }
90          else
91          {
92                  if (type == "sc")
93                          sc_generator_new(edge_N);
94                  if (type == "bcc")
95                          bcc_generator_new(edge_N);
96                  if (type == "fcc")
97                          fcc_generator_new(edge_N);
98                  if (type == "hcp")
99                          hcp_generator_new(edge_N);
100                 if (SPHERICAL)
101                 {
102                         max_N = det_spheric();
103                         new_pos();
104                 }
105                 else
106                 {
107                         max_N = max_N_temp;
108                         cout << max_N << " in cube.\n";
109                 }
110         }
111
112
113         InitMain(edition);
114
115         auto t1 = chrono::high_resolution_clock::now();
116
117         vector<double> comp_t;
118         comp_t.push_back(0.);
119         double Et = 0.;
120
121         // ************* Scans **************
122
123         for (int cfg = 0; cfg < CFG; cfg++)  // config loop
124         {
125
126                 InitArray();
```

```
127                 anisosave(edition);
128                 aniso_test();
129 
130                 cout << "start T sweep\n";
131                 // ***************** T-Scan ******************
132                 T[xx] = Tpt[0];
133                 Bx[xx] = Bxfix[0];
134                 //BBx[xx] = Cz * Bx[xx] // OFF In para0 not necessary
135 
136                 for (c = 0; c < MR; c++) // sweep counter loop
137                 {
138 
139 
140                         // one single temperature step at the beginning
141                         if (Tsteps[c] > 0)
142                         {
143                                 T[xx] = Tpt[c];
144                                 Bx[xx] = Bxfix[c];
145                                 array<double, 2> arr = rel_perm(Bx[xx]);
146                                 double mu_r = arr[0];
147                                 double M = arr[1];
148                                 for (lx = 0; lx < TRlxLoops[c]; lx++) // Relaxation loop
149                                 {
150                                         MCS_one(xx, Bx[xx], KB*T[xx], mu_r, M);
151                                 }
152 
153                                 for (lm = 0; lm < TAvgLoops[c]; lm++) // Averaging loop
154                                 {
155                                         MCS_one(xx, Bx[xx], KB*T[xx], mu_r, M);
156                                         Mx[xx] += measure();
157                                 }
158 
159                                 Mx[xx] /= ((double) TAvgLoops[c]);
160 
161 
162                                 auto t2 = chrono::high_resolution_clock::now();
163 
164                                 chrono::duration<double, milli> fp_ms = t2-t1;
165                                 comp_t.push_back(fp_ms.count());
166 
167                                 spinsave(edition);       // NOT advised here! GB files!
168                                 filesave(edition, xx);
169                                 double Et = E_filesave(edition, xx);
170                                 detailed_measure(edition, xx);
171 
172                                 cout << "cfg: " << cfg + 1 << ", T = " << T[xx] << ", B = " <<
                                        Bx[xx] << ", Mx = " << Mx[xx] << ", Etot = " << Et << " in "
                                        ;
173                                 cout << (comp_t[xx]-comp_t[xx-1])/1000 << " [s]\n";
174 
175 
176                                 cout << "rel Permeability: " << mu_r;
177                                 if (mu_r < 1)
178                                 {
179                                         cout << " -> physical?!";
180                                         //abort();
181                                 }
182                                 cout << endl;
183 
184 
185 
186                                 xx++;
187                                 xxmax++;
188 
189                                 T[xx] = T[xx-1] + dT[c];
190 
191                         }
192 
193 
194                         for (StepNr = 0; StepNr < Tsteps[c]; StepNr++)
195                         {
196                                 Bx[xx] = Bxfix[c];
197                                 array<double, 2> arr = rel_perm(Bx[xx]);
198                                 double mu_r = arr[0];
199                                 double M = arr[1];
200                                 for (lx = 0; lx < TRlxLoops[c]; lx++) // Relaxation loop
201                                 {
202                                         MCS_one(xx, Bx[xx], KB*T[xx], mu_r, M);
203                                 }
204 
205 
206                                 for (lm = 0; lm < TAvgLoops[c]; lm++) // Averaging loop
```

```
207                         {
208                                 MCS_one(xx, Bx[xx], KB*T[xx], mu_r, M);
209                                 Mx[xx] += measure();
210                         }
211 
212                         Mx[xx] /= ((double) TAvgLoops[c]);
213 
214 
215                         auto t2 = chrono::high_resolution_clock::now();
216 
217                         chrono::duration<double, milli> fp_ms = t2-t1;
218                         comp_t.push_back(fp_ms.count());
219 
220                         spinsave(edition);      // NOT ADVISED: GB files!
221                         filesave(edition, xx);
222                         Et = E_filesave(edition, xx);
223                         detailed_measure(edition, xx);
224 
225                         cout << "cfg: " << cfg + 1 << ", T = " << T[xx] << ", B = " <<
                                Bx[xx] << ", Mx = " << Mx[xx] << ", Etot = " << Et << " in "
                                ;
226                         cout << (comp_t[xx]-comp_t[xx-1])/1000 << " [s]\n";
227 
228 
229                         cout << "rel Permeability: " << mu_r;
230                         if (mu_r < 1)
231                         {
232                                 cout << " -> physical?!";
233                                 //abort();
234                         }
235                         cout << endl;
236 
237                         //cout << "spin update\n";
238 
239                         xx++;
240                         xxmax++;
241 
242                         T[xx] = T[xx-1] + dT[c];
243 
244                 }
245 
246 
247         }
248 
249         // cout << "start B sweep\n";
250         // ************ B-Scan *************** // Won't be activated here as Tsteps[] =
                0;
251 
252         Bx[xx] = Bxpt[0];
253         T[xx] = Tfix[0];
254 
255         for (c = 0; c < MR; c++)
256         {
257                 // one single field step at the beginning
258                 // int status = 0;      // variable 'status' and function 'countdown'
                        are strictly cosmetic: ~ computation status bar
259                 if (Bsteps[c] > 0)
260                 {
261                         Bx[xx] = Bxpt[c];
262                         T[xx] = Tfix[c];
263                         array<double, 2> arr = rel_perm(Bx[xx]);
264                         double mu_r = arr[0];
265                         double M = arr[1];
266                         //cout << "Single step, relaxation: ";
267 
268                         for (lx = 0; lx < BRlxLoops[c]; lx++) // Relaxation loop
269                         {
270                                 MCS_one(xx, Bx[xx], KB*T[xx], mu_r, M);
                                                        //status = countdown(BRlxLoops[c],
                                        lx, status);
271                         }
272 
273                         for (lm = 0; lm < BAvgLoops[c]; lm++) // Averaging loop
274                         {
275                                 MCS_one(xx, Bx[xx], KB*T[xx], mu_r, M);
276                                 Mx[xx] += measure();
277                         }
278 
279                         Mx[xx] /= ((double) TAvgLoops[c]);
280 
281                         auto t2 = chrono::high_resolution_clock::now();
282 
```

```
283                     chrono::duration<double, milli> fp_ms = t2-t1;
284                     comp_t.push_back(fp_ms.count());
285
286                     spinsave(edition);      // NOT ADVISED: GB files!
287                     filesave(edition, xx);
288                     Et = E_filesave(edition, xx);
289                     detailed_measure(edition, xx);
290
291                     cout << "cfg: " << cfg + 1 << ", T = " << T[xx] << ", B = " <<
                             Bx[xx] << ", Mx = " << Mx[xx] << ", Etot = " << Et << " in "
                             ;
292                     cout << (comp_t[xx]-comp_t[xx-1])/1000 << " [s]\n";
293
294
295
296                     ++xx;
297                     ++xxmax;
298
299                     Bx[xx] = Bx[xx-1] + dBx[c];
300             }
301
302             //status = 0;
303             for (StepNr = 0; StepNr < Bsteps[c]; StepNr++)
304             {
305                     T[xx] = Tfix[c];
306                     array<double, 2> arr = rel_perm(Bx[xx]);
307                     double mu_r = arr[0];
308                     double M = arr[1];
309                     //cout << "Relaxation progress: ";
310                     for (lx = 0; lx < BRlxLoops[c]; lx++) // Relaxation loop
311                     {
312                             MCS_one(xx, Bx[xx], KB*T[xx], mu_r, M);
313                             //status = countdown(BRlxLoops[c], lx, status);
314
315                     }
316
317                     for (lm = 0; lm < BAvgLoops[c]; lm++) // Averaging loop
318                     {
319                             //cout << "Schleife\n";
320
321                             MCS_one(xx, Bx[xx], KB*T[xx], mu_r, M);
322                             Mx[xx] += measure();
323                             //status = countdown(BAvgLoops[c], lm, status);
324                     }
325
326                     Mx[xx] /= ((double) TAvgLoops[c]);
327                     //cout << "Averaging done\n";
328
329                     auto t2 = chrono::high_resolution_clock::now();
330
331                     chrono::duration<double, milli> fp_ms = t2-t1;
332                     comp_t.push_back(fp_ms.count());
333
334                     spinsave(edition);      // NOT ADVISED: GB files!
335                     filesave(edition, xx);
336                     Et = E_filesave(edition, xx);
337                     detailed_measure(edition, xx);
338
339                     cout << "cfg: " << cfg + 1 << ", T = " << T[xx] << ", B = " <<
                             Bx[xx] << ", Mx = " << Mx[xx] << ", Etot = " << Et << " in "
                             ;
340                     cout << (comp_t[xx]-comp_t[xx-1])/1000 << " [s]\n";
341
342
343                     ++xx;
344                     ++xxmax;
345
346                     Bx[xx] = Bx[xx-1]+dBx[c];
347             }
348
349             // output.dat only updated after 1 sweep, NOT after every step as spin.
                     dat !
350
351
352
353
354
355             // OFF spintable();
356
357     } // \configuration loop
358
359
```

```
360         }
361         return 0;
362 }
363
364
365 //end.
```

B.1.2 parameter.h

```
 1 // parameter.h
 2 // configuration file
 3 // for MC simulation
 4
 5 using namespace std;
 6
 7
 8 extern const string data_file = "output";
 9 extern const string table_file = "spin";
10 extern const string aniso_file = "aniso";
11 extern const string Edata_file = "Edata";
12 extern const string Mdata_file = "Mdata";
13 //extern const string blockT_file = "blockT.dat";
14
15
16 // Natural constants. Only to be changed for convenience!!
17 const double pi = M_PI; // pi
18 const double KB = 1.38064852e-23; // Boltzmann constant
19 const double mu0 = pi*4e-7; // vacuum permeability
20 const double eV = 1.60217662e-19; // electron charge for easy conversion Joule <-> electron Volt
21 const double hbar = 1.0545718e-34; // h bar
22
23 // Parameters that distinguish the sample/experiment and its dynamics
24 const double d_m = 1.; // test-rotation vector length
25 const double Mag = 0.38e6*pi/6*pow(20e-9,3); // Magnetic moment (saturization mag. * Volume)
26 //const double KV = 0.;
27 const double KV = 1.34e4*pi/6*pow(20e-9,3); // anisotropy constant * Volume //
28 const double latt_const_0 = 1.*20e-9*1.1*sqrt(2.); // lattice constant of outer sc lattice
29 double latt_const; // lattice updated via edition parameter!!
30 const double def_mur = 15.;
31
32
33
34 // Parameters that define size of system and thus largely influence computation time!
35 //const bool DIPOLAR = true;
36 bool DIPOLAR;
37 const bool PERIODIC = true; // Calculate periodic system with ORF approx. If true, SPHERICAL is
       inactive
38 const bool PRINT_ENERGIES = false; // if true, all total energies are regularly calculated and
       saved
39 const bool PRINT_SPINS = false; // if true, spin_tables are regularly printed (GB files!!)
40 const bool DETAILED_MEASURE = true; // if true, other magnetizations than mx and their abs are
       recorded
41 const bool ONSDAT = false; // if true, all (!!!) Onsager energies will be recorded -> GB files!!
42 extern const string type = "fcc";
43 const bool SPHERICAL = false; // if true, original cube is cut off to a sphere
44
45 const int edge_N = 7; // # spins per edge
46 //const int max_N_temp = 2046 // # spins being considered
47 //const int max_N_temp = pow(edge_N,3); // general sc
48 int max_N;
49 //const int max_N_temp = pow(edge_N,3) + pow(edge_N-1,3); // general bcc
50 const int max_N_temp = 4*pow(edge_N,3); // general fcc
51 //const int max_N_temp = pow(edge_N,3) + 3*edge_N*pow(edge_N-1,2);
52 const int loopNrAvg = 10; // # MC Avgloops per step
53 const int loopNrRlx = 500; // # MC Rlxloops per step // !! in anisoBlock: loopNr2 = loopNr
       always !!
54 const double CUTOFF = 2; // maximum distance up to which dipole energy is calculated
55 double BEST_CUTOFF; // largest radius containing same neighbours as above value
56
57
58 // Parameters with mostly computational rather than physical meaning
59 const int numThreads = 4; // number of possible parallel threads
60 const int numThreads2 = 2; // alternative number of parallel threads
61 //const int Ntrunc = max_N/num_threads;
62 const int SEED = 11102017; //date as seed for rnd250()
63 const int MR = 3; // number of possible sweeps
64 const int MAX_DATA = 100000; // max. number of data points
65 const int CFG = 1; // number of configurations
66
67
68 // Temperature T in K(elvin)
```

```
69 // and
70 // magnetic field B in T(esla)
71
72 // Points for T - sweeps
73 const double Tpt[MR+1] = {1000, 25, 1000, 1000};
74 const double dT[MR] = {-25, 25,-100};
75 const double Bxfix[MR] = {0.0, 0.015, 0.015};
76 const int TRlxLoops[MR] = {loopNrRlx, loopNrRlx, loopNrRlx};
77 const int TAvgLoops[MR] = {loopNrAvg, loopNrAvg, loopNrAvg};
78
79 // Points for B- sweeps
80 const double Bxpt[MR+1] = {0.15, 0.15, 0.15,0.15};
81 const double dBx[MR] = {0.005, -0.005,0.005};
82 const double Tfix[MR] = {10, 10,10};
83 const int BRlxLoops[MR] = {loopNrRlx, loopNrRlx, loopNrRlx};
84 const int BAvgLoops[MR] = {loopNrAvg, loopNrAvg, loopNrAvg};
```

B.1.3 input-ini.h

```
1 // input_ini.h
2
3 // based on ini_spd5tbl.h
4 // computational necessities without physical meaning are defined here.
5 //
6
7
8 using namespace std;
9
10
11 // Spin positions
12 double POS[max_N_temp][3]; // Generates the sc lattice of spin positions.
13 bool intern[max_N_temp]; // flags if sites are within spherical cutoff
14 double N_POS[max_N_temp][3];
15
16
17 // Spin tables
18 double s[max_N_temp][3];
19 double sx[max_N_temp], sy[max_N_temp], sz[max_N_temp];
20
21 // Anisotropy easy axes
22 double k[max_N_temp][3];
23 double kx[max_N_temp], ky[max_N_temp], kz[max_N_temp];
24
25
26 void ini_config(); // initialize configuration
27 void InitArray(); // array initialization function -> CFG loop
28 void dist_matrix(); // calculate distance matrix for spin interaction
29 void r_vectors(); // calculate normalized distance vectors
30 void local_matrix(); // determine neighbourhood positions for dipole interaction
31 void line_generator(); // generates line of positions
32 void sc_generator(); // sc generator
33 void bcc_generator(); // bcc generator
34 void fcc_generator(); // fcc generator
35 void sc_only(); // POS only with sc generated
36 double best_cutoff();
37
38 int Tsteps[MR], Bsteps[MR];
39 int xxmax;
40 double T[MAX_DATA], Bx[MAX_DATA];
41 double Mx[MAX_DATA];
42 double TBsteps[CFG][5];
43
44
45
46
47
48 void InitMain(int ed)
49 {
50         int maxdat, i, xx;
51         int SEED_cluster;
52         if (ed < 1000)
53                 SEED_cluster = SEED ;
54         else
55                 SEED_cluster = SEED + ed;
56         seed250(SEED_cluster);
57         srand(SEED_cluster);
58
59         maxdat = 0;
60         for (i = 0; i < MR; i++)
61         {
62                 //Tsteps[i] = abs( (int) ((Tpt[i]-Tpt[i+1])/dT[i]));
```

```
63                  // Pure B Sweep!!!
64                  Tsteps[i] = abs( (int) ((Tpt[i]-Tpt[i+1])/dT[i]));
65                  Bsteps[i] = abs( (int) ((Bxpt[i]-Bxpt[i+1])/dBx[i]));
66                  //Bsteps[i] = 0; // Only T sweeps here!!
67                  maxdat += Tsteps[i]+Bsteps[i];
68          }
69
70          cout << (maxdat+1)*CFG << " Data points\n";
71
72          if ((maxdat*CFG+1) > MAX_DATA)
73          {
74                  cout << "too many data points!!\n";
75                  abort();
76          }
77
78          for (xx = 0; xx < MAX_DATA; xx++)
79          {
80                  Mx[xx] = 0.0;
81                  T[xx] = 0.0;
82                  Bx[xx] = 0.0;
83          }
84          for (int i = 0; i < CFG; i++)
85          {
86                  for (int j = 0; j <=5; j++)
87                  {
88                          TBsteps[i][j] = 0;
89                  }
90          }
91
92          cout << "data variables initialized!\n";
93  }
94
95  void InitArray()
96  {
97
98          cout << "initialize arrays...\n";
99          ini_config();
100         cout << "arrays initialized!\n";
101
102         // If no dipole-dipole interaction is considered, below functions are NOT necessary
103
104         cout << "calculate distances...\n";
105         dist_matrix();
106         cout << "distances calculated!\n";
107
108         if (PERIODIC)
109                 BEST_CUTOFF = best_cutoff();
110
111         if (PRINT_ENERGIES or not PERIODIC)
112         {
113
114                 cout << "calculate distance vectors...\n";
115                 r_vectors();
116                 cout << "distance vectors calculated!\n";
117
118         }
119
120
121
122                 /*
123         cout << "calculate dipole neighbours\n";
124         local_matrix();
125         cout << "dipole neighbours determined\n";
126         */
127 }
128
129 void ini_config()
130 {
131
132         for (int i = 0; i < max_N; i++)
133         {
134                 // Both spin and easy axis are stored as unit vectors because we need only the
                        angle between them via scalar product
135
136                 // Set initial spin vector directions
137                 Marsaglia(s[i]);
138
139                 sx[i] = s[i][0];
140                 sy[i] = s[i][1];
141                 sz[i] = s[i][2];
142
143                 // Set initial easy axes directions
```

```
144                 Marsaglia(k[i]);
145
146                 kx[i] = k[i][0];
147                 ky[i] = k[i][1];
148                 kz[i] = k[i][2];
149
150
151
152         }
153         cout << "Initial_spin_directions_set.\n";
154         cout << "Easy_axes_are_set.\n";
155 }
156
157
158 // Two optional functions that are only useful when comparing magnetization to theoretical
        paramagnetic case!
159
160 double Langevin(double Ezee, double kbT)
161 {
162         return (1/tanh(Ezee/kbT)-kbT/Ezee);
163 }
164
165 // Reset all output files
166 void reset_files(int ed)
167 {
168         string data_file_cluster = data_file + to_string(ed) + ".dat";
169         string table_file_cluster = table_file + to_string(ed) + ".dat";
170         string aniso_file_cluster = aniso_file + to_string(ed) + ".dat";
171         string Edata_file_cluster = Edata_file + to_string(ed) + ".dat";
172         string Mdata_file_cluster = Mdata_file + to_string(ed) + ".dat";
173
174         ofstream fout;
175         fout.open(data_file_cluster, ios::trunc);
176         fout.close();
177
178         ofstream spinout;
179         spinout.open(table_file_cluster, ios::trunc);
180         spinout.close();
181
182         ofstream anisout;
183         anisout.open(aniso_file_cluster, ios::trunc);
184         anisout.close();
185
186         ofstream Efout;
187         Efout.open(Edata_file_cluster, ios::trunc);
188         Efout.close();
189
190         ofstream Mfout;
191         Mfout.open(Mdata_file_cluster, ios::trunc);
192         Mfout.close();
193
194         if (ONSDAT)
195         {
196                 ofstream ons;
197                 ons.open("ons.dat", ios::trunc);
198                 ons.close();
199         }
200
201 /*
202         ofstream blockout;
203         blockout.open(blockT_file, ios::trunc);
204         blockout.close();
205 */
206 }
207
208
209 // Save data_file which contains magnetization at applied field, temperature
210 void filesave(int ed, int xx)
211 {
212         string data_file_cluster = data_file + to_string(ed) + ".dat";
213         ofstream fout;
214         fout.open(data_file_cluster, ios::app);
215         //fout << "cgf T B M\n";
216
217         fout << xx << "\t" << T[xx] << "\t" << Bx[xx] << "\t" << Mx[xx] << endl;
218
219
220         fout.close();
221 }
222
223
224 // Save spin positions. Every block of max_N rows corresponds to one row in data_file.
```

```
225 void spinsave(int ed)
226 {
227         if (PRINT_SPINS)
228         {
229
230                 string table_file_cluster = table_file + to_string(ed) + ".dat";
231
232                 ofstream spinout;
233                 spinout.open(table_file_cluster, ios::app);
234                 //fout << "cgf T B M\n";
235
236                 double X,Y,Z,VX,VY,VZ;
237
238
239
240                 for (int n = 0; n < max_N; n++)
241                         {
242
243
244                                 VX = sx[n];
245                                 VY = sy[n];
246                                 VZ = sz[n];
247
248                                 X = POS[n][0];
249                                 Y = POS[n][1];
250                                 Z = POS[n][2];
251
252                                 /*
253
254                                 // Operations on sx, POS in order to make the plots in gnuplot
                                        easier
255                                 VX = 0.5*sx[n];
256                                 VY = 0.5*sy[n];
257                                 VZ = 0.5*sz[n];
258
259                                 X = POS[n][0]/latt_const -0.5*VX;
260                                 Y = POS[n][1]/latt_const -0.5*VY;
261                                 Z = POS[n][2]/latt_const -0.5*VZ;
262                                 */
263                                 spinout << X << "\t" << Y << "\t" << Z << "\t" << VX << "\t" <<
                                        VY << "\t" << VZ << endl;
264
265                                 //spinout << POS[n][0] << "\t" << POS[n][1] << "\t" << POS[n][2]
                                        << "\t" << sx[n] << "\t" << sy[n] << "\t" << sz[n] << endl;
266                         }
267                         //fout << endl;
268
269                 spinout.close();
270         }
271 }
272
273
274 // Save easy axes at every site analogously to spinsave.
275 void anisosave(int ed)
276 {
277
278         string aniso_file_cluster = aniso_file + to_string(ed) + ".dat";
279         ofstream anisout;
280         anisout.open(aniso_file_cluster);
281
282         double X,Y,Z,VX,VY,VZ;
283
284
285
286         for (int n = 0; n < max_N; n++)
287                 {
288
289                         VX = kx[n];
290                         VY = ky[n];
291                         VZ = kz[n];
292
293                         X = POS[n][0];
294                         Y = POS[n][1];
295                         Z = POS[n][2];
296
297                         /*
298                         VX = 0.5*kx[n];
299                         VY = 0.5*ky[n];
300                         VZ = 0.5*kz[n];
301
302                         X = POS[n][0]/latt_const -0.5*VX;
303                         Y = POS[n][1]/latt_const -0.5*VY;
```

```
304                     Z = POS[n][2]/latt_const-0.5*VZ;
305                     */
306
307
308                     anisout << X << "\t" << Y << "\t" << Z << "\t" << VX << "\t" << VY << "\
                            t" << VZ << endl;
309
310                     //spinout << POS[n][0] << "\t" << POS[n][1] << "\t" << POS[n][2] << "\t"
                            << sx[n] << "\t" << sy[n] << "\t" << sz[n] << endl;
311             }
312             //fout << endl;
313
314         cout << "Wrote easy axes table.\n";
315         anisout.close();
316 }
317
318
319 //end.
```

B.1.4 str.h

```
 1 // str.h
 2 // functions and definitions that translate the physical structure of the simulated system are
       done here
 3
 4 using namespace std;
 5
 6
 7
 8 double dist[max_N_temp][max_N_temp];
 9 double rVectors[max_N_temp][max_N_temp][3];
10
11 // function that calculates array dist[max_N][max_N]
12
13 void line_generator(int edge, double start[3], int direction, int start_index)
14 {
15         int n = start_index;
16         double a = 1.;
17         int dir[3] = {0,0,0};
18         dir[direction] = 1;
19
20         for (int i = 0; i < edge; i++)
21         {
22                 POS[n][0] = a*i*dir[0]+start[0];
23                 POS[n][1] = a*i*dir[1]+start[1];
24                 POS[n][2] = a*i*dir[2]+start[2];
25                 n++;
26         }
27 }
28
29 void sc_generator(int edge, double start[3], int start_index)
30 {
31         int n = start_index;
32         double a = 1.;
33
34         for (int i = 0; i<edge; i++)
35         {
36                 for (int j = 0; j< edge; j++)
37                 {
38                         for (int k = 0; k< edge; k++)
39                         {
40                                 POS[n][0] = a*i+start[0];
41                                 POS[n][1] = a*j+start[1];
42                                 POS[n][2] = a*k+start[2];
43                                 n++;
44                         }
45                 }
46         }
47 }
48
49 void sc_only(int outer_edge)
50 {
51         double sc_start[3] = {0,0,0};
52         sc_generator(outer_edge, sc_start,0);
53         if (max_N_temp != pow(edge_N,3))
54         {
55                 cout << "Error in sc generation, check site numbers!" << endl;
56                 abort();
57         }
58 }
59
```

```
60  void bcc_generator(int outer_edge)
61  {
62          double a = 1.;
63          double outer_start[3] = {0,0,0};
64          double inner_start[3] = {0.5*a,0.5*a,0.5*a};
65          if (outer_edge < 2 or max_N_temp != pow(edge_N,3)+pow(edge_N-1,3))
66          {
67                  cout << "error in bcc generation, check site numbers!" << endl;
68                  abort();
69          }
70          sc_generator(outer_edge, outer_start, 0);
71          int outer_index = pow(outer_edge,3);
72          sc_generator(outer_edge-1, inner_start, outer_index);
73
74  }
75
76  void fcc_generator(int outer_edge)
77  {
78          double a = 1.;
79          double outer_start[3] = {0,0,0};
80
81          if (outer_edge < 2 or max_N_temp != pow(edge_N,3)+3*edge_N*pow(edge_N-1,2))
82          {
83                  cout << "error in fcc generation, check site numbers!" << endl;
84                  abort();
85          }
86          sc_generator(outer_edge, outer_start, 0);
87          int current_index = pow(outer_edge,3);
88
89
90  // generate lines in x-direction
91          double current_start[3] = {0,0,0};
92
93          for (int i = 0; i < outer_edge-1; i++)
94          {
95                  for (int j = 0; j < outer_edge-1; j++)
96                  {
97                          current_start[1] = a*(0.5+i);
98                          current_start[2] = a*(0.5+j);
99                          line_generator(outer_edge, current_start, 0, current_index);
100                         current_index += outer_edge;
101                 }
102
103         }
104         current_start[1] = 0;
105         current_start[2] = 0;
106
107 // generate lines in y-direction
108
109
110         for (int i = 0; i < outer_edge-1; i++)
111         {
112                 for (int j = 0; j < outer_edge-1; j++)
113                 {
114                         current_start[0] = a*(0.5+i);
115                         current_start[2] = a*(0.5+j);
116                         line_generator(outer_edge, current_start, 1, current_index);
117                         current_index += outer_edge;
118                 }
119
120         }
121
122         current_start[0] = 0;
123         current_start[2] = 0;
124
125 // generate lines in z-direction
126
127
128         for (int i = 0; i < outer_edge-1; i++)
129         {
130                 for (int j = 0; j < outer_edge-1; j++)
131                 {
132                         current_start[0] = a*(0.5+i);
133                         current_start[1] = a*(0.5+j);
134                         line_generator(outer_edge, current_start, 2, current_index);
135                         current_index += outer_edge;
136                 }
137
138         }
139         current_start[0] = 0;
140         current_start[1] = 0;
141
```

```
142 }
143
144 array<double, 4> calc_cent_rad()
145 {
146         array<double, 4> CENTER = {0,0,0,0};
147         for (int i = 0; i < max_N_temp; i++)
148         {
149                 CENTER[0] += POS[i][0];
150                 CENTER[1] += POS[i][1];
151                 CENTER[2] += POS[i][2];
152         }
153
154         CENTER[0] /= max_N_temp;
155         CENTER[1] /= max_N_temp;
156         CENTER[2] /= max_N_temp;
157
158         CENTER[3] = 0.5 * (edge_N -1);
159         return CENTER;
160 }
161
162 bool in_sphere(double x, double y, double z, array<double,4> center)
163 {
164         double x2 = center[0];
165         double y2 = center[1];
166         double z2 = center[2];
167         double R = center[3];
168
169         double r = pow(x-x2,2) + pow(y-y2,2) + pow(z-z2,2);
170
171         return (r <= R*R);
172
173 }
174
175
176 int det_spheric()
177 {
178         int MAX = 0;
179
180         array<double,4> CENTER = calc_cent_rad();
181
182         //cout << "Center: " << CENTER[0] << " " << CENTER[1] << " " << CENTER[2] << endl;
183         //cout << "Radius: " << CENTER[3] << endl;
184
185         fill_n(intern, max_N_temp, false);
186
187
188         double x, y, z;
189
190         for (int i = 0; i < max_N_temp; i++)
191         {
192
193                 x = POS[i][0];
194                 y = POS[i][1];
195                 z = POS[i][2];
196
197                 //cout << x << " " << y << " " << z << endl;
198
199                 if (in_sphere(x, y, z, CENTER))
200                 {
201                         MAX++;
202                         intern[i] = true;
203
204                 }
205
206         }
207
208         cout << MAX << " of " << max_N_temp << " in sphere.\n";
209
210         return MAX;
211 }
212
213
214 void new_pos()
215 {
216         int n = 0;
217
218         for (int i = 0; i < max_N_temp; i++)
219         {
220                 if (intern[i])
221                 {
222                         N_POS[n][0] = POS[i][0];
223                         N_POS[n][1] = POS[i][1];
```

```
224                             N_POS[n][2] = POS[i][2];
225
226                             n++;
227                     }
228
229             }
230
231             if (n != max_N)
232             {
233                     cout << "Error in spherical cutoff!\n";
234                     abort();
235             }
236
237             // reset POS with arbitrary value (-42)
238             fill(POS[0], POS[0] + max_N_temp * 3, -42.);
239
240             // for relevant indices, fill POS with N_POS
241             for (int i = 0; i < max_N; i++)
242             {
243                     POS[i][0] = N_POS[i][0];
244                     POS[i][1] = N_POS[i][1];
245                     POS[i][2] = N_POS[i][2];
246             }
247
248
249
250
251     }
252     /*
253             const int MAX = max_N;
254
255             double N_POS[MAX][3];
256
257     */
258
259     void dist_matrix()
260     {
261             int i = 0;
262
263             while(i < max_N)
264             {
265                     int j = i;
266                     while (j<max_N)
267                     {
268                             dist[i][j] = latt_const* sqrt(pow(POS[i][0]-POS[j][0],2) + pow(POS[i
                                    ][1]-POS[j][1],2) + pow(POS[i][2]-POS[j][2],2));
269                             j++;
270                     }
271                     i++;
272             }
273             int j = 0;
274             // counter loop for case i>j
275             while(j < max_N)
276             {
277                     int i = j;
278                     while (i<max_N)
279                     {
280                             dist[i][j] = dist[j][i];
281                             i++;
282                     }
283                     j++;
284             }
285
286             //cout << "Test dist " << dist[2][53] << " " << dist[53][2] << endl;
287     }
288
289
290     // calculation of the normalized distance vectors between sites -> rVectors[max_N][max_N][3]
291     void r_vectors()
292     {
293             for (int i = 0; i<max_N; i++)
294             {
295                     for (int j = i+1; j<max_N; j++)
296
297                     {
298                             //double d = dist[i][j];
299                             double vec[3] = {POS[i][0]-POS[j][0],POS[i][1]-POS[j][1],POS[i][2]-POS[j
                                    ][2]};
300                             double mod = sqrt(pow(vec[0],2)+pow(vec[1],2)+pow(vec[2],2));
301                             vec[0] /= mod;
302                             vec[1] /= mod;
303                             vec[2] /= mod;
```

```
304
305                     rVectors[i][j][0] = vec[0];
306                     rVectors[i][j][1] = vec[1];
307                     rVectors[i][j][2] = vec[2];
308                     //cout << vec[2] << endl;
309
310                     //Ed += 1./pow(d,3) * (3* (sx[j]*vec[0]+sy[j]*vec[1]+sz[j]*vec[2])*(sx[i
                            ]*vec[0]+sy[i]*vec[1]+sz[i]*vec[2]) - (sx[j]*sx[i]+sy[j]*sy[i]+sz[j
                            ]*sz[i]));
311             }
312
313         }
314
315
316         int j = 0;
317         // counter loop for case i>j
318         while(j < max_N)
319         {
320             int i = j;
321             while (i<max_N)
322             {
323                 rVectors[i][j][0] = rVectors[j][i][0];
324                 rVectors[i][j][1] = rVectors[j][i][1];
325                 rVectors[i][j][2] = rVectors[j][i][2];
326                 i++;
327             }
328             j++;
329         }
330
331         //cout << "Test r-Vec: " << rVectors[2][53][0] << " " << rVectors[2][53][1] << " " <<
                rVectors[2][53][2] << " " << rVectors[53][2][0] << " " << rVectors[53][2][1] << " "
                << rVectors[53][2][2] << endl;
332 }
333
334
335 // get the neighbours that are within the cutoff range and store their indices.
336 // Note that only i=0 gets all neighbours explicitly, neighbours for i>0 are partly contained
        for smaller i !!
337 /*
338 void local_matrix()
339 {
340         int min = max_N;        // completely optional:
341         int max = 0;            // show the minimal/maximal number of neighbours that will be
                considered for E_dipole
342
343         // loop over all sites i
344         for (int i = 0; i < max_N; i++)
345         {
346             int loc = 1;    // counts # neighbours for site i
347
348             loc_POS[i][0] = 0;      // 0th element contains number of neighbours, used in
                    loc_E_dipole()
349
350             // because of symmetry of E_dipole, only upper triangle matrix needs to filled
351             for (int j = i+1; j < max_N; j++) // only viable if loc_E_dipole() works with
                    corresponding structure!
352             {
353                 // loc_POS only filled with neighbouring sites within a distance of '
                        cutoff' around site i
354                 if (dist[i][j] <= cutoff)
355                 {
356                     loc_POS[i][loc] = j;    // So, loc_POS[a][b] = c reads: the
                            position vector of the bth neighbour of site a is stored as
                            cth element of the ORIGINAL POS array.
357                     loc++;
358                     loc_POS[i][0]+=1;
359                 }
360                 // Because of this construction, no dipole-dipole pair will be either
                        counted double or forgotten!
361             }
362             if (loc > max)
363             {
364                 max = loc;
365             }
366             if (loc < min)
367             {
368                 min = loc;
369             }
370
371
372         }
373
```

```
374         cout << "Up to " << max-1 << " neighbours per site considered instead of original " <<
                max_N-1 << endl;
375 }
376 */
377
378 double dipole(int i, int j)
379 {
380         double d = dist[i][j];
381
382         double vec[3];
383         vec[0] = rVectors[i][j][0];
384         vec[1] = rVectors[i][j][1];
385         vec[2] = rVectors[i][j][2];
386
387         if (i!=j)
388                 return -mu0*Mag*Mag/(4*PI * pow(d,3)) * (3* (sx[j]*vec[0]+sy[j]*vec[1]+sz[j]*vec
                        [2])*(sx[i]*vec[0]+sy[i]*vec[1]+sz[i]*vec[2]) - (sx[j]*sx[i]+sy[j]*sy[i]+sz[
                        j]*sz[i]));
389         else
390                 return 0;
391 }
392
393 // this function calculates the sum of all dipole-dipole energies w/o cutoff!
394 double E_dipole()
395 {
396         double Ed = 0;
397         //int status;
398
399         for (int i = 0; i<max_N; i++)
400         {
401                 for (int j = i+1; j<max_N; j++) // upper triangle matrix: main diagonal entries
                        must be 0, rest isn't determined because dipole pairs mustn't be counted
                        twice
402
403                 {
404
405                         double d = dist[i][j];
406
407                         double vec[3];
408                         vec[0] = rVectors[i][j][0];
409                         vec[1] = rVectors[i][j][1];
410                         vec[2] = rVectors[i][j][2];
411
412                         //cout << vec[2] << endl;
413
414
415                         Ed -= mu0*Mag*Mag/(4*PI * pow(d,3)) * (3* (sx[j]*vec[0]+sy[j]*vec[1]+sz[
                                j]*vec[2])*(sx[i]*vec[0]+sy[i]*vec[1]+sz[i]*vec[2]) - (sx[j]*sx[i]+
                                sy[j]*sy[i]+sz[j]*sz[i]));
416                         //status = countdown(max_N, i, status);
417                 }
418
419         }
420         return Ed;
421 }
422
423 double E_dipole_quick(int j)
424 {
425         double Ed = 0;
426         //int status;
427
428         for (int i = 0; i<max_N; i++)
429         {
430                 if (i != j)
431                 {
432                         double d = dist[i][j];
433
434                         double vec[3];
435                         vec[0] = rVectors[i][j][0];
436                         vec[1] = rVectors[i][j][1];
437                         vec[2] = rVectors[i][j][2];
438
439                         //cout << vec[2] << endl;
440
441
442                         Ed -= mu0*Mag*Mag/(4*PI * pow(d,3)) * (3* (sx[j]*vec[0]+sy[j]*vec[1]+sz[
                                j]*vec[2])*(sx[i]*vec[0]+sy[i]*vec[1]+sz[i]*vec[2]) - (sx[j]*sx[i]+
                                sy[j]*sy[i]+sz[j]*sz[i]));
443                         //status = countdown(max_N, i, status);
444                 }
445
446         }
```

```
447         return Ed;
448
449 }
450
451 double E_dipole_total()
452 {
453         double Ed = 0;
454
455         omp_set_num_threads(numThreads);
456         #pragma omp parallel for reduction(+:Ed)
457         for (int i = 0; i < max_N-1; i++)
458         {
459                 for (int j = i+1; j < max_N; j++)
460                 {
461                         Ed += dipole(i,j);
462                 }
463         }
464
465         Ed /= pow((edge_N-1)*latt_const,3);
466         return Ed;
467 }
468
469 double E_aniso_total()
470 {
471         double Ea = 0;
472
473         omp_set_num_threads(numThreads);
474         #pragma omp parallel for reduction(+:Ea)
475         for (int n = 0; n < max_N; n++)
476                 //Ea += KV *(1- pow(sx[n]*kx[n]+sy[n]*ky[n]+sz[n]*kz[n], 2));
477                 Ea += KV *(0- pow(sx[n]*kx[n]+sy[n]*ky[n]+sz[n]*kz[n], 2));
478
479         Ea /= pow((edge_N-1)*latt_const,3);
480         return Ea;
481 }
482
483 double E_zee_total(double BBx)
484 {
485         double Ezee = 0;
486
487         omp_set_num_threads(numThreads);
488         #pragma omp parallel for reduction(-:Ezee)
489         for (int n = 0; n < max_N; n++)
490                 Ezee += -Mag * BBx * sx[n];
491
492         Ezee /= pow((edge_N-1)*latt_const,3);
493         return Ezee;
494
495 }
496
497
498 double E_surf_total()
499 {
500         double Esurf = 0;
501
502         omp_set_num_threads(numThreads);
503         #pragma omp parallel for reduction(+:Esurf)
504         for (int n = 0; n < max_N-1; n++)
505         {
506                 for (int m = n+1; m < max_N; m++)
507                         Esurf += sx[n]*sx[m] + sy[n]*sy[m] + sz[n]*sz[m];
508         }
509
510         Esurf *= mu0 * Mag * Mag / (2 * pow((edge_N-1)*latt_const, 6));
511
512         return Esurf;
513 }
514
515
516 double E_dipole_quick_MULT(int j)
517 {
518         double Ed = 0;
519         //int Ntrunc = max_N/num_threads;
520         //int status;
521         omp_set_num_threads(numThreads);
522         #pragma omp parallel for reduction(+:Ed)
523         for (int i = 0; i < max_N; i++)
524         {
525                 Ed += dipole(i,j);
526         }
527
528         return Ed;
```

```
529 }
530
531
532
533 double E_filesave(int ed, int xx)
534 {
535         double Etotal = 0;
536
537         if (PRINT_ENERGIES)
538         {
539                 string data_file_cluster = Edata_file + to_string(ed) + ".dat";
540                 ofstream fout;
541                 fout.open(data_file_cluster, ios::app);
542                 //fout << "cgf T B M\n";
543
544                 double Ed, Ea, Ezee, Esurf;
545
546                 Ed = E_dipole_total();
547                 Ea = E_aniso_total();
548                 Ezee = E_zee_total(Bx[xx]);
549                 Esurf = E_surf_total();
550                 Etotal = Ed + Ea + Ezee + Esurf;
551
552                 fout << xx << "\t" << T[xx] << "\t" << Bx[xx] << "\t" << Ezee << "\t" << Ea << "
                        \t" << Ed << "\t" << Esurf << "\t" << Etotal << endl;
553
554
555                 fout.close();
556         }
557
558         return Etotal;
559 }
560
561 void detailed_measure(int ed, int xx)
562 {
563         if (DETAILED_MEASURE)
564         {
565                 double m_x, m_y, m_z, MX, MY, MZ;
566                 m_x = 0;
567                 m_y = 0;
568                 m_z = 0;
569                 MX = 0;
570                 MY = 0;
571                 MZ = 0;
572
573                 omp_set_num_threads(2);
574                 #pragma omp parallel for reduction(+:m_x), reduction(+:m_y), reduction(+:m_z),
                        reduction(+:MX), reduction(+:MY), reduction(+:MZ)
575                 for (int n = 0; n < max_N; n++)
576                 {
577                         m_x += sx[n]; // measure in x-direction
578                         m_y += sy[n];
579                         m_z += sz[n];
580
581                         MX += abs(sx[n]); // measure total x-alignment
582                         MY += abs(sy[n]);
583                         MZ += abs(sz[n]);
584
585                 }
586
587                 m_x /= max_N;
588                 m_y /= max_N;
589                 m_z /= max_N;
590                 MX /= max_N;
591                 MY /= max_N;
592                 MZ /= max_N;
593
594                 string data_file_cluster = Mdata_file + to_string(ed) + ".dat";
595                 ofstream fout;
596                 fout.open(data_file_cluster, ios::app);
597                 //fout << "cgf T B M\n";
598
599
600                 fout << xx << "\t" << T[xx] << "\t" << Bx[xx] << "\t" << m_x << "\t" << m_y << "
                        \t" << m_z << "\t" << MX << "\t" << MY << "\t" << MZ << endl;
601
602
603                 fout.close();
604
605         }
606 }
607
```

```
608
609
610  /*
611  // calculate E_dipole, but at every site i, only the local neighbours determined in local_matrix
         () are considered
612  double loc_E_dipole()
613  {
614          double Ed = 0;
615          for (int i = 0; i<max_N; i++)
616                  {
617                          int max = loc_POS[i][0]; // exploit that we already calculated the
                                 number of neighbours = number of loops
618
619                          // the upper triangle matrix is already implemented in local_matrix(),
                                 therefore j starts always at 1 as the 0th element is #(neighbours),
                                 NOT an index !!
620                          for (int j = 1; j<max; j++)
621
622                          {
623                                  int l = loc_POS[i][j];  // l is the 'old' index when all sites
                                         were considered so that we can still use dist[][] and
                                         rVectors[][][]
624
625                                  // exactly the same as in E_dipole() from here.
626                                  double d = dist[i][l];
627
628                                  double vec[3];
629                                  vec[0] = rVectors[i][l][0];
630                                  vec[1] = rVectors[i][l][1];
631                                  vec[2] = rVectors[i][l][2];
632
633
634
635                                  Ed -= mu0*Mag*Mag/(4*PI * pow(d,3)) * (3* (sx[l]*vec[0]+sy[l]*
                                         vec[1]+sz[l]*vec[2])*(sx[i]*vec[0]+sy[i]*vec[1]+sz[i]*vec
                                         [2]) - (sx[l]*sx[i]+sy[l]*sy[i]+sz[l]*sz[i]));
636                                  //status = countdown(max_N, i, status);
637                          }
638
639                  }
640                  return Ed;
641
642  }
643  */
644
645
646
647  double measure()
648  {
649          double m_x = 0.;
650          omp_set_num_threads(numThreads);
651          #pragma omp parallel for reduction(+:m_x)
652          for (int n = 0; n < max_N; n++)
653          {
654                  m_x += sx[n]; // measure in x-direction
655          }
656
657          return m_x/max_N;
658  }
659
660  array<double,2> rel_perm(double B)
661  {
662          array<double,2> res;
663          double mu_r;
664          double M;
665          double meas = measure();
666          double VOL = pow((edge_N-1)*latt_const,3);
667
668
669          //double z = mu0*max_N*measure()*Mag/(3*VOL*B);
670          M = max_N*meas*Mag/VOL;
671
672          if (B == 0)
673                  mu_r = def_mur;
674          else
675          {
676                  //mu_r = (1+2*z)/(1-z);
677                  mu_r = 1 + mu0*M/B;
678                  //mu_r = B / (B-mu0*max_N*measure()*Mag);
679
680
681          }
```

```
682
683
684         //mu_r = def_mur;
685         res[0] = mu_r;
686         res[1] = M;
687
688         return res;
689 }
690
691 double best_cutoff()
692 {
693         double best_CUTOFF = edge_N-1;
694         double curr_dist;
695
696         for (int i = 0; i < max_N-1; i++)
697         {
698                 for (int j = i + 1; j < max_N; j++)
699                 {
700                         curr_dist = dist[i][j]/latt_const;
701
702                         if (curr_dist > CUTOFF and curr_dist < best_CUTOFF)
703                                 best_CUTOFF = curr_dist;
704
705                         //cout << curr_dist << "\t";
706                 }
707         }
708         best_CUTOFF -= 0.000001;
709         cout << "cutoff_is_" << best_CUTOFF << "_up_from_" << CUTOFF << endl;
710
711         return best_CUTOFF;
712 }
713
714
715
716 vector<int> shuffle_list()
717 {
718         vector<int> rand_list;
719         for (int i = 0; i < max_N; i++)
720                 rand_list.push_back(i);
721         random_shuffle(rand_list.begin(), rand_list.end());
722         return rand_list;
723 }
724
725 void MCS_ALT(int xx, double BBx, double kt)
726 {
727         int n;
728         double E0, E1, E_a, mod_M, M_temp[3], d_M[3];
729         //int status;
730         for (n = 0; n<max_N; n++)
731         {
732                 E_a = -KV * pow(sx[n]*kx[n]+sy[n]*ky[n]+sz[n]*kz[n], 2); // Anistropy energy
733                 //E_d = loc_E_dipole();          // choose cutoff approximation for speed-up
734                 //E_d = E_dipole();              // choose to consider ALL sites for dipole-
                        dipole i.a.
735                 E0 = - Mag * BBx * sx[n]; // field in x-direction (Zeeman energy)
736                 E0 += E_a; // sum of dipole energies
737
738                 M_temp[0] = sx[n];
739                 M_temp[1] = sy[n];
740                 M_temp[2] = sz[n];
741
742                 Marsaglia(d_M);
743
744                 sx[n] += (d_m * d_M[0]);
745                 sy[n] += (d_m * d_M[1]);
746                 sz[n] += (d_m * d_M[2]);
747                 mod_M = sqrt(pow(sx[n],2)+pow(sy[n],2)+pow(sz[n],2));
748
749                 sx[n] /= mod_M;
750                 sy[n] /= mod_M;
751                 sz[n] /= mod_M;
752
753                 E_a = -KV * pow(sx[n]*kx[n]+sy[n]*ky[n]+sz[n]*kz[n], 2); // Anistropy energy
754                 //E_d = loc_E_dipole();
755                 //E_d = E_dipole();
756                 E1 = - Mag * BBx * sx[n]; // must be same (updated) calculation as E0
757                 E1 += E_a;
758
759                 if (E1 > E0)
760                 {
761                         if (rand0_1() > exp((E0-E1)/kt))
762                         {
```

```
763                                 sx[n] = M_temp[0];
764                                 sy[n] = M_temp[1];
765                                 sz[n] = M_temp[2];
766                         }
767                 }
768
769
770                 //status = countdown(max_N, n, status);
771
772         }
773
774 }
775
776 void MCS_one(int xx, double BBx, double kt, double mu_r, double M)
777 {
778         double E0, E1, E_a, E_d, mod_M, M_temp[3], d_M[3];
779
780         vector<int> rand_list = shuffle_list(); // experimental DO NOT USE IN THIS VERSION
781         //int status;
782         for (int m = 0; m<max_N; m++)
783         {
784                 E_d = 0;
785                 int n = rand_list[m]; // every superspin updated exactly once per MCS_one call,
                        but in a random fashion -> permutation
786                 //int n = rnd250()%max_N;
787                 //int n = m;
788
789                 E_a = -KV * pow(sx[n]*kx[n]+sy[n]*ky[n]+sz[n]*kz[n], 2); // Anistropy energy
790                 //E_d = loc_E_dipole();         // choose cutoff approximation for speed-up
791                 if (DIPOLAR)
792                 {
793                         if (PERIODIC)
794                         {
795                                 E_d = ORF(n, mu_r, BBx, M);
796                                 E0 = 0;
797                                 //E_a *= (3*mu_r / (1+2*mu_r));
798                         }
799                         else
800                         {
801                                 E_d = E_dipole_quick_MULT(n);
802                                 E0 = - Mag * BBx * sx[n]; // field in x-direction (Zeeman energy
                                    )
803                         }
804                 }
805
806                 else
807                         E0 = - Mag * BBx * sx[n]; // field in x-direction (Zeeman energy)
808
809
810
811                 E0 += E_a; //
812                 E0 += E_d; // sum of dipole energies
813
814                 M_temp[0] = sx[n];
815                 M_temp[1] = sy[n];
816                 M_temp[2] = sz[n];
817
818                 Marsaglia(d_M);
819
820                 sx[n] += (d_m * d_M[0]);
821                 sy[n] += (d_m * d_M[1]);
822                 sz[n] += (d_m * d_M[2]);
823                 mod_M = sqrt(pow(sx[n],2)+pow(sy[n],2)+pow(sz[n],2));
824
825                 sx[n] /= mod_M;
826                 sy[n] /= mod_M;
827                 sz[n] /= mod_M;
828
829                 E_d = 0;
830
831                 E_a = -KV * pow(sx[n]*kx[n]+sy[n]*ky[n]+sz[n]*kz[n], 2); // Anistropy energy
832                 //E_d = loc_E_dipole();
833
834                 if (DIPOLAR)
835                 {
836                         if (PERIODIC)
837                         {
838                                 E_d = ORF(n, mu_r, BBx, M);
839                                 E1 = 0;
840                                 //E_a *= (3*mu_r / (1+2*mu_r));
841                         }
842                         else
```

```
843                     {
844                             E_d = E_dipole_quick_MULT(n);
845                             E1 = - Mag * BBx * sx[n]; // field in x-direction (Zeeman energy
                                    )
846                     }
847             }
848
849             else
850                     E1 = - Mag * BBx * sx[n]; // field in x-direction (Zeeman energy)
851
852
853             E1 += E_a;
854             E1 += E_d;
855
856             if (E1 > E0)
857             {
858                     if (rand0_1() > exp((E0-E1)/kt))
859                     {
860                             sx[n] = M_temp[0];
861                             sy[n] = M_temp[1];
862                             sz[n] = M_temp[2];
863                     }
864             }
865
866
867             //status = countdown(max_N, n, status);
868
869     }
870
871 }
872
873
874
875
876
877 void aniso_test()
878 {
879     int n;
880     double t_kx = 0.0;
881     double t_ky = 0.0;
882     double t_kz = 0.0;
883
884     double t_sx = 0.0;
885     double t_sy = 0.0;
886     double t_sz = 0.0;
887
888
889     for (n = 0; n < max_N; n++)
890     {
891             t_kx += kx[n];
892             t_ky += ky[n];
893             t_kz += kz[n];
894
895             t_sx += sx[n];
896             t_sy += sy[n];
897             t_sz += sz[n];
898
899     }
900
901     t_kx /= max_N;
902     t_ky /= max_N;
903     t_kz /= max_N;
904
905     cout << "Avg easy axis components xyz: " << t_kx <<" "<< t_ky <<" "<< t_kz << endl;
906
907
908
909     t_sx /= max_N;
910     t_sy /= max_N;
911     t_sz /= max_N;
912
913     cout << "Avg Superspin components xyz: " << t_sx <<" "<< t_sy <<" "<< t_sz << endl;
914
915 }
```

B.1.5 perAux.h

```
1 // perAux.h
2 // additional structural functions for interacting periodic systems.
3 // intended for simulations with ORF method
4 // makes use of openMP for easily implemented multiprocessing
```

```
5   // for MC simulation
6   using namespace std;
7
8   const int perFactor = 3;
9   const int NN = 10*(pow(edge_N+2*CUTOFF,3)-pow(edge_N,3))+max_N_temp;
10
11  const double rho_bcc = 1. + pow(((double) (edge_N-1))/((double)(edge_N)), 3);
12  const double rho_fcc = 1. + 3* pow(((double) (edge_N-1))/((double)(edge_N)),2);
13
14  const int CUT_VOL = 100*pow(CUTOFF/1.5,3)+1;
15
16  double perPOS[NN][3];
17  double dist_per[max_N_temp][NN];
18  double rVectors_per[max_N_temp][NN][3];
19  int loc_POS[max_N_temp][CUT_VOL];
20
21  int max_N_per;
22  vector<int> perInd;
23
24  void sc_generator_new(int edge)
25  {
26          int n = 0;
27          double a = 1;
28
29          for (int i = 0; i < edge; i++)
30          {
31                  for (int j = 0; j < edge; j++)
32                  {
33                          for (int k = 0; k < edge; k++)
34                          {
35                                  POS[n][0] = 0+a*i;
36                                  POS[n][1] = 0+a*j;
37                                  POS[n][2] = 0+a*k;
38
39                                  n++;
40                          }
41                  }
42          }
43  }
44
45  void fcc_generator_new(int edge)
46  {
47          int n = 0;
48          double a = 1.;
49
50          for (int i = 0; i<edge; i++)
51          {
52                  for (int j = 0; j< edge; j++)
53                  {
54                          for (int k = 0; k< edge; k++)
55                          {
56                                  POS[n][0] = 0+a*i;
57                                  POS[n][1] = 0+a*j;
58                                  POS[n][2] = 0+a*k;
59
60                                  POS[n+1][0] = 0.5+a*i;
61                                  POS[n+1][1] = 0.5+a*j;
62                                  POS[n+1][2] = 0+a*k;
63
64                                  POS[n+2][0] = 0.5+a*i;
65                                  POS[n+2][1] = 0+a*j;
66                                  POS[n+2][2] = 0.5+a*k;
67
68                                  POS[n+3][0] = 0+a*i;
69                                  POS[n+3][1] = 0.5+a*j;
70                                  POS[n+3][2] = 0.5+a*k;
71
72
73                                  n+=4;
74                          }
75                  }
76          }
77  }
78
79  void bcc_generator_new(int edge)
80  {
81          int n = 0;
82          double a = 1.;
83
84          for (int i = 0; i<edge; i++)
85          {
86                  for (int j = 0; j< edge; j++)
```

```
87                  {
88                          for (int k = 0; k< edge; k++)
89                          {
90                                  POS[n][0] = 0+a*i;
91                                  POS[n][1] = 0+a*j;
92                                  POS[n][2] = 0+a*k;
93
94                                  POS[n+1][0] = 0.5+a*i;
95                                  POS[n+1][1] = 0.5+a*j;
96                                  POS[n+1][2] = 0.5+a*k;
97
98                                  n+=2;
99                          }
100                 }
101         }
102 }
103
104 void hcp_generator_new(int edge)
105 {
106         int n = 0;
107         double a = 1.;
108
109         for (int i = 0; i<edge; i++)
110         {
111                 for (int j = 0; j< edge; j++)
112                 {
113                         for (int k = 0; k< edge; k++)
114                         {
115                                 POS[n][0] = (2*i+((j+k)%2))*a/2.;
116                                 POS[n][1] = sqrt(3.)*(j+1./3.*(k%2))*a/2.;
117                                 POS[n][2] = 2*sqrt(6.)/3.*k*a/2.;
118
119
120                                 n+=1;
121                         }
122                 }
123         }
124 }
125
126 bool inn_cube(double x, double y, double z, double marg)
127 {
128         if (x < 0 + marg)
129                 return false;
130         if (x > (edge_N -1)-marg)
131                 return false;
132
133         if (y < 0 + marg)
134                 return false;
135         if (y > (edge_N -1)-marg)
136                 return false;
137
138         if (z < 0 + marg)
139                 return false;
140         if (z > (edge_N -1)-marg)
141                 return false;
142
143         return true;
144 }
145
146 vector<int> find_edgeInd2()
147 {
148         vector<int> EDGE;
149
150         double x,y,z;
151
152         for (int i = 0; i < max_N; i++)
153         {
154                 x = POS[i][0];
155                 y = POS[i][1];
156                 z = POS[i][2];
157
158                 if (not inn_cube(x,y,z, CUTOFF))
159                         EDGE.push_back(i);
160         }
161
162         return EDGE;
163 }
164
165 bool copyable(double x, double y, double z, int cx, int cy, int cz)
166 {
167
168         bool trivial =  (cx == 0 and cy ==0 and cz == 0);
```

```
169
170         if (inn_cube(x,y,z,-sqrt(3)*CUTOFF) and not trivial)
171                 return true;
172
173         return false;
174
175 }
176
177 vector<array<double,3>> copies(array<double,3> original)
178 {
179         vector<array<double, 3>> CAND;
180         double x0 = original[0];
181         double y0 = original[1];
182         double z0 = original[2];
183
184         double x, y, z;
185
186         int sgn[3] = {-1,0,1};
187         array<double,3> cand;
188
189
190         for (int i = 0; i < 3; i++)
191         {
192                 for (int j = 0; j<3; j++)
193                 {
194                         for (int k = 0; k < 3; k++)
195                         {
196                                 x = x0 + edge_N*sgn[i];
197                                 y = y0 + edge_N*sgn[j];
198                                 z = z0 + edge_N*sgn[k];
199
200                                 if(copyable(x,y,z, sgn[i], sgn[j], sgn[k]))
201                                 {
202                                         cand[0] = x;
203                                         cand[1] = y;
204                                         cand[2] = z;
205                                         CAND.push_back(cand);
206                                 }
207
208                         }
209                 }
210
211         }
212
213         return CAND;
214 }
215
216 double distance(int i, int j)
217 {
218         double x1, x2, y1, y2, z1, z2, r;
219
220         x1 = perPOS[i][0];
221         y1 = perPOS[i][1];
222         z1 = perPOS[i][2];
223
224         x2 = perPOS[j][0];
225         y2 = perPOS[j][1];
226         z2 = perPOS[j][2];
227
228         r = sqrt(pow(x1-x2,2)+pow(y1-y2,2)+pow(z1-z2,2));
229
230         return r;
231 }
232
233
234 void makePeriodic()
235 {
236
237         vector<int> EDGE;
238         EDGE = find_edgeInd2();
239
240
241         int len = EDGE.size();
242         int edgeInd;
243
244
245         cout << len << " of " << max_N << " in edge.\n";
246
247         if (len >= max_N - 1)
248         {
249                 cout << "Superspins will interact with own copies -> not physical!\n";
250                 bool cont;
```

```
251                 cout << "Continue regardless? ";
252                 cin >> cont;
253                 if (not cont)
254                         abort();
255         }
256 
257 
258         array<double, 3> orig_pos;
259         vector<array<double, 3>> copy_vector;
260 
261 
262         int len2;
263         int PERlen = 0;
264         double per_x, per_y, per_z;
265 
266 
267         for (int i = 0; i < max_N; i++)
268         {
269                 perPOS[i][0] = POS[i][0];
270                 perPOS[i][1] = POS[i][1];
271                 perPOS[i][2] = POS[i][2];
272 
273                 perInd.push_back(i);
274         }
275 
276 
277         for (int j = 0; j < len; j++)
278         {
279                 edgeInd = EDGE[j];
280 
281                 orig_pos[0] = POS[edgeInd][0];
282                 orig_pos[1] = POS[edgeInd][1];
283                 orig_pos[2] = POS[edgeInd][2];
284 
285                 copy_vector = copies(orig_pos);
286 
287                 len2 = copy_vector.size();
288                 for (int i = 0; i < len2; i++)
289                 {
290                         per_x = copy_vector[i][0];
291                         per_y = copy_vector[i][1];
292                         per_z = copy_vector[i][2];
293 
294 
295 
296                         perPOS[PERlen+max_N][0] = per_x;
297                         perPOS[PERlen+max_N][1] = per_y;
298                         perPOS[PERlen+max_N][2] = per_z;
299 
300 
301                         perInd.push_back(edgeInd);
302 
303 
304 
305                         PERlen++;
306 
307                 }
308 
309         }
310 
311         cout << PERlen << " copies determined.\n";
312 
313         cout << "Index conversion saved.\n";
314 
315         int neighb, k;
316         double d;
317         omp_set_num_threads(numThreads);
318         #pragma omp parallel for private(k, neighb, d)
319         for (int i = 0; i < max_N; i++)
320         {
321                 //cout << "here\n";
322 
323                 neighb = 0;
324 
325                 for (k = 0; k < max_N + PERlen; k++)
326                 {
327                         d = distance(i,k);
328                         if (i != k and d <= CUTOFF)
329                         {
330                                 neighb++;
331                                 loc_POS[i][neighb] = k;
332                                 dist_per[i][k] = d;      // no symmetrization necessary b/c k
```

```
                                    runs through ALL indices
333                             double vec[3] = {perPOS[i][0]-perPOS[k][0],perPOS[i][1]-perPOS[k
                                    ][1],perPOS[i][2]-perPOS[k][2]};
334                             double mod = sqrt(pow(vec[0],2)+pow(vec[1],2)+pow(vec[2],2));
335                             vec[0] /= mod;
336                             vec[1] /= mod;
337                             vec[2] /= mod;
338
339                             rVectors_per[i][k][0] = vec[0];
340                             rVectors_per[i][k][1] = vec[1];
341                             rVectors_per[i][k][2] = vec[2];
342                     }
343
344             }
345
346             loc_POS[i][0] = neighb;
347
348     }
349
350
351
352     cout << loc_POS[0][0] << "_neighbours_per_site." << endl;
353
354 double ORF(int i, double mu_r, double B, double M)
355 {
356
357     double Ed = 0;
358     double E_RF = 0;
359
360     double sx1 = sx[i];
361     double sy1 = sy[i];
362     double sz1 = sz[i];
363
364     double Sx = sx1;
365     double Sy = sy1;
366     double Sz = sz1;
367
368
369     int len = loc_POS[i][0];
370
371
372
373     omp_set_num_threads(numThreads2);
374     #pragma omp parallel for reduction(+:Ed), reduction(+:Sx), reduction(+:Sy), reduction(+:
            Sz)
375     for (int j = 1; j <= len; j++)
376     {
377
378             int neighbInd1 = loc_POS[i][j];
379
380             int neighbInd2 = perInd[neighbInd1];
381
382             double sx2 = sx[neighbInd2];
383             double sy2 = sy[neighbInd2];
384             double sz2 = sz[neighbInd2];
385
386             Sx += sx2;
387             Sy += sy2;
388             Sz += sz2;
389
390
391             double d = dist_per[i][neighbInd1];
392
393             double vec[3];
394
395
396             vec[0] = rVectors_per[i][neighbInd1][0];
397             vec[1] = rVectors_per[i][neighbInd1][1];
398             vec[2] = rVectors_per[i][neighbInd1][2];
399
400             Ed += -mu0*Mag*Mag/(4*PI * pow(d*latt_const,3)) * (3* (sx1*vec[0]+sy1*vec[1]+sz1
                    *vec[2])*(sx2*vec[0]+sy2*vec[1]+sz2*vec[2]) - (sx1*sx2+sy1*sy2+sz1*sz2));
401
402     }
403
404     E_RF = -sx1*(B-0) * Mag * (3*mu_r / (1+2*mu_r));
405
406     E_RF += mu0* Mag*Mag * (sx1*Sx + sy1*Sy + sz1*Sz) * (1-mu_r)/(2*mu_r+1)/(2*PI*pow(
            BEST_CUTOFF*latt_const,3));
407
408     if (ONSDAT)
409     {
```

```
410                 ofstream ons_file;
411                 ons_file.open("ons.dat", ios::app);
412                 ons_file << Ed << "\t" << E_RF << endl;
413                 ons_file.close();
414         }
415         return Ed + E_RF;
416 
417 }
```

B.1.6 rnd250.c

```
 1 /*
 2  *  RND250.C
 3  *
 4  *  Zufallszahlengenerator nach dem Verfahren von Kirkpatrick und Stoll.
 5  *  Dieses C-Modul enthaelt nur die globale Definition der Daten und die
 6  *  Initialisierungsroutine. Der eigentliche Zufallszahlengenerator ist
 7  *  als Makro in der Datei RND250.H kodiert.
 8  *
 9  *  Implementation: Ralf Meyer, Fred Hucht
10  *  Version        : 2.0
11  *  entwickelt am : 14. Februar 1995
12  *
13  *  Copyright (c) 1995 Ralf Meyer, 47058 Duisburg, Germany
14  */
15 
16 #include "rnd.h"
17 struct st_rnd250 Rnd250;
18 
19 
20 void
21 seed250 (long seed)
22 /*
23  *  Initialisiert den Zufallszahlengenerator. Hierzu wird ein Modulo-
24  *  Zufallszahlengenerator benutzt. Um die lineare Unabhaengigkeit der
25  *  einzelnen Bits zu garantieren werden nachtraeglich noch Bitmasken
26  *  einem Teil der Daten ueberlagert. Um die dadurch verursachten An-
27  *  fangskorrelationen zu beseitigen werden die ersten Zufallszahlen
28  *  verworfen.
29  *
30  *  Parameter:
31  *    seed -- Startwert
32  */
33 {
34   int    i;
35   long   j, k;
36 
37   if (seed < 1)                                /* keine negativen Startwerte */
38     seed = 1;
39 
40   for (i=0; i<250; ++i)  {       /* Schieberegister mit Zufallszahlen fuellen */
41     k = seed / 127773;
42     seed = 16807 * (seed - k*127773) - 2836*k;
43     if (seed < 0)
44       seed += 0x7FFFFFFF;
45     Rnd250.field[i] = seed;
46   }
47 
48   k = 0x7FFFFFFF;                                      /* Masken ueberlagern */
49   j = 0x40000000;
50   for (i=1; i<250; i+=8)
51     Rnd250.field[i] = (Rnd250.field[i] & k) | j;
52 
53   Rnd250.point = 249;                               /* Zeiger initialisieren */
54 
55   for (i=0; i<4711; ++i)                          /* Anfangszahlen verwerfen */
56     rnd250();
57 }
```

B.1.7 random-spd5.h

```
1 //
2 // random_spd5.h
3 //
4 // functions that require randum number generators are collected here
5 // intended to use rnd.h by Ralf Meyer
6 
7 const double PI = 3.1415926535897932385;
8 
```

```
9    double GaussDis(double,double,double);
10
11
12   const double real_RND250_MAX = (double) MAXRAND250;
13
14   /*
15           generate a random number in [0,1)
16   */
17   double rand0_0999()
18   {
19           return (rnd250() / (real_RND250_MAX+1.0));
20   }
21
22
23   /*
24           generate a random number in (0,1)
25   */
26   double rand0001_0999()
27   {
28           return ((rnd250()+1.0) / (real_RND250_MAX+2.0));
29   }
30
31
32   /*
33           generate a random number in [0,1]
34   */
35   double rand0_1()
36   {
37           return (rnd250() / real_RND250_MAX);
38   }
39
40
41
42   /*
43           yield a 3D random unit vector
44   */
45   void Marsaglia(double *V)
46   {
47           double rsq, y1, y2;
48           do
49       {
50                   y1 = rand0001_0999() * 2.0 - 1.0;
51                   y2 = rand0001_0999() * 2.0 - 1.0;
52                   rsq = y1 * y1 + y2 * y2;
53           }
54           while (rsq > 1.0);
55           V[0] = 2.0 * y1 * sqrt((1.0 - rsq));
56           V[1] = 2.0 * y2 * sqrt((1.0 - rsq));
57           V[2] = 1.0 - 2.0 * rsq;
58   }
59
60   double GaussDis(double x, double avg, double sig)
61   {
62           double fct;
63
64           fct = (1.0/(sqrt(2.0*PI)*sig))*exp(-0.5*(x-avg)*(x-avg)/(sig*sig));
65
66           return(fct);
67   }
68
69
70   /*
71       yields a 3D unit vector, but with Gauss-Distribution in y,z-coordinates
72      <-> anisotropy axis distribution - easy axis in x- direction
73
74
75   void AnisotropyAxis(double *V)
76   {
77           double rsq, y,z,y2,z2,ss;
78           do
79           {
80                   do
81                   {
82                           y = rand0_1() * 2.0 - 1.0;
83                           y2= rand0_1();
84                   }
85                   while (y2 > GaussDis(y, y_avg, y_sig));
86                   do
87                   {
88                           z = rand0_1() * 2.0 - 1.0;
89                           z2= rand0_1();
90                   }
```

```
91                  while (z2 > GaussDis(z, z_avg, z_sig));
92                  rsq = y*y + z*z;
93          }
94          while (rsq > 1.0);
95          ss = rand0_1() * 2.0 - 1.0;
96          V[0] = (ss/fabs(ss))*sqrt(1.0-rsq);
97          V[1] = y;
98          V[2] = z;
99  }
100 */
```

B.2 domainFinder.cpp

This is the programme used to generate the visualizations of 3D spin landscapes that are presented in the later parts of the thesis. This programme finds sublattices in a data set of spin positions and -orientations which are parallel, and analyzes size and orientation of all theses sublattices within one set. In conjunction with *gnuplot*, an entire set of spins can be visualized and via colorization by the programme, these images can be interpreted by the user even if thousands of spins in three dimensions are displayed at once.

Algorithm design and implementation in C++11 were done by the author.

```
1   #include <iostream>
2   #include <fstream>
3   #include <sstream>
4   #include <cmath>
5   #include <vector>
6   #include <array>
7   #include <string>
8   #include <algorithm>
9   #include <thread>
10  #include <chrono>
11
12  #include <stdlib.h>
13  //#include <omp.h>
14
15  using namespace std;
16
17  string data_file = "spin1000.dat"; // Input file name
18  string ofile_name = "showDomain.dat"; // gnuplot plotable coordinates of all vectors in a LARGE
        domain
19  string ofile2_name = "DomNegative.dat"; // same as above, only reduced to "interesting" domain
        PAIRS
20  string gnu_file = "colorDom.gnu"; // file name of gnu-script
21
22  const int max_N = 4*8*8*8; // Number of lines registered in input file == number of vectors
        considered
23
24  double latt_const = 1;
25  double POS[max_N][3]; // position vectors of sites
26  double VEC[max_N][3]; // orientation vectors of sites
27  bool forbidden[max_N]; // flag if a site has yet to be assigned to a domain
28  bool neg[max_N]; // flag if a site should be printed in negative file
29  double dist[max_N][max_N]; // distances btw sites
30  double MAX_DIST = 10.; // maximum distance btw to sites that are considered neighbours
31  //double MIN_CORR = -1.;
32  double MIN_CORR = cos(20*3.142/180.); // minimum scalar product btw to orientations that are
        considered belonging to same domain
33  double ACC;
34  double min_size = 80; // minimum size of a 'domain' that is significant enough to be stored in
        LARGEDOMAINS
35  //double perp_corr = 2.;
36  double perp_corr = cos(85*3.142/180.); // minimum scalar product btw to domains that are
        considered antiferromagnetic pairs
37  int dcounter; // stores number of domains that have been identified
38  int mem_counter; // stores number of member in one such domain
39  vector<vector<int>> DOMAINS; // stores every index vector of all small(!!) domains
40
41
42  //read data files and fill position and vector arrays
43  void read_data(string filename, int num)
44  {
45          ifstream data;
```

```
46          data.open(filename);
47          std::string line;
48
49          for (int i = 0; i < num*max_N; i++)
50          {
51
52                  getline(data, line);
53                  double col1, col2, col3, col4, col5, col6;
54
55                  istringstream ss(line);
56                  ss >> col1 >> col2 >> col3 >> col4 >> col5 >> col6;
57          }
58
59          int counter = 0;
60          for (int j = num*max_N; j < (num+1)*max_N; j++)
61          {
62
63                  getline(data, line);
64                  double col1, col2, col3, col4, col5, col6;
65
66                  istringstream ss(line);
67                  ss >> col1 >> col2 >> col3 >> col4 >> col5 >> col6;
68
69                  POS[counter][0] = col1;
70                  POS[counter][1] = col2;
71                  POS[counter][2] = col3;
72
73                  VEC[counter][0] = col4;
74                  VEC[counter][1] = col5;
75                  VEC[counter][2] = col6;
76                  counter++;
77          }
78
79          data.close();
80  }
81
82
83
84  // calculate distance btw two positions
85  double distance(int i, int j)
86  {
87          double x1, x2, y1, y2, z1, z2;
88
89          x1 = POS[i][0];
90          y1 = POS[i][1];
91          z1 = POS[i][2];
92
93          x2 = POS[j][0];
94          y2 = POS[j][1];
95          z2 = POS[j][2];
96
97          return sqrt(pow(x1-x2,2) + pow(y1-y2,2) + pow(z1-z2,2));
98  }
99
100 // scalar product btw two vectors
101 double scp(int i, int j)
102 {
103         double vx1, vx2, vy1, vy2, vz1, vz2;
104
105         vx1 = VEC[i][0];
106         vy1 = VEC[i][1];
107         vz1 = VEC[i][2];
108
109         vx2 = VEC[j][0];
110         vy2 = VEC[j][1];
111         vz2 = VEC[j][2];
112
113         return vx1*vx2+vy1*vy2+vz1*vz2;
114 }
115
116 // calculate center of mass of a given vector of sites
117 vector<double> POS_center(vector<int> sample)
118 {
119         double center[3];
120         vector<double> result;
121         center[0] = 0;
122         center[1] = 0;
123         center[2] = 0;
124
125         int len = sample.size();
126
127         for (int i = 0; i < len; i++)
```

```
128         {
129                 center[0]+=POS[sample[i]][0];
130                 center[1]+=POS[sample[i]][1];
131                 center[2]+=POS[sample[i]][2];
132         }
133
134         result.push_back(center[0]/len);
135         result.push_back(center[1]/len);
136         result.push_back(center[2]/len);
137
138         return result;
139
140 }
141
142 // calculate distance of two vectors (NOT indices)
143 double center_dist(vector<double> a, vector<double> b)
144 {
145         double x1, x2, y1, y2, z1, z2;
146
147         x1 = a[0];
148         y1 = a[1];
149         z1 = a[2];
150
151         x2 = b[0];
152         y2 = b[1];
153         z2 = b[2];
154
155         return sqrt(pow(x1-x2,2) + pow(y1-y2,2) + pow(z1-z2,2));
156 }
157
158 // ignore vectors that aren't unit vectors.
159 void test_data()
160 {
161         for (int i = 0; i < max_N; i++)
162                 if (scp(i,i) < 0.99 or scp(i,i) > 1.01)
163                 {
164                         VEC[i][0] = 0;
165                         VEC[i][1] = 0;
166                         VEC[i][2] = 0;
167
168                         cout << i+1 << " ignored!\n";
169                 }
170 }
171
172
173 // fill distance matrix
174 void dist_matrix()
175 {
176         //upper triangle matrix
177         for (int i = 0; i < max_N-1; i++)
178         {
179                 for (int j = i+1; j < max_N; j++)
180                         dist[i][j] = distance(i,j);
181         }
182
183         // symmetry i<->j
184         for (int i = 0; i < max_N-1; i++)
185         {
186                 for (int j = i+1; j < max_N; j++)
187                         dist[j][i] = dist[i][j];
188         }
189 }
190
191 // return list of neighbours within max_dist radius
192 vector<int> neighbours(int i, double max_dist)
193 {
194         vector<int> neighb;
195         for (int j = 0; j < max_N; j++)
196         {
197                 if (dist[i][j] <= max_dist)
198                 {
199                         //cout << i << " NB von " << j << endl;
200                         neighb.push_back(j);
201                 }
202
203         }
204         return neighb;
205 }
206
207 // Calculate avg direction of a domain given its index in DOMAINS
208 vector<double> vec_mean(int domain)
209 {
```

```
210         double new_mean[3];
211         new_mean[0] = 0;
212         new_mean[1] = 0;
213         new_mean[2] = 0;
214
215         vector<double> result;
216
217         int len = DOMAINS[domain].size();
218
219         for (int i = 0; i < len; i++)
220         {
221                 new_mean[0]+=VEC[DOMAINS[domain][i]][0];
222                 new_mean[1]+=VEC[DOMAINS[domain][i]][1];
223                 new_mean[2]+=VEC[DOMAINS[domain][i]][2];
224         }
225         /*
226         new_mean[0] = pow(new_mean[0], 1./len);
227         new_mean[1] = pow(new_mean[1], 1./len);
228         new_mean[2] = pow(new_mean[2], 1./len);
229         */
230         double mod = sqrt(pow(new_mean[0],2)+pow(new_mean[1],2)+pow(new_mean[2],2));
231         for (int i = 0; i < 3; i++)
232                 result.push_back(new_mean[i]/mod);
233
234         return result;
235
236 }
237
238 // scalar product of a given vector old and a vector at index new_ind
239 double scp_vec_ind(vector<double> old, int new_ind)
240 {
241         double vx1, vx2, vy1, vy2, vz1, vz2;
242
243         vx1 = old[0];
244         vy1 = old[1];
245         vz1 = old[2];
246
247         vx2 = VEC[new_ind][0];
248         vy2 = VEC[new_ind][1];
249         vz2 = VEC[new_ind][2];
250
251         return vx1*vx2+vy1*vy2+vz1*vz2;
252 }
253
254 // scalar product of two directly given vectors
255 double dom_dom(vector<double> a, vector<double> b)
256 {
257         double vx1, vx2, vy1, vy2, vz1, vz2;
258
259         vx1 = a[0];
260         vy1 = a[1];
261         vz1 = a[2];
262
263         vx2 = b[0];
264         vy2 = b[1];
265         vz2 = b[2];
266
267         return vx1*vx2+vy1*vy2+vz1*vz2;
268 }
269
270
271 // return list of "good" neighbours among neighbours
272 vector<int> good_neighbours(vector<double> dom_mean, vector<int> neighb, double min_corr, int
        domain)
273 {
274         int n;
275
276         int len = neighb.size();
277         vector<int> goodNs;
278         for (int i=0; i<len; i++)
279         {
280                 n = neighb[i];
281                 //if (scp(n, j) >= min_corr and not forbidden[n])
282                 if (scp_vec_ind(dom_mean, n) >= min_corr and not forbidden[n])
283                 {
284
285                         forbidden[n] = true;
286                         DOMAINS[domain].push_back(n);
287                         //cout << n << " found as good neighbor\n";
288                         goodNs.push_back(n);
289
290                 }
```

```
291         }
292         // flag if a site is isolated = w/o a single good neighbour = break recursive loop in
                findDomain()
293         if (goodNs.size() == 0)
294                 goodNs.push_back(-1);
295 
296         return goodNs;
297 }
298 
299 
300 // Central algorithm: Recursively, find 'good neighbours' among connected and not yet domain-
        associated vectors
301 void findDomain(int start, double max_dist, double min_corr, int domain)
302 {
303         vector<int> goodNs;
304         vector<int> neighb;
305         vector<double> dom_mean;
306 
307         int gn_size;
308         int new_start;
309 
310         // continuous update of what would be considered a good neighbour from avg(!!!)
                orientation of current domain members
311         if (DOMAINS[domain].size() >= 1)
312                 dom_mean = vec_mean(domain);
313         else
314         {
315                 dom_mean.push_back(VEC[start][0]);
316                 dom_mean.push_back(VEC[start][1]);
317                 dom_mean.push_back(VEC[start][2]);
318         }
319 
320         neighb = neighbours(start, max_dist);
321         goodNs = good_neighbours(dom_mean, neighb, min_corr, domain);
322         gn_size = goodNs.size();
323 
324         for (int i = 0; i < gn_size; i++)
325         {
326                 new_start = goodNs[i];
327                 if (new_start != -1)
328                 {
329                         //cout << "start findDomain from " << new_start << endl;
330 
331                         findDomain(new_start, max_dist, min_corr, domain);
332                 }
333         }
334 
335 }
336 
337 // print a single vector's coordinates for gnuplot to ofile_name (!!)
338 void print_coord(int i)
339 {
340         ofstream ofile;
341         ofile.open(ofile_name, ios::app);
342 
343         double x, y, z, vx, vy, vz;
344         double scale = 0.5;
345 
346         vx = scale * VEC[i][0];
347         vy = scale * VEC[i][1];
348         vz = scale * VEC[i][2];
349 
350         x = POS[i][0] - 0.5*vx;
351         y = POS[i][1] - 0.5*vy;
352         z = POS[i][2] - 0.5*vz;
353 
354 
355         ofile << x << "\t" << y << "\t" << z << "\t" << vx << "\t" << vy << "\t" << vz << endl;
356 
357         ofile.close();
358 
359 }
360 
361 // print a single vector's coordinates for gnuplot to ofile2_name (!!)
362 void print_coord_alt(int i)
363 {
364         ofstream ofile2;
365         ofile2.open(ofile2_name, ios::app);
366 
367         double x, y, z, vx, vy, vz;
368 
369         vx = VEC[i][0];
```

```
370         vy = VEC[i][1];
371         vz = VEC[i][2];
372
373         x = POS[i][0] - 0.5*vx;
374         y = POS[i][1] - 0.5*vy;
375         z = POS[i][2] - 0.5*vz;
376
377
378         ofile2 << x << "\t" << y << "\t" << z << "\t" << vx << "\t" << vy << "\t" << vz << endl;
379
380         ofile2.close();
381
382 }
383
384 // generate the complete gnuplot script to visualize all large domains
385 void gnu_script(vector<vector<int>> data)
386 {
387         ofstream gnu;
388         gnu.open(gnu_file);
389
390         vector<int> param;
391
392         int lines = data.size();
393
394         for (int i = 0; i < lines; i++)
395                 param.push_back(data[i].size());
396
397
398         gnu << "reset" << endl;
399         gnu << "set␣view␣equal␣xyz" << endl;
400         gnu << "set␣title␣''" << endl;
401         gnu << "splot␣";
402
403         char c = '"';
404
405
406         int von = 0;
407         int bis = param[0]-1;
408
409         for (int i = 0; i < lines; i++)
410         {
411
412                 gnu << "'" << ofile_name << "'␣u␣1:2:3:4:5:6␣every␣1::";
413                 gnu << von << "::" << bis << "␣with␣vectors␣";
414                 gnu << "title␣" << c << "Sublattice␣" << i+1 << c << ",␣";
415
416                 von += param[i];
417                 bis += param[i+1];
418         }
419
420         gnu << endl;
421
422         gnu.close();
423 }
424
425
426
427 // MAIN FUNCTION
428 int main()
429 {
430
431         int number;
432
433         cout << "Which␣part?␣";
434         cin >> number;
435         // Some initializations.
436         dcounter=0;
437
438         ofstream ofile;
439         ofstream ofile2;
440         ofstream gnu;
441         ofile.open(ofile_name, ios::trunc);
442         ofile.close();
443         ofile2.open(ofile2_name, ios::trunc);
444         ofile2.close();
445         gnu.open(gnu_file, ios::trunc);
446         gnu.close();
447
448         cout << "Output␣files␣reset!\n";
449
450         vector<vector<int>> LARGEDOMAINS;
451         vector<vector<double>> LARGEDOMAINS_AXIS;
```

```
452
453         // read and test input file
454
455
456         read_data(data_file, number);
457         test_data();
458         cout << "data_file_read...\n";
459
460
461         // Every spin is eligible for a domain and all distances are stored
462         for (int i = 0; i < max_N; i++)
463         {
464                 neg[i] = true;
465                 forbidden[i] = false;
466         }
467         dist_matrix();
468         cout << "distances_calculated...\n";
469
470         // Necessary initialization of vectors
471         vector<int> placeholder;
472         DOMAINS.push_back(placeholder);
473
474         //int domainSize;
475
476         // Call the central algorithm until every site has been assigned to an element of
                DOMAINS or has been found as isolated
477         for (int i = 0; i < max_N; i++)
478         {
479                 //cout << "Test " << i << endl;
480                 if (not forbidden[i])
481                 {
482                         DOMAINS.push_back(placeholder);
483                         dcounter++;
484                         findDomain(i, MAX_DIST, MIN_CORR, dcounter);
485                 }
486         }
487
488         cout << "How_accurate?_";
489         cin >> ACC;
490
491         // From the partitioning, remove deviating spins and find LARGE domains (minimum number
                of members) and store them together with their orientation in LARGEDOMAINS
492         double ratio = 0;
493         int DCOUNTER = 0;
494
495         double x, y, z;
496         vector<double> dom_mean;
497         int DOMsize = DOMAINS.size();
498
499         for (int i = 0; i < DOMsize; i++)
500         {
501                 vector<int> v_candidate = DOMAINS[i];
502                 int len_candidate = v_candidate.size();
503                 dom_mean = vec_mean(i);
504
505                 vector<int> v;
506
507                 // remove/don't add spins that deviate too much from mean direction
508                 for (int j = 0; j < len_candidate; j++)
509                 {
510                         if (scp_vec_ind(dom_mean, v_candidate[j]) >= ACC * sqrt((1+MIN_CORR)/2))
511                         {
512                                 v.push_back(v_candidate[j]);
513                                 neg[v_candidate[j]] = false;
514
515                         }
516                 }
517
518                 int len = v.size();
519
520                 // find best sets with significant size
521                 if (len >= min_size)
522                 {
523                         LARGEDOMAINS.push_back(v);
524                         DCOUNTER++;
525                         ratio += len;
526
527                         LARGEDOMAINS_AXIS.push_back(dom_mean);
528                         x = dom_mean[0];
529                         y = dom_mean[1];
530                         z = dom_mean[2];
531
```

```
532
533
534                     cout << "Sublattice " << DCOUNTER << " of size " << len << " w/ 
                            direction " << x << " "  << y << " "  << z << endl;
535
536                     for (int j = 0; j < len; j++)
537                             print_coord(v[j]);
538                     //ofile << endl;
539
540             }
541
542     }
543
544
545
546
547     ratio /= max_N;
548
549     cout << DCOUNTER << " sublattices found!\n";
550     cout << "Sublattice in/out ratio = " << ratio << endl;
551
552
553     // Among the significant domains, find "interesting" domains, i.e. w/ anti-parallel or
            perpendicular orientation
554
555
556     int large_size = LARGEDOMAINS.size();
557
558
559     int pair_counter = 0; // count all pairs (perp, parall, anti-parall)
560     int anti_counter = 0; // count anti-pairs
561     int red_counter;  // count almost parallel pairs
562     int RED_COUNTER = 0;
563     if (large_size >= 2)
564     {
565             for (int i = 0; i < large_size-1; i++)
566             {
567                     red_counter = 0;
568
569                     for (int j = i+1; j < large_size; j++)
570                     {
571                             vector<double> a = LARGEDOMAINS_AXIS[i];
572                             vector<double> b = LARGEDOMAINS_AXIS[j];
573                             vector<int> v1 = LARGEDOMAINS[i];
574                             vector<int> v2 = LARGEDOMAINS[j];
575                             //if (center_dist(POS_center(v1), POS_center(v2)) <= min_size*
                                    MAX_DIST)
576                             if (dom_dom(a,b) <= - ACC * sqrt(0.5*(MIN_CORR+1)) or abs(
                                    dom_dom(a,b)) <= perp_corr)
577                             {
578                                     pair_counter++;
579                                     cout << i+1 << " vs " << j+1 << " interesting!";
580                                     if (dom_dom(a,b) <= - ACC * sqrt(0.5*(MIN_CORR+1)))
581                                     {
582                                             cout << " (anti-parr)\n";
583                                             anti_counter++;
584                                     }
585                                     else
586                                             cout << " (perpendicular)\n";
587
588                                     /*
589                                     int len1 = v1.size();
590                                     int len2 = v2.size();
591                                     for (int k = 0; k < len1; k++)
592                                             print_coord_interesting(v1[k]);
593                                     for (int k = 0; k < len2; k++)
594                                             print_coord_interesting(v2[k]);
595                                     */
596                             }
597                             else
598                             {
599                                     if(dom_dom(a,b) >= + ACC * sqrt(0.5*(MIN_CORR+1)))
600                                     {
601                                             pair_counter++;
602                                             cout << i+1 << " vs " << j+1 << " interesting!";
603                                             cout << " (almost_parr)\n";
604                                             red_counter++;
605
606                                     }
607
608                             }
609
```

```
610                                     RED_COUNTER += red_counter;
611
612
613                             }
614                     }
615
616                     if (2* anti_counter == large_size-RED_COUNTER)
617                             cout << "!!! Each sublattice has exact antiparallel partner !!!\n";
618             }
619
620
621
622             // In case any large domains were found, produce the complete gnuscript for colored
                    visualization
623             cout << pair_counter << " interesting pairs.\n";
624
625
626             if (large_size >= 1)
627             {
628                     gnu_script(LARGEDOMAINS);
629
630                     for (int i = 0; i < max_N; i++)
631                     {
632                             if (neg[i])
633                                     print_coord_alt(i);
634                     }
635             }
636
637             return 0;
638
639     }
```

C References

[1] J.-O. Andersson, C. Djurberg, T. Jonsson, P. Svedlindh, and P. Nordblad. Monte carlo studies of the dynamics of an interacting monodispersive magnetic-particle system. *Phys. Rev. B*, 56:13983–13988, Dec 1997.

[2] International Mineralogical Association. IMA List of Minerals. `http://ima-cnmnc.nrm.se/IMA_Master_List_%282017-11%29.pdf`.

[3] Subhankar Bedanta, Oleg Petracic, and Wolfgang Kleemann. *Chapter 1 - Supermagnetism*, volume 23 of *Handbook of Magnetic Materials*. Elsevier, 2015.

[4] Sabrina Disch, Erik Wetterskog, Raphaël P. Hermann, German Salazar-Alvarez, Peter Busch, Thomas Brückel, Lennart Bergström, and Saeed Kamali. Shape induced symmetry in self-assembled mesocrystals of iron oxide nanocubes. *Nano Letters*, 11(4):1651–1656, 2011. PMID: 21388121.

[5] Zhendong Fu, Yinguo Xiao, Artem Feoktystov, Vitaliy Pipich, Marie-Sousai Appavou, Yixi Su, Erxi Feng, Wentao Jin, and Thomas Brückel. Field-induced self-assembly of iron oxide nanoparticles investigated using small-angle neutron scattering. *Nanoscale*, 8:18541–18550, 2016.

[6] Axel Gelfert and Wolfgang Nolting. The absence of finite-temperature phase transitions in low-dimensional many-body models: a survey and new results. *Journal of Physics: Condensed Matter*, 13(27):R505, 2001.

[7] Giuseppe Grosso and Giuseppe Pastori Parravicini. *Chapter 2 - Geometrical Description of Crystals: Direct and Reciprocal Lattices*. Academic Press, Amsterdam, second edition edition, 2014.

[8] Cherie R. Kagan, Efrat Lifshitz, Edward H. Sargent, and Dmitri V. Talapin. Building devices from colloidal quantum dots. *Science*, 353(6302), 2016.

[9] J M Kosterlitz and D J Thouless. Ordering, metastability and phase transitions in two-dimensional systems. *Journal of Physics C: Solid State Physics*, 6(7):1181, 1973.

[10] David Landau and Kurt Binder. *A Guide to Monte Carlo Simulations in Statistical Physics*. Cambridge University Press, New York, NY, USA, 2005.

[11] Bishop Kyle J. M., Wilmer Christopher E., Soh Siowling, and Grzybowski Bartosz A. Nanoscale forces and their uses in self-assembly. *Small*, 5(14):1600–1630, 2009.

[12] Gil Markovich. Magneto-transport and magnetization dynamics in magnetic nanoparticle assemblies. *MRS Bulletin*, 38(11):939–944, 2013.

[13] G. Marsaglia and T. A. Bray. A convenient method for generating normal variables. *SIAM Review*, 6(3):260–264, 1964.

[14] N. D. Mermin and H. Wagner. Absence of ferromagnetism or antiferromagnetism in one- or two-dimensional isotropic heisenberg models. *Phys. Rev. Lett.*, 17:1133–1136, Nov 1966.

[15] Nicholas Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[16] Lars Onsager. Electric moments of molecules in liquids. *Journal of the American Chemical Society*, 58(8):1486–1493, 1936.

[17] F. X. Redl, K.-S. Cho, C. B. Murray, and S. O'Brien. Three-dimensional binary superlattices of magnetic nanocrystals and semiconductor quantum dots. *Nature*, 423:968 EP –, Jun 2003.

[18] L.-M. Wang, O. Petracic, E. Kentzinger, U. Rucker, M. Schmitz, X.-K. Wei, M. Heggen, and Th. Brückel. Strain and electric-field control of magnetism in supercrystalline iron oxide nanoparticle-batio3 composites. *Nanoscale*, 9:12957–12962, 2017.

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den