Design and Development of a GPU-accelerated Micromagnetic Simulator

D i s s e r t a t i o n

zur Erlangung des akademischen Grades

Dr. rer. nat.

an der Fakultät für Mathematik, Informatik und Naturwissenschaften
der Universität Hamburg

eingereicht beim
Fach-Promotionsausschuss Informatik von

Gunnar Selke

aus Hamburg

Juli 2013

– korrigierte Fassung –

Gutachter	Prof. Dr. Dietmar P. F. Möller PD Dr. Guido Meier Prof. Dr. Stephan Olbrich
Datum der Disputation	2. April 2014

Abstract

Micromagnetic simulators are important software tools to investigate ferromagnetic nano- and microstructures. In this work the fast micromagnetic simulator *MicroMagnum* is developed, which runs on CPUs as well as on graphics processing units (GPUs). Its high performance on GPUs allows the investigation of large simulation problems that could not be treated before by previous software. From the beginning MicroMagnum was developed with strong feedback by the users to obtain a high usability. Fast algorithms using the finite-difference method are presented that implement the micromagnetic model efficiently. By employing graphics processing units, a speedup of up to two orders of magnitude is achieved.

Modern graphics processors allow general programming on graphic processing units. GPUs have many compute cores and their programming model is inherently parallel. Thus the algorithms have to be parallelized before they are ported to the GPU architecture. Several parallel software patterns are adapted to the GPU architecture and applied to the implementation of the micromagnetic model.

The software requirements of performance, correctness, extendability, usability, portability, and maintainability are desired properties. The simulator is written in Python and C++. To achieve high software extendability, the dynamic scripting language Python is used for the implementation of the non-speed-critical software parts. Likewise the users of the simulator write their simulation scripts in Python. A module system that manages the dependencies of the model variables of the micromagnetic model is presented. An abstraction layer that hides the implementation details of the CPU and GPU algorithms and data representations is developed. This speed-critical software layer is written in C++ and CUDA C.

Several benchmarks that compare the performance of the CPU and GPU routines are performed.

As a use case, non-linear vortex-core dynamics are investigated using simulations performed by MicroMagnum.

Zusammenfassung

Mikromagnetische Simulatoren sind wichtige Werkzeuge, um ferromagnetische Mikro- und Nanostrukturen zu untersuchen. In dieser Arbeit wird MicroMagnum, ein performanter mikromagnetischer Simulator, entwickelt. Der Simulator läuft sowohl auf CPUs als auch auf Grafikprozessoren (GPUs). Die hohe Performanz auf GPUs erlaubt die Durchführung von komplexen Simulationen, die voher nicht berechenbar waren. Von Anfang an wurde MicroMagnum unter ständigem Feedback seiner Benutzer entwickelt, um eine hohe Benutzbarkeit zu garantieren.

Schnelle Algorithmen, die das mikromagnetische Modell mittels der finite Differenzen Methode berechnen, werden präsentiert. Durch die Nutzung von Grafikprozessoren wird gegenüber der CPU eine Geschwindigkeitssteigerung von bis zu zwei Größenordnungen erreicht.

Moderne Grafikkarten erlauben die freie Programmierung ihrer Grafikprozessoren. Die Prozessoren haben viele Rechenkerne, und ihr Programmiermodell ist von Natur aus parallel. Dadurch müssen die Algorithmen parallelisiert werden, bevor sie auf die GPU-Architektur portiert werden. Mehrere parallele Software-Entwurfsmuster werden auf die GPU-Architektur adaptiert und auf das mikromagnetische Modell angewendet.

Die Software-Qualitätskriterien Performanz, Korrektheit, Erweiterbarkeit, Benutzbarkeit, Portabilität und Wartbarkeit sind erwünschte Eigenschaften. Der Simulator wurde in Python und C++ entwickelt. Um eine gute Software-Erweiterbarkeit zu erreichen, wird die dynamisch typisierte Python-Skriptprogrammiersprache für die Implementation der Geschwindigkeitsunkritischen Programmteile verwendet. Ebenso schreiben die Benutzer der Software ihre Simulationsprogramme in Python. Ein Modulsystem wird präsentiert, welches die Abhängigkeiten zwischen den Modellvariablen des mikromagnetischen Modells verwaltet. Eine Abstraktionsschicht enthält die Implementationsdetails der CPU- und GPU-algorithmen und Datenstrukturen. Diese Performanzkritischen Routinen sind in C++ und CUDA C implementiert.

Mehrere Benchmarks werden durchgeführt, um die Geschwindigkeit auf CPUs und GPUs zu vergleichen.

Als Anwendungsfall wird die nicht-lineare Dynamik von magnetischen Vortizes durch MicroMagnum untersucht.

Contents

Introduction

In recent decades simulations of ferromagnetic nanostructures received a great deal of interest in basic research and industry. In physical experiments, simulations are used to explain the observed magnetization dynamics. Technology companies employ micromagnetic simulations to develop storage devices based on ferromagnetic properties. Ferromagnets possess magnetic moments that lead to a permanent magnetization. The magnetic moments can be manipulated by external magnetic fields and electrical currents. Due to these properties ferromagnets are suitable for the representation of the binary state inside storage devices. To make the storage devices small enough, ferromagnets with spatial dimensions ranging from some nanometers up to some microns, so-called ferromagnetic microstructures, are required. If a current is applied through ferromagnets, the electrical resistance depends on the orientation of the magnetic moments. These so-called magnetoresistive effects are used to detect the orientation of the magnetic moments. For example, hard-disk read heads exploit magnetoresistive effects. In contrast to hard magnetic materials like iron, cobalt, and nickel, the magnetization of soft magnetic materials can be manipulated by small amplitudes of excitation. A prominent example of a soft magnetic material is Permalloy ($Ni_{80}Fe_{20}$), which has permanent magnetic moments that are easily manipulated. In recent years a large number of memory concepts that contain ferromagnetic microstructures have been proposed. The memory concepts differ in how they represent binary states and how these states are read and manipulated. In the nineties the Magnetoresistive Random Access Memory (MRAM) was invented. It

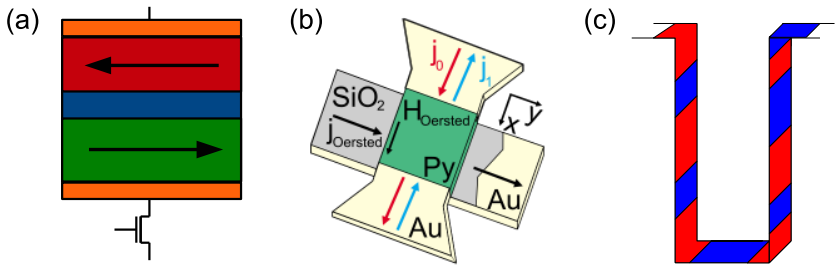


Figure 0.1: (a) Magnetoresistive Random Access Memory (MRAM) cell, (b) Vortex Memory cell (VRAM) (Reprinted with permission from Ref. [1]. Copyright 2008, American Institute of Physics.), (c) Racetrack memory.

was first sold commercially in 2007. A cell of an MRAM[2], see Fig. 0.1 (a), consists of a stack of ferromagnetic and non-ferromagnetic layers. The binary state is represented by the orientation of the magnetic moments in the so-called free layer. In a free layer the binary state is stable but can be easily manipulated by a current or a magnetic field. The binary state is read out by applying a low current through the layers. The electrical resistance depends on the orientation of the magnetization due to the Giant Magneto Resistance or Tunneling Magneto Resistance. For example, a low resistance represents a

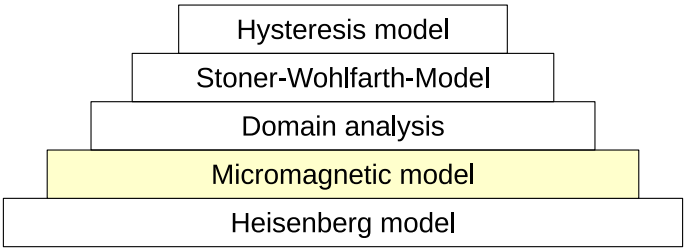


Figure 0.2: Ferromagnetic models sorted by level of abstraction. The applicability of the models ranges from the atomistic level at the bottom to the macroscopic level at the top. The micromagnetic model is highlighted. (Figure adapted from Ref. [6]).

binary zero, and a high resistance represents a binary one. Another new memory device is the proposed race track memory[3, 4]. Here multiple bits are stored magnetically along a ferromagnetic nanowire, see Fig. 0.1 (c). A bit-wise one or zero is represented by the polarization of a magnetic domain on a section of the nanowire. A spin-polarized current is used to shift the magnetic domains around the nanowire in order to position one bit under the fixed read and write heads. This way a shift register is realized. The nanowire may be extended into the third dimension, thus potentially allowing a very high storage density. A Vortex random access memory stores the binary state by the magnetization pattern of a vortex[1, 5], see Fig. 0.1 (b). The binary state is modified by a spin-polarized and current and its concomitant Oersted field. The cells can be arranged in an array to realize a word and a bit line like in a MRAM. In comparison to an MRAM less energy is required for writing a bit and the writing process is faster.

The physical origin of the representation, writing, and reading of binary states in storage concepts can be acquired by experiments, theoretical considerations, and simulations. Experiments can be costly and time consuming and theoretical models are often strongly simplified. Thus, for the investigation of ferromagnetic nano- and microstructures, micromagnetic simulations have become a powerful tool to predict their magnetic behavior. In recent years, advancements of computers in computation speed and memory size allowed increasingly realistic simulations. However, quantum-mechanical simulations can still be performed for a few number of atoms only. In 1935, Landau and Lifshitz introduced the micromagnetic model. In this model the magnetic moments are replaced by classical magnetization vectors, and a ferromagnetic body is represented by a continuous magnetization vector field. The magnetization vectors interact by effective fields. An equation of motion, the so-called Landau-Lifshitz-Gilbert equation, describes the magnetization dynamics. For most problems, the nonlinear Landau-Lifshitz-Gilbert equation can only be solved numerically. In micromagnetic simulations, the ferromagnetic body is discretized by simulation cells. Each simulation cell contains a local magnetization and effective field. Typical simulations have millions of simulation cells. Thus fast algorithms with high computation power are required.

Common micromagnetic simulations on CPUs run sequentially. Often, in order to

achieve acceptable computation times, simulation elements with a smaller size than the experimental samples are chosen. Simulations with large simulation elements in the micrometer range can take days to months to complete. It is therefore important to develop numerical methods for the simulation that are as efficient as possible.

During the last two decades several micromagnetic simulators for use on commodity hardware have been developed, see Tab. 0.1. They are distinguished by their use of a finite difference method[7, 8, 9], a hybrid finite/boundary element method[10, 11, 12, 13], a fast multipole[14, 15, 16, 17], or another numerical method to compute the Landau-Lifshitz-Gilbert equation including the effective fields.

The simulators that employ the finite difference method usually use rectangular meshes for the spatial discretization. Due to the widely used Fourier transform method for calculating magnetic fields they are typically very fast, because efficient software libraries to compute fast Fourier transforms are available. However, the regular discretizing mesh can be disadvantageous when non-rectangular and/or irregular samples are investigated[18]. The most established micromagnetic simulator that employs a finite difference scheme is the Object Oriented MicroMagnetic Framework (OOMMF)[19]. It runs on multi-core processors and is implemented in C++ and the Tcl scripting language.

In contrast, the finite-element simulators allow more flexible meshes. Additionally, an adaptive mesh refinement is possible[20]. Meshes with tetrahedral elements are most widely used. In the finite-element method the effective field is numerically approximated by the solution of a linear equation system[13]. The solution of linear equation systems is a common problem in the field of computational mathematics and there exist many numerical methods and readily usable software implementations. Due to the use of standard algorithms the computation can be executed parallely. In comparison to the finite difference method, the finite element method still requires more memory and computation time. Thus for rectangular-shaped geometries usually a finite difference simulator is preferred. Established micromagnetic simulators that employ a finite-element scheme are NMag[21, 22] and MagPar[23], both of which use tetrahedral meshes. They are implemented in the C++, OCaml and Python programming languages.

In recent years, general purpose programming on graphics processing units (GPGPU) has become available. Graphics processors are used as accelerators for efficient numerical computations due to their high number of cores and high memory bandwidth. Their many cores enable them to compute highly in parallel. GPUs have been used to speed up micromagnetic simulations with the finite-element method[30] and the non-uniform grid interpolation method[31]. During the writing of this thesis micromagnetic simulators using the finite difference method on GPUs, both open-source and commercial, have emerged[26, 24].

In this thesis the design and implementation of the MicroMagnum simulator, which is available online[32] as an open source project[33], is presented. It is a finite difference micromagnetic simulator that solves the Landau-Lifshitz-Gilbert equation parallely on

Name	License	Parallel execution	Implementation language	Scripting language	Online
Finite difference method simulators					
GPMagnet[24]	comm.	GPU	n/a	n/a	(1)
JaMM	PD	no	Java	n/a	(2)
LLG	comm.	n/a	n/a	n/a	(3)
M^3S[25]	n/a	no	Matlab	Matlab	n/a
MicroMagnum	GPL	GPU	C++, Python	Python	(4)
MicroMagus	comm.	n/a	n/a	n/a	(5)
muMax[26]	GPL	GPU	Go	yes	(6)
muMax 2	GPL	GPU	Go	yes	(7)
OOMMF[19]	PD	SMP	Tcl, C++	Tcl	(8)
RKMAG	n/a	no	FORTRAN	n/a	(9)
YAMMS	GPL	no	Java	JavaScript	(10)
Finite element method simulators, including hybrid methods					
MagFEM3d	GPL	no	FORTRAN	-	(11)
MagPar[23]	GPL	MPI	C++	-	(12)
NMag[21, 22]	GPL	MPI	OCaml, Python	Python	(13)
Fast multipole method / multigrid method simulators					
AlaMag[27]	GPL	no	C++	-	(14)
FastMag[28]	n/a	GPU	n/a	n/a	(15)

Table 0.1: Overview of different micromagnetic simulation programs including MicroMagnum, which is described in this work. For each simulator the license (public domain (PD), GNU General Public License[29] (GPL), or a commercial license), parallel execution support (Symmetric multiprocessing (SMP), on graphics processing units (GPU), or using parallel processes using the message passing interface (MPI)), the implementation and scripting languages and the web site, if available, is listed. (Table adapted and extended from Ref. [25].)

[1] http://www.goparallel.net/index.php/gp-software
[2] http://jamm.uno.edu/
[3] http://llgmicro.home.mindspring.com/
[4] http://micromagnum.informatik.uni-hamburg.de
[5] http://www.micromagus.de
[6] http://code.google.com/p/mumax
[7] http://code.google.com/p/mumax2
[8] http://math.nist.gov/oommf/
[9] http://www.rkmag.com
[10] http://github.com/c-abird/yamms_core
[11] http://magfem3d.sourceforge.net/
[12] http://www.magpar.net/
[13] http://nmag.soton.ac.uk/nmag/
[14] http://faculty.mint.ua.edu/~visscher/AlaMag/
[15] http://cem.ucsd.edu/index_files/fastmag.html

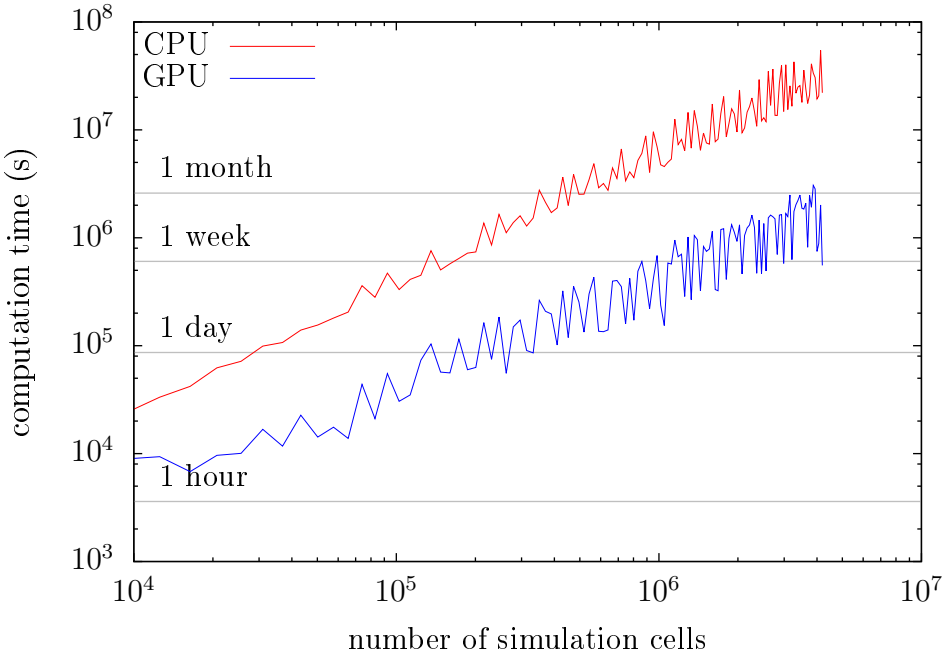


Figure 0.3: Computation time of MicroMagnum for 10^7 simulation steps during a micromagnetic simulation using one core of an Intel Xeon X5650 CPU (red line) and an Nvidia M2050 graphics processor (blue line) in dependence of the number of simulation cells. A speedup of up to 40 is reached compared to single-threaded computation on the CPU.

Nvidia graphics processing units. The simulator is implemented using the programming languages C++[34], CUDA C[35] and Python[36]. Due to the transfer of the program from the CPU to the GPU, a considerable speedup can be reached. Figure 0.3 gives a first impression of the speed that is gained in MicroMagnum; detailed benchmark results are presented in section 7.3. Compared to computations on a modern CPU, a speedup of up to 40 is reached. The actual speedup depends on the problem size (the number of simulation cells and whether the discretization mesh is two- or three-dimensional) and the selected numerical floating point precision of either 32-bit or 64-bit on the GPU. Some kinds of simulations require 64 bit precision, and some kinds tolerate 32-bit precision in order to obtain useful results.

This thesis is organized as follows: There are three main parts. The first part describes the micromagnetic model and its finite difference discretization. Fast algorithms for the numerical solution are also presented. The second part deals with the parallel computation of the presented algorithms on multi-core CPUs as well as graphics processing units. The third part describes the development of a micromagnetic simulation framework MicroMagnum. Here some aspects of the software development are discussed. As a use-case, the simulator is applied to investigate nonlinear vortex motion[37], which could be utilized in vortex random access memory.

Part I

Micromagnetic Model and its Discretization

In this part the micromagnetic model and its extensions for electrical and spin currents is introduced. Its finite-difference discretization is presented. Finally, fast algorithms to compute the micromagnetic model are presented.

1 Micromagnetic Model

In ferromagnetic materials, the spins of adjacent electrons align in parallel, leading to intrinsic magnetic moments. The parallel alignment remains even in the absence of an external excitation such as a magnetic field which causes a permanent magnetization. To describe the spin dynamics, the atomistic Heisenberg model[38] was introduced. In this model the electron spins are located on the crystal lattice. Usually this spatial resolution is too high for simulations. In 1935, the phenomenological micromagnetic model was introduced by Landau and Lifshitz[39] to describe larger systems. In this model the electron spins are replaced by a classical magnetization function. The temporal evolution of the magnetization is given by the Landau-Lifshitz-Gilbert equation. It can be used to investigate magnetization configurations like domain walls[40] and magnetic vortices[41, 42].

In the micromagnetic model the individual magnetic moments are represented by a classical field of magnetization vectors

$$\vec{M}(\vec{r}) = \frac{\sum \mu_{\mathrm{B}}}{V} \tag{1.1}$$

with the Bohr magneton $\mu_{\mathrm{B}} = e\hbar/(2m_e) = 9.27 \cdot 10^{-24}\mathrm{Am}^2$. This field gives the volume density of the magnetic moments of a material and thus has the unit A/m. The length of the magnetization vectors is the saturation magnetization $M_{\mathrm{s}} = |\vec{M}|$ and is a measure for the number of elementary magnetic moments μ_B per volume of a material. In the following the magnetization $\vec{M}$ field is a function of the position $\vec{r}$ at time t.

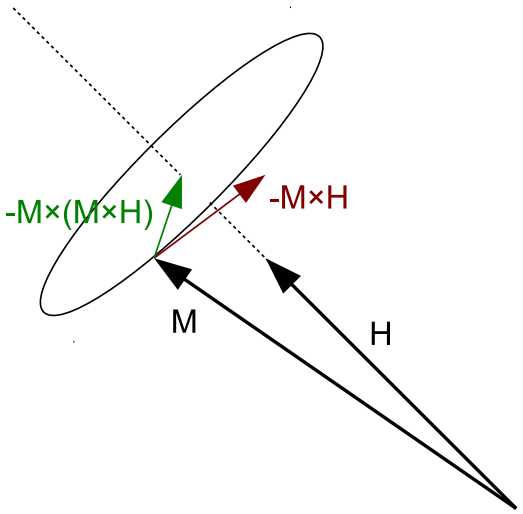


Figure 1.1: Gyration and damping of a magnetization vector as described by the Landau-Lifshitz-Gilbert equation. The green arrow indicates the damping term and the red arrow indicates the gyration term.

1.1 Landau-Lifshitz-Gilbert equation

The dynamical behaviour of the magnetization of a ferromagnet is described by the Landau-Lifshitz-Gilbert equation[39, 43] (LLGE),

$$\frac{\mathrm{d}\vec{M}}{\mathrm{d}t} = -\gamma\vec{M} \times \vec{H}_{\mathrm{tot}} - \frac{\gamma\alpha}{M_{\mathrm{s}}}\vec{M} \times (\vec{M} \times \vec{H}_{\mathrm{tot}}). \tag{1.2}$$

The magnetization vectors interact via the total field $\vec{H}_{\mathrm{tot}}$, which depends on the position $\vec{r}$ and time t. The material parameter α is the phenomenological Gilbert damping constant. The two terms on the right hand side, i.e. the gyration term and the damping term, cause a damped gyration around the total field, see Fig. 1.1. The damping term was added by T. L. Gilbert [44, 45] to account for dissipation effects. Both terms include the gyromagnetic ratio γ. The second term is proportional to the phenomenological Gilbert damping parameter α. Without external excitation, the magnetization vectors eventually align parallel to the total field.

1.2 Effective field terms

In the basic micromagnetic model the total energy[6] of a ferromagnet

$$\begin{aligned} E &= E_{\mathrm{exch}} + E_{\mathrm{demag}} + E_{\mathrm{aniso}} + E_{\mathrm{ext}} \\ &= \mu_0 \int \left[A(\vec{\nabla}\vec{M})^2 + \frac{1}{2}\vec{H}_{\mathrm{demag}}\vec{M} + k\vec{a}\vec{M} + \vec{H}_{\mathrm{ext}}\vec{M} \right] \mathrm{d}V \end{aligned} \tag{1.3}$$

is a sum of the exchange energy, the demagnetization energy, the anisotropy energy, and the energy of the external field. The magnetic permeability of vacuum μ_0 is a natural constant. The exchange energy E_{exch} with the exchange constant A is the result of the quantum-mechanical interaction between neighboring spins. It causes adjacent magnetization vectors to align parallel. The demagnetization energy $\vec{H}_{\text{demag}}$ is the interaction energy of the magnetic field that is generated by the magnetization vectors themselves. This long-range interaction causes the magnetic moments to prefer an anti parallel alignment. The name stems from the fact that the demagnetization field causes the total magnetic moment to reduce. The anisotropy energy E_{aniso} is an energy penalty due to the spin-orbit coupling of the electrons in the ferromagnetic material. It favors the alignment of the magnetization vectors to a preferred direction given by the easy axis $\vec{a}$. The anisotropy constant k is the strength of this penalty. The external energy $\vec{E}_{\text{ext}}$ is the interaction energy of an external field $\vec{H}_{\text{ext}}$ with the magnetization.

From these energy terms the corresponding effective fields in the Landau-Lifshitz-Gilbert equation (Eq. 1.2) can be derived by the variational derivative

$$\vec{H}_{\text{tot}} = -\frac{1}{\mu_0}\frac{\delta E}{\delta \vec{M}}. \tag{1.4}$$

Several terms contribute to the total field to account for the different origins of total energy:

$$\vec{H}_{\text{tot}} = \vec{H}_{\text{exch}} + \vec{H}_{\text{demag}} + \vec{H}_{\text{aniso}} + \vec{H}_{\text{ext}} \tag{1.5}$$

In the following, the four fields in Eq. 1.5 are described.

1.2.1 Exchange field

The exchange energy is the result of the quantum-mechanical interaction between neighboring spins. It causes adjacent magnetization vectors to align parallel. Applying the variational derivative in Eq. 1.4 to the exchange energy yields the exchange field

$$\vec{H}_{\text{exch}} = \frac{2A}{\mu_0 M_s^2}\Delta\vec{M} \tag{1.6}$$

with the exchange stiffness constant A and the saturation magnetization M_s.

1.2.2 Demagnetization field

In the absence of electrical currents the Maxwell's equations reduce to

$$\vec{\nabla} \times \vec{H} = 0 \text{ (Ampere's law)} \tag{1.7}$$

and

$$\vec{\nabla} \cdot \vec{B} = 0 \text{ (Gauss' law)}. \tag{1.8}$$

The magnetic field $\vec{H}$ and the magnetic flux density $\vec{B}$ are related by

$$\vec{B} = \mu_0(\vec{M} + \vec{H}). \tag{1.9}$$

The solution of Eq. 1.7 can be expressed as the gradient of a scalar potential field ϕ:

$$\vec{H} = -\vec{\nabla}\phi. \tag{1.10}$$

Substituting Eq. 1.9 and Eq. 1.10 into Eq. 1.8 yields

$$\mu_0\vec{\nabla} \cdot (\vec{M} + \vec{H}) = \mu_0\vec{\nabla} \cdot (\vec{M} - \vec{\nabla}\phi) = 0 \tag{1.11}$$

and thus

$$\vec{\nabla} \cdot \vec{M} = \Delta\phi \tag{1.12}$$

inside the magnetized volume. Outside the magnetized volume $\Delta\phi = 0$. Given a magnetization $\vec{M}$, the solution for the scalar potential ϕ is [46]

$$\phi_M(\vec{r}) = -\frac{1}{4\pi}\int_V \frac{\nabla' \cdot \vec{M}(r')}{|\vec{r} - \vec{r}'|}\,\mathrm{d}V' + \frac{1}{4\pi}\int_{\partial V} \frac{\vec{n} \cdot \vec{M}(r')}{|\vec{r} - \vec{r}'|}\,\mathrm{d}S'. \tag{1.13}$$

resulting in the demagnetization field

$$\vec{H}_{\mathrm{demag}} = -\vec{\nabla}\phi_M. \tag{1.14}$$

In this work the terms demagnetization field and stray field are used interchangeably.

1.2.3 Anisotropy field

Magnetically anisotropic materials possess a direction dependence where the magnetic moments tend to align parallely to one or more preferred directions, the so-called easy axes[6]. One source of this effect is the atomic crystal structure of the material. Here the cases of one easy axis and three orthonormal axes are considered. In the case of one easy axis the variational derivative of the anisotropy energy yields the uniaxial anisotropy field

$$\vec{H}_{\mathrm{uni}} = \frac{2k}{\mu_0 M_s^2}(\vec{M} \cdot \vec{a})\vec{a}, \quad |\vec{a}| = 1, \tag{1.15}$$

with the unit vector $\vec{a}$ pointing into the easy axis and the anisotropy constant k, which in case of the crystalline anisotropy is a material constant. In the case of three easy axes the variational derivative of the cubic anisotropy energy yields the cubic anisotropy field

$$\vec{H}_{\mathrm{cub}} = \frac{2k}{\mu_0 M_s}\left(\alpha(\beta^2 + \gamma^2)\vec{a_1} + \beta(\alpha^2 + \gamma^2)\vec{a_2} + \gamma(\alpha^2 + \beta^2)\vec{a_3}\right). \tag{1.16}$$

It includes the three orthonormal easy axes $\vec{a_1}$, $\vec{a_2}$, and $\vec{a_3}$. The parameters α, β, and γ are the direction cosines of the magnetization to these axes (such that $\alpha\vec{a_1} + \beta\vec{a_2} + \gamma\vec{a_3} = \vec{m}(\vec{r})$).

1.2.4 External field

In the micromagnetic model the external field

$$\vec{H}_{\text{ext}} \tag{1.17}$$

is a term of the effective field.

1.3 Extensions for spin currents and electrical currents

The magnetization vectors can be excited by a magnetic field as well as by an electrical current. The spins of the itinerant electrons of the current exert a torque on the magnetization vectors. This spin torque is proportional to the electrical current density. For a spatially varying current density the strength of the spin torque varies with the current density. Additionally, the magnetization is influenced by the Oersted field that is generated by a spatially inhomogeneous current.

1.3.1 Spin-torque

If a current passes through a ferromagnetic monolayer a majority of the spins of itinerant electrons points up (or down), which is called spin polarization. The itinerant electron spins of a spin-polarized current exert a torque on the gradient of the localized magnetization of the ferromagnet. The spin torque was first introduced by Berger[47] and extended by Zhang and Li[48, 49]. This torque is modeled by the addition of a spin-torque term

$$\begin{aligned} ST(\vec{M}, \vec{j}) = &-(1 + \xi\alpha)\frac{b_j}{M_s(1+\alpha^2)}\vec{M} \times (\vec{M} \times (\vec{j} \cdot \vec{\nabla})\vec{M}) \\ &-(\xi - \alpha)\frac{b_j}{M_s^2(1+\alpha^2)}\vec{M} \times (\vec{j} \cdot \vec{\nabla})\vec{M} \end{aligned} \tag{1.18}$$

to the right hand side of the Landau-Lifshitz-Gilbert equation with the nonadiabaticity constant ξ, and the coupling constant b_j.

1.3.2 Macrolayer spin-torque

In the macro spin model only one local magnetization vector is assumed to be located in a ferromagnetic layer[50]. If a current is applied in the out-of-plane direction of a stack of ferromagnetic and nonmagnetic layers like in an MRAM cell, the itinerant electron

spins of the current exert a torque

$$ST(\vec{M},\vec{j}) = -\frac{\gamma a_j j}{M_s(1+\alpha^2)}\vec{M}\times(\vec{M}\times\vec{P}) + \frac{\gamma\alpha a_j}{1+\alpha^2}\vec{M}\times\vec{P}) \tag{1.19}$$

on the magnetization vector in the so-called free layer. In the free layer the magnetization can rotate in all directions while in the remaining layers of an MRAM-cell the magnetization is fixed by, e.g., antiferromagnets to polarize the spin current. This torque term is added to the right hand side of the Landau-Lifshitz Gilbert equation and includes the coupling constant a_j, and the spin polarization P.

1.3.3 Current paths

To investigate current-induced magnetization dynamics experimentally, electrical contacts are positioned at the surface of the ferromagnetic microstructure and a voltage is applied to allow an electron flow from one contact to the other. The resulting current paths show spatial variations, e.g. due to the sample's shape, electrical resistance, and magnetization. The difference of the electrostatic potential Φ at both contacts results in an electrical field $\vec{E} = -\vec{\nabla}\Phi$ which generates a current density $\vec{j}$ in the sample due to Ohm's law

$$\vec{j} = \sigma\vec{E} = -\sigma\vec{\nabla}\Phi. \tag{1.20}$$

The current density j is proportional to the conductance tensor σ which incorporates the conductance in all three spatial directions. The conductance tensor is defined as the inverse of the resistance tensor $\sigma = \rho^{-1}$. The resistance tensor includes the ohmic resistance ρ_Ω and the anisotropic magnetostatic resistance ρ_{AMR}. The local anisotropic magnetostatic resistance

$$\rho = \rho_\perp + (\rho_{||} - \rho_\perp)\cos^2(\theta(\vec{j},\vec{M})) = \rho_\perp + \Delta\rho\cos^2(\theta(\vec{j},\vec{M})) \tag{1.21}$$

depends on the angle $\theta(\vec{M},\vec{j})$ between the local magnetization $\vec{M}$ and the local current density $\vec{j}$ as given by the product of the difference between resistance $\rho_{||}$ with current parallel to the magnetization and the resistance with current perpendicular to the magnetization and the term $\cos^2(\theta(\vec{j},\vec{M}))$. In tensor notation the resistance tensor reads[51]

$$\rho = \begin{pmatrix} \rho_\perp(m_x^2+m_z^2)+\rho_\Omega & \Delta\rho m_x m_y & \Delta\rho m_x m_z \\ \Delta\rho m_x m_y & \rho_\perp(m_x^2+m_z^2)+\rho_{||}m_y^2+\rho_\Omega & \Delta\rho m_y m_z \\ \Delta\rho m_x m_z & \Delta\rho m_y m_z & \rho_\perp(m_x^2+m_y^2)+\rho_{||}m_z^2+\rho_\Omega \end{pmatrix}. \tag{1.22}$$

The current density inside the sample is computed using the continuity equation

$$\frac{\partial\rho}{\partial t} = \vec{\nabla}\cdot\vec{j} \tag{1.23}$$

with the electrical charge density ρ. Inserting Eq. 1.20 into Eq. 1.23, employing the divergence theorem $\int_V \vec{\nabla} \cdot \vec{j}\, \mathrm{d}V = \oint_S \vec{j} \cdot \vec{n}\, \mathrm{d}S$ with the surface element S and the normal $\vec{n}$ and assuming the quasi-static case $\frac{\partial \rho}{\partial t} = 0$ yields

$$\int \mathrm{d}S \vec{n} \sigma \vec{\nabla} \Phi = 0. \tag{1.24}$$

This equation uses the conductivity tensor σ. In case of more than one magnetization vector Eq. 1.24 can be brought in a form of a Linear Set of Equations. Solving this LSE gives the electrostatic potential inside the whole sample. Employing Eq. 1.20 one obtains the current density j.

1.3.4 Oersted field

Electrical currents flowing through the sample generate a magnetic field called the Oersted field. The field is a part of the effective field and thus influences the magnetization dynamics. Although the field is typically weak compared to the exchange and demagnetization fields, its influence can be significant. For example, the Oersted field contributes significantly to the current pulse induced switching behavior of magnetic vortices[52]. Given a current density field $\vec{j}(\vec{r})$ inside a volume V, the field is given by the Biot-Savart law

$$\vec{H}_{\mathrm{Oersted}}(\vec{r}) = \frac{1}{4\pi} \int_V \vec{j}(\vec{r}') \times \frac{\vec{r} - \vec{r}'}{|\vec{r} - \vec{r}'|^3}\, \mathrm{d}V'. \tag{1.25}$$

2 Discretization

Until now a continuous micromagnetic model has been introduced. In this chapter, the model is discretized in order to solve it by numerical methods. In section 2.1, the simulated volume is discretized spatially using a grid of rectangular cells. In section 2.2, the time is discretized into finite time steps by using Runge-Kutta solvers.

2.1 Finite-difference discretization

The simulation volume is spatially discretized by a rectangular grid with equally sized cuboid simulation cells. The simulation cells have the side lengths l_x, l_y and l_z along each direction. The mesh has a number of n_x, n_y and n_z cells in each direction, with a total of $n = n_x n_y n_z$ cells. The cells are enumerated using an index i counting from 0 to $(n-1)$. They have the volumes V_i and are located by their cell middle points $\vec{r}_i$.

The discretized magnetization field $\vec{M}_i$ at cell i is approximated by the average of the continuous magnetization field in that cell volume,

$$\vec{M}_i = \frac{1}{l_x l_y l_z} \int_{V_i} \vec{M}(\vec{r}) \, \mathrm{d}V. \tag{2.1}$$

Similarly, the terms of the total effective field are each discretized by

$$\vec{H}_{\mathrm{eff},i} = \frac{1}{l_x l_y l_z} \int_{V_i} \vec{H}_{\mathrm{x}}(\vec{r}) \, \mathrm{d}V. \tag{2.2}$$

All material parameters such as M_s, A, and α are specified per cell as well. Usually they are either specified by the user or drawn from a material database.

Some assumptions have to be made to discretized the continuous micromagnetic model. All fields of the continuous model are spatially averaged over each simulation cell. Thus inside each cell, the fields are assumed to be homogeneous. This approach is valid only for sufficiently small simulation cells. The cell size should not exceed the exchange length $l_\mathrm{ex} = \sqrt{\frac{A}{2M_s}}$ to resolve all magnetic phenomena which are described by the continuous model[6]. It is determined by the exchange stiffness constant A and saturation magnetization M_s.

In a computer implementation, the discretized fields are represented by three-dimensional arrays. They have the size (n_x, n_y, n_z) so that each array entry corresponds to one simulation cell. For example, the magnetization is stored in the magnetization array of vectors

$$\vec{M}[m,n,o] = \vec{M}_i \;\text{ where }\; \vec{r}_i = \left(l_x(m+\frac{1}{2}), l_y(n+\frac{1}{2}), l_z(o+\frac{1}{2}) \right), \tag{2.3}$$

where the indices of the array are enclosed in square brackets ([,]). Similarly, any other fields and material parameters that are given per cell are stored in arrays.

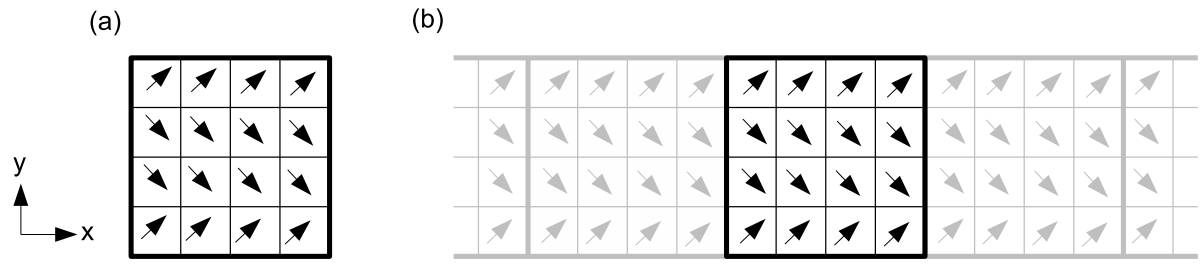


Figure 2.1: (a) Magnetized volume with open boundary conditions discretized by a rectangular mesh. (b) Magnetized volume with periodic boundary condition in x-direction. The thin lines denote the simulation cell sizes, the thick lines the periodic tiles. The magnetization dynamics are computed in the black tiles, and the grey tiles contain repetitions.

2.1.1 Boundary conditions

Numerical simulations are restricted to a finite number of cells. Thus finite-difference methods can only handle a finite region in which the magnetization dynamics are computed, and boundary conditions have to be defined. In micromagnetic simulations open and periodic boundary conditions can be applied.

With *open boundary conditions* (Fig. 2.1 (a)), the simulated magnetized volume is finite in space, i.e. outside this volume all magnetization vectors are zero. In this case only the magnetized volume has to be discretized by the mesh.

With *periodic boundary conditions* (Fig. 2.1 (b)), the assumption is that the magnetization pattern repeats itself infinitely along one or more principle directions. The magnetized volume is partitioned into tiles, where each tile contains one repetition of the magnetization. Non-repeating directions are handled as with open boundary conditions. It will be shown that with periodic boundary conditions, if suffices to compute the magnetization pattern for one tile only.

2.1.2 Exchange field

The Laplace operator in three dimensions in Cartesian coordinates is

$$\Delta = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right). \tag{2.4}$$

In the finite-difference scheme, the simplest numerical approximation of the Laplace operator is to calculate the second derivatives using central difference quotients. For example, along the x-axis with cell size l_x,

$$\frac{\partial^2}{\partial x^2} f(x,y,z) = \frac{f(x+l_x,y,z) - 2f(x,y,z) + f(x-l_x,y,z)}{l_x^2} + \mathcal{O}(l_x^2). \tag{2.5}$$

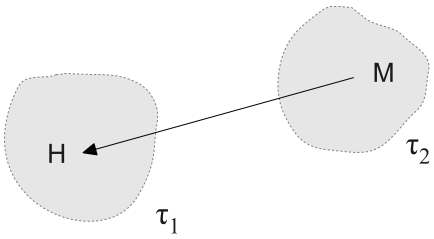


Figure 2.2: Two volumes that interact by the magnetostatic field. $\vec{M}$ is the average magnetization inside τ_2, and $\vec{H}$ is the average field seen in τ_1 that is generated by τ_2.

For small values of l_x the numerical error scales with l_x^2. Several exchange field approximations where investigated in Ref. [53]. It is shown that the simple nearest-neighbor scheme in Eq. 2.5 is an adequate approximation for micromagnetic simulations. Having cells with edge lengths (l_x, l_y, l_z), one can thus use

$$H_{\mathrm{exch},k}[i] = \frac{2A[i]}{\mu_0 M_\mathrm{s}[i] l_k^2} \sum_{\substack{j \in \mathrm{NN}(i), \\ M_\mathrm{s}[j]>0}} \left(m_k[i] - m_k[j]\right), \quad k \in \{\mathrm{x}, \mathrm{y}, \mathrm{z}\}, \tag{2.6}$$

where $\mathrm{NN}(i)$ returns the indices of the six nearest neighbors of cell i if i is inside the array (and less than six indices when i lies on the border), and $\vec{m}$ is the unit magnetization. Finite differences between cells where one M_s is zero are omitted from the sum.

Periodic boundary conditions With periodic boundary conditions, one has to make sure that the nearest-neighbor function NN in Eq. 2.6 returns the nearest neighbors by wrapping around the grid borders in all periodic directions.

2.1.3 Demagnetization field

Here the computation of the demagnetization field in a finite difference mesh is described. Each simulation cell holds a magnetization vector, which interacts with all magnetization vectors in the mesh, including itself. The average magnetic field seen in volume τ_1 due to a uniformly magnetized region τ_2 (see Fig. 2.2) can be generally expressed[54] as

$$\vec{H}_{\tau_1 \leftarrow \tau_2} = -\mathbf{N} \cdot \vec{M}_{\tau_2}, \tag{2.7}$$

Here $(\mathbf{N}_{ij})$ is the "generalized demagnetization tensor" and has 3×3 entries. It depends only on the geometry of the regions τ_1 and τ_2 and their distance. In the case of a finite-difference mesh, all regions τ_i that are considered are cuboid simulation cells of the same size. Thus, for any given cell pair (i, j) with midpoints $\vec{r}_i$ and $\vec{r}_j$ their generalized demagnetization tensor can be derived from the distance vector of the cells, and Eq. 2.7 can be written as

$$\vec{H}_{j \leftarrow i} = -\mathbf{N}(\vec{r}_i - \vec{r}_j) \cdot \vec{M}_i. \tag{2.8}$$

In order to get the total average demagnetization field reaching cell j, the generated fields from all cells, including cell j, are summed up,

$$\vec{H}_j = \sum_{i=0}^{n-1} \vec{H}_{j\leftarrow i} = -\sum_{i=0}^{n-1} \mathbf{N}(\vec{r}_i - \vec{r}_j) \cdot \vec{M}_i. \tag{2.9}$$

The vector field made up by all $\vec{H}_j$ is the discretized demagnetization field term of the total field. In order to calculate this field, the tensors $\mathbf{N}(\vec{r}_i - \vec{r}_j)$ have to be known for the midpoint-to-midpoint distances $\vec{r}_i - \vec{r}_j$ of all pairwise cells i and j. In the following it is shown how these tensors can be calculated.

Newell et al.[54] show how the demagnetization tensor in Eq. 2.9 can be derived. This is a generalization of the formulas derived in Ref. [55] where the cells are cubic. In a similar fashion, in Ref. [56] demagnetization tensors for prism-shaped cells and in Ref. [55] cubic cells were calculated. The scalar potential given in Eq. 1.13 includes volume and surface charges. By integration by parts the scalar potential can be brought to the form

$$\phi(\vec{r}) = \frac{1}{4\pi}\vec{M}_j \cdot \int_{V_j'} \vec{\nabla}'(\frac{1}{|\vec{r} - \vec{r}'|})\, \mathrm{d}V_j' \tag{2.10}$$

where the assumption that the magnetization inside each cell is uniform is used. Here V' is the volume of the cell, and $\vec{\nabla}'$ is the gradient with respect to $\vec{r}'$. The average field reaching cell i is then

$$\begin{aligned}
\vec{H}_{i\leftarrow j} &= \frac{1}{V_i}\int_{V_i} \left[-\vec{\nabla}\phi\right]\, \mathrm{d}V_i \\
&= \frac{1}{V_i}\int_{V_i} \left[-\vec{\nabla}\frac{1}{4\pi}\vec{M}_j \cdot \int_{V_j'} \vec{\nabla}'(\frac{1}{|\vec{r} - \vec{r}'|})\, \mathrm{d}V_j'\right]\, \mathrm{d}V_i \\
&= -\mathbf{N} \cdot \vec{M}_j,
\end{aligned} \tag{2.11}$$

where the components of the demagnetization tensor $\mathbf{N}$ are given by

$$N_{\alpha\beta} = -\frac{1}{4\pi V_i}\int_{V_i} \mathrm{d}V \int_{V_j'} \vec{\nabla}'_\alpha \vec{\nabla}'_\beta(\frac{1}{|\vec{r} - \vec{r}'|})\, \mathrm{d}V_j' \tag{2.12}$$

using $\vec{\nabla}(1/|\vec{r} - \vec{r}'|) = -\vec{\nabla}'(1/|\vec{r} - \vec{r}'|)$. The volume integral is converted to surface integrals using the divergence theorem,

$$\mathbf{N}_{\alpha\beta} = \frac{1}{4\pi V}\int_S \mathrm{d}S\vec{n}_\alpha \int_{S'} \frac{\mathrm{d}S'\vec{n}'_\beta}{|\vec{r} - \vec{r}'|}. \tag{2.13}$$

where S and S' are the surfaces of the source and target volumes. For pairs of cuboids of size $(\Delta X, \Delta Y, \Delta Z)$ the integral can be solved analytically. This equation is a sum of

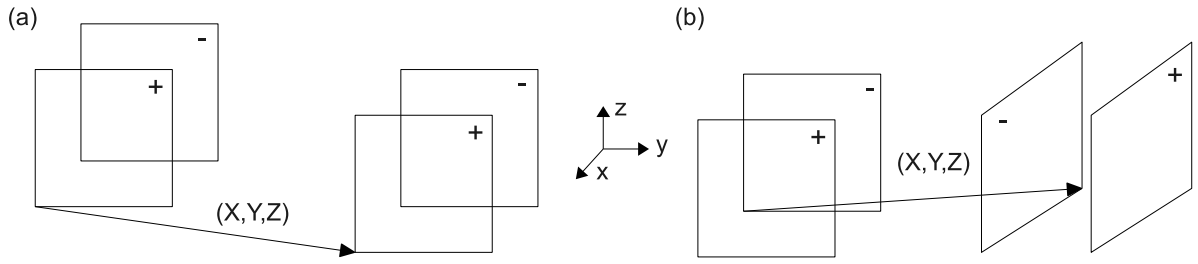


Figure 2.3: Surfaces of two interacting cells with a distance of (X, Y, Z) that contribute with a total of four surface pairs to (a) $\mathbf{N}_{\mathrm{xx}}$, see Eq. 2.14, and (b) $\mathbf{N}_{\mathrm{xy}}$, see Eq. 2.17. Surface pairs with different (same) signs contribute a negative (positive) term. (Figure adapted from Ref. [54].)

integrals for each pair of faces between the two cuboids. In total this sum has $6 \times 6 = 36$ integrals. For $\mathbf{N}_{xx}$, the integrals for face pairs with normals parallel to the x-direction are non-zero. Thus only four integrals remain, and Eq. 2.13 becomes

$$\mathbf{N}_{xx} = \frac{1}{4\pi V}(2F(X,Y,Z) - F(X+\Delta X,Y,Z) - F(X-\Delta X,Y,Z)), \tag{2.14}$$

see Fig. 2.3 (a). The integrals are of the form[54]

$$F(X,Y,Z) = \int_0^{\Delta z}\int_0^{\Delta y}\int_0^{\Delta z}\int_0^{\Delta y} \frac{\mathrm{d}z\,\mathrm{d}y\,\mathrm{d}z'\,\mathrm{d}y'}{\sqrt{X^2+(y+Y-y')^2+(z+Z-z')^2}}. \tag{2.15}$$

Rotation of the coordinate system yields the remaining diagonal components

$$\begin{aligned}\mathbf{N}_{yy}(X,Y,Z,\Delta X,\Delta Y,\Delta Z) &= \mathbf{N}_{xx}(Y,X,Z,\Delta Y,\Delta X,\Delta Z)\\ \mathbf{N}_{zz}(X,Y,Z,\Delta X,\Delta Y,\Delta Z) &= \mathbf{N}_{xx}(Z,Y,X,\Delta Z,\Delta Y,\Delta X).\end{aligned} \tag{2.16}$$

Similarly, for $\mathbf{N}_{xy}$ any integrals for a face pair where the first face has a normal along the x-direction and the second has a normal along the y-direction are non-zero. Again four integrals remain and Eq. 2.13 becomes

$$\mathbf{N}_{xy} = \frac{1}{4\pi V}(G(X,Y,Z) - G(X-\Delta X,Y,Z) - G(X,Y+\Delta Y,Z) + G(X-\Delta X,Y+\Delta Y,Z)), \tag{2.17}$$

see Fig. 2.3 (b). Here, the integrals are of the form

$$G(X,Y,Z) = \int_0^{\Delta z}\int_0^{\Delta y}\int_0^{\Delta z}\int_0^{\Delta x} \frac{\mathrm{d}z\,\mathrm{d}y\,\mathrm{d}z'\,\mathrm{d}x'}{\sqrt{(X-x')^2+(Y+y)^2+(z+Z-z')^2}}. \tag{2.18}$$

Rotation of the coordinate system yields the off-diagonal components

$$\begin{aligned}\mathbf{N}_{xz}(X,Y,Z,\Delta X,\Delta Y,\Delta Z) &= \mathbf{N}_{xy}(X,Z,Y,\Delta X,\Delta Z,\Delta Y)\\ \mathbf{N}_{yz}(X,Y,Z,\Delta X,\Delta Y,\Delta Z) &= \mathbf{N}_{xy}(Y,Z,X,\Delta Y,\Delta Z,\Delta X).\end{aligned} \tag{2.19}$$

Since the demagnetization tensor is symmetric, the remaining off-diagonal components can be obtained by the relation

$$\mathbf{N}_{\alpha\beta} = \mathbf{N}_{\beta\alpha}. \tag{2.20}$$

The integrals in Eq. 2.15 and Eq.2.18 and thus the whole demagnetization tensor can be calculated analytically for arbitrary cell distances[54]. After the demagnetization tensors for each cell midpoint-to-midpoint distance have been precalculated, Eq. 2.9 for the x-component (and similarly for the y- and z-component) of the demagnetization field reads

$$\begin{aligned} -H_{\text{demag,x}}(\vec{r}_i) = &\sum_j \mathbf{N}_{xx}(\vec{r}_i - \vec{r}_j) \cdot M_x(\vec{r}_j) \\ &+ \sum_j \mathbf{N}_{xy}(\vec{r}_i - \vec{r}_j) \cdot M_y(\vec{r}_j) \\ &+ \sum_j \mathbf{N}_{xz}(\vec{r}_i - \vec{r}_j) \cdot M_z(\vec{r}_j). \end{aligned} \tag{2.21}$$

This equation contains three discrete convolutions. In a computer implementation, the fields in Eq. 2.21 are stored in arrays. The linear convolution of two three-dimensional arrays A and B is defined as

$$(A * B)[l, m, n] = \sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} \sum_{k=0}^{n_z-1} A[i, j, k] \cdot B[l - i, m - j, n - k], \tag{2.22}$$

where $0 \leq l < n_x$, $0 \leq m < n_y$ and $0 \leq n < n_z$. Each component of the demagnetization tensor $N_{\alpha\beta}(\vec{r})$ is stored in a number array of size $(2n_x - 1, 2n_y - 1, 2n_z - 1)$, where each distance vector $\vec{r}$ is mapped to the array indices (i, j, k) using

$$\begin{aligned} N_{\alpha\beta}[i, j, k] \leftrightarrow N_{\alpha\beta}(&l_x((n_x + i) \bmod (2n_x - 1) - n_x), \\ &l_y((n_y + j) \bmod (2n_y - 1) - n_y), \\ &l_z((n_z + k) \bmod (2n_z - 1) - n_z)), \end{aligned} \tag{2.23}$$

see Fig. 2.4. Here the self-demagnetization tensor is mapped to the indices $(0, 0, 0)$. Eq. 2.21 can now be expressed using arrays as

$$\begin{aligned} -H_{\text{demag,x}} &= \mathbf{N}_{xx} * M_x + \mathbf{N}_{xy} * M_y + \mathbf{N}_{xz} * M_z \\ -H_{\text{demag,y}} &= \mathbf{N}_{yx} * M_x + \mathbf{N}_{yy} * M_y + \mathbf{N}_{yz} * M_z \\ -H_{\text{demag,z}} &= \mathbf{N}_{zx} * M_x + \mathbf{N}_{zy} * M_y + \mathbf{N}_{zz} * M_z \end{aligned} \tag{2.24}$$

where $(*)$ is the array convolution operator defined in Eq. 2.22.

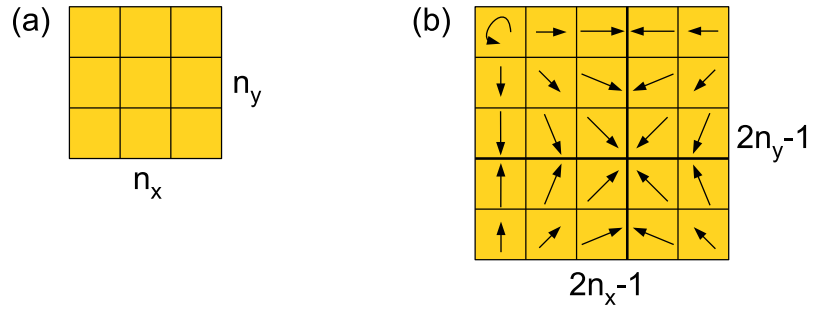


Figure 2.4: (a) Demagnetization array. (b) Mapping of distance vectors (black arrows) to indices (small rectangles) for the arrays of the demagnetization tensor components.

Periodic boundary conditions With periodic boundary conditions, the convolution sum has to be extended to take into account the magnetization outside the simulation volume. Because the magnetization repeats itself indefinitely, only one period is stored in the magnetization array. The extended sum is

$$-\vec{H}_{\mathrm{demag,p}}[j] = \sum_{k,l,m} \left(\sum_{i} \mathbf{N}(\vec{r}_i - \vec{r}_j + \mathrm{offset}(k,l,m)) \cdot \vec{M}[i] \right) \tag{2.25}$$

where $k, l, m \in \mathbb{Z}$ iterate over an infinite number of repetitions in three dimensions and $\mathrm{offset}(k,l,m) = kL_x\vec{e_x} + lL_y\vec{e_y} + mL_y\vec{e_z}$ is the distance vector of the tile at k, l, m from the tile at the origin. Rearranging Eq. 2.25 yields

$$-\vec{H}_{\mathrm{demag,p}}[j] = \sum_{i} \mathbf{N}_p(\vec{r}_i - \vec{r}_j) \cdot \vec{M}[i] \tag{2.26}$$

with the generalized demagnetization tensor for a infinitely repeating volume

$$\mathbf{N}_{\mathrm{p}}(\vec{r}_i - \vec{r}_j) = \sum_{k,l,m} \mathbf{N}(\vec{r}_i - \vec{r}_j + \mathrm{offset}(k,l,m)). \tag{2.27}$$

Again the tensor field $\mathbf{N}_p$ is constant as it only depends on the geometry of the mesh. In numeric simulations, the infinite sum in Eq. 2.27 needs to be approximated at the start of the simulation. As the magnetic field that is generated by one tile diminishes with the squared distance, it suffices to include only near tiles up to a limit, see Ref. [57].

2.1.4 Demagnetization field (scalar potential method)

An alternate way to calculate the demagnetization field is to calculate its scalar potential field Φ first. The magnetic field can then be derived by calculating its gradient using the relation $\vec{H} = -\vec{\nabla}\Phi$. In this section a fast algorithm for calculating the potential field is presented. Because it is similar to the fast convolution method described in the previous

section, all mentioned optimizations apply here too. However, the calculation of the scalar potential requires a smaller number of fast Fourier transforms, making it potentially faster. In addition, it requires less memory. The rest of this section follows closely the publication *A Fast Finite-Difference Method for Micromagnetics Using the Magnetic Scalar Potential* by C. Abert, G. Selke, B. Krüger, and A. Drews[58], Copyright 2012 IEEE. Similar techniques using fast Fourier transforms to calculate the scalar potential have been described in Ref. [59]. Equations 1.13 and 1.14 can be written as

$$\vec{H}_{\mathrm{demag}}(\vec{r}) = -\vec{\nabla}\phi = -\vec{\nabla}\int \mathbf{S}(\vec{r}-\vec{r}')\vec{M}(\vec{r}')\,\mathrm{d}\vec{r}'. \tag{2.28}$$

Discretization of the vector field $\mathbf{S}$ assuming cuboid cells yields

$$\mathbf{S}(\vec{r}_i - \vec{r}_j) = \frac{1}{4\pi}\int_{V_j} \nabla' \frac{1}{|\vec{r}-\vec{r}'|}\mathrm{d}^3\vec{r}'\Big|_{\vec{r}=\vec{r}_i} \tag{2.29}$$

where V_j is the cuboid homogeneously magnetized cell volume that generates the demagnetization field, and $\vec{r}_i - \vec{r}_j$ is the distance vector between target cell j and source cell i, measured from the center of the cells. The scalar potential can be expressed by the following discrete convolution

$$\phi(\vec{r}_i) = \sum_j \mathbf{S}(\vec{r}_i - \vec{r}_j)\vec{M}(\vec{r}_j) \tag{2.30}$$

Now the integral for $\mathbf{S}$ in Eq. 2.29 is solved for a cuboid volume V_j with side lengths l_x, l_y, and l_z. Applying the divergence theorem yields

$$\mathbf{S}_z = \frac{1}{4\pi}\int_{\partial V} \frac{\vec{n}'\cdot\vec{e}_z}{|\vec{r}-\vec{r}'|}\,\mathrm{d}A' \tag{2.31}$$

for the z-component of the vector field $\mathbf{S}$. The surface normal is denoted by $\vec{n}'$, and $\vec{e}_z$ is the unit vector in z-direction. The integral has only contributions on the two faces in the xy-plane:

$$\mathbf{S}_z = \sum_{\pm} \pm\frac{1}{4\pi}\int_{-l_x/2}^{l_x/2}\int_{-l_y/2}^{l_y/2} \frac{\mathrm{d}x'\mathrm{d}y'}{\sqrt{(x-x')^2+(y-y')^2+(z\mp l_z/2)^2}} \tag{2.32}$$

with $\vec{r} = (x, y, z)$. Solving the indefinite integral yields

$$F(x,y,z) = \frac{1}{4\pi}\int_0^x \int_0^y \frac{\mathrm{d}x'\mathrm{d}y'}{\sqrt{x'^2+y'^2+z'^2}} \tag{2.33}$$

$$= \frac{1}{4\pi}\Bigg\{ -x + z\arctan\left(\frac{x}{z}\right) - z\arctan\left(\frac{xy}{z\sqrt{x^2+y^2+z^2}}\right) + y\ln\left[2\left(x+\sqrt{x^2+y^2+z^2}\right)\right] + x\ln\left[2\left(y+\sqrt{x^2+y^2+z^2}\right)\right]\Bigg\}. \tag{2.34}$$

Now the definite integrals can be assembled:

$$\begin{aligned}
\mathbf{S}_z(\vec{r}) &= -\sum_{i,j,k} ijkF(x+i\tfrac{l_x}{2}, y+j\tfrac{l_y}{2}, z+k\tfrac{l_z}{2}),\\
\mathbf{S}_x(\vec{r}) &= -\sum_{i,j,k} ijkF(y+i\tfrac{l_y}{2}, z+j\tfrac{l_z}{2}, x+k\tfrac{l_x}{2}),\\
\mathbf{S}_y(\vec{r}) &= -\sum_{i,j,k} ijkF(z+i\tfrac{l_z}{2}, x+j\tfrac{l_x}{2}, y+k\tfrac{l_y}{2}).
\end{aligned} \tag{2.35}$$

with $i, j, k \in \{-1, +1\}$. With the formulation in Eq. 2.35 for $\mathbf{S}(\vec{r})$ the scalar potential at any point r of the demagnetization field that is generated by a set of homogeneously magnetized cuboid cells with center points r_k can now be determined exactly using the scalar product

$$\Phi(\vec{r}) = \sum_{\vec{r}_k} \mathbf{S}(\vec{r} - \vec{r}') \cdot \vec{M}(\vec{r}_k) \tag{2.36}$$

or

$$\Phi = \mathbf{S}_x * M_x + \mathbf{S}_y * M_y + \mathbf{S}_z * M_z. \tag{2.37}$$

where $*$ is the discrete convolution. The gradient of Φ yields the demagnetization field.

2.1.5 Anisotropy field

The continuous equations for the uniaxial and cubic anisotropy field directly translate into their spatially discrete counterparts. The discrete uniaxial anisotropy field approximation at cell i is

$$\vec{H}_{\mathrm{uni}}[i] = \frac{2k[i]}{\mu_0 M_{\mathrm{s}}[i]}(\vec{m}[i] \cdot \vec{a})\vec{a}, \tag{2.38}$$

where $\vec{a}$ is the unit vector pointing in the direction of the easy axis. The cubic anisotropy field approximation becomes

$$\vec{H}_{\mathrm{cub}}[i] = \frac{2k[i]}{\mu_0 M_{\mathrm{s}}[i]}\left(\alpha(\beta^2+\gamma^2)\vec{a_1} + \beta(\alpha^2+\gamma^2)\vec{a_2} + \gamma(\alpha^2+\beta^2)\vec{a_3}\right). \tag{2.39}$$

with $\alpha = \vec{m}[i] \cdot \vec{a_1}$, $\beta = \vec{m}[i] \cdot \vec{a_1}$, and $\gamma = \vec{m}[i] \cdot \vec{a_1}$.

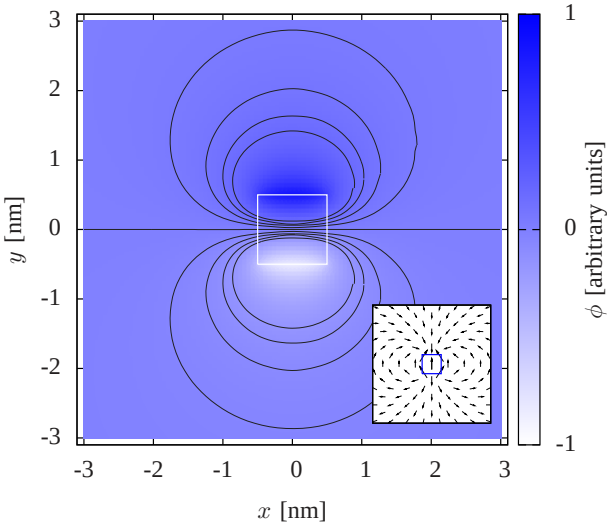


Figure 2.5: Scalar potential Φ of a cube, which is homogeneously magnetized in the z-direction, of size 1 nm $\times$ 1 nm $\times$ 1 nm. The inlay shows the gradient field of the potential. (Figure from Ref. [58], Copyright 2012 IEEE.)

Periodic boundary conditions As the anisotropy field is computed locally for each simulation cell, no special treatment regarding different boundary conditions is needed. Thus Eq. 2.38) and Eq. 2.39 can be used without modification.

2.1.6 External field

The external field is simply averaged over each simulation cell,

$$H_{\text{ext}}[i] = H_{\text{ext,i}}. \tag{2.40}$$

Periodic boundary conditions No special treatment is required.

2.1.7 Oersted field

For the finite-difference approximation of the Oersted field the Biot-Savart law has to be discretized. As the Oersted field is treated as homogeneous in each cell, its average value is used.

$$\vec{H}_{\text{Oer}}[i] = \frac{1}{V_i} \int_{V_i} \vec{H}_{\text{Oersted}}(\vec{r})\, \mathrm{d}V_i, \tag{2.41}$$

where V_i is the volume of cell i. Similar to the demagnetization field, the Oersted field can be expressed by a convolution

$$\vec{H}_{\text{Oer}}, j = \sum_i \mathbf{K}(\vec{r}_j - \vec{r}_i) \cdot \vec{j}(\vec{r}_i) \tag{2.42}$$

with the antisymmetric tensor $\mathbf{K}$ and the current density j. A detailed calculation of the Oersted field is presented in Ref. [60].

Periodic boundary conditions The Oersted field under periodic boundary conditions can be calculated analogously to the periodic demagnetization field, see also Ref. [60].

2.1.8 Current paths

The continuity equation in Eq. 1.23 is solved numerically using the method described in Ref. [61]. The continuity equation must hold for every simulation cell, resulting in $i = 0, 1, \ldots (n-1)$ equations

$$\int_{S_i} \sigma_i \vec{\nabla} \Phi_i \vec{n} \, \mathrm{d}S = 0. \tag{2.43}$$

The conductivity tensor σ_i and the electrostatic potential Φ_i are now assumed non-changing in each cell. The surface integral incorporates six cell surfaces, yielding

$$\begin{aligned} & l_y l_z (\sigma_{xx} \, \mathrm{D}_x \, \Phi_i + \sigma_{xy} \, \mathrm{D}_y \, \Phi_i + \sigma_{xz} \, \mathrm{D}_z \, \Phi_i) \\ + & l_x l_z (\sigma_{yx} \, \mathrm{D}_x \, \Phi_i + \sigma_{yy} \, \mathrm{D}_y \, \Phi_i + \sigma_{yz} \, \mathrm{D}_z \, \Phi_i) \\ + & l_x l_y (\sigma_{zx} \, \mathrm{D}_x \, \Phi_i + \sigma_{zy} \, \mathrm{D}_y \, \Phi_i + \sigma_{zz} \, \mathrm{D}_z \, \Phi_i) = 0 \end{aligned} \tag{2.44}$$

as the finite-difference approximation. Here D_d is the finite difference approximation of a partial first derivative in dimension d. Now the discretized equations form a system of n linear equations, which is solved numerically[61] for Φ_i, $i = 0..(n-1)$.

$$\mathbf{A}\vec{\Phi} = (0, 0, \ldots, 0)^T \tag{2.45}$$

where $\mathbf{A}$ is a $n \times n$ matrix and $\vec{\Phi}$ is a vector with n components. Applying Eq. 1.20 to the computed $\vec{\Phi}$ yields the current density.

Periodic boundary conditions Periodic boundary conditions for the computation of current paths are not covered in this thesis.

2.2 Time discretization

Together with an initial magnetization $\vec{M}_0$, the LLG equation forms an initial value problem[62] with one magnetization vector for each simulation cell $i = 0 \ldots (n-1)$,

$$\begin{aligned} \vec{M}(\vec{r}_i, t = 0) &= \vec{M}_0(\vec{r}_i) \\ \frac{\mathrm{d}\vec{M}(\vec{r}_i, t)}{\mathrm{d}t} &= f(t, \vec{M}(\vec{r}_i, t)) \end{aligned} \tag{2.46}$$

where f is the right hand side of Eq. 1.2. The magnetization at some time t is given by the integral

$$\vec{M}(\vec{r}_i, t) = \vec{M}(\vec{r}_i, t_0) + \int_{t_0}^{t} f_i(\tau, \vec{M}(\vec{r}_i, \tau))\, \mathrm{d}\tau. \tag{2.47}$$

A numerical solver for an initial value problem calculates an approximation of the above integral using discretized time steps.

Explicit Euler method One of the simplest explicit Runge-Kutta methods for the numerical approximation of Eq. 2.47 is the explicit Euler integration method. Starting from the Taylor series up to the second order

$$\vec{M}(\mathbf{r}, t) = \vec{M}(\mathbf{r}, t_0) + \frac{\mathrm{d}\vec{M}(\mathbf{r}, t_0)}{\mathrm{d}t} \cdot (t - t_0) + \mathcal{O}((t - t_0)^2) \tag{2.48}$$

the substitutions $h := t - t_0$ and $t := t_0$ yield

$$\vec{M}(\mathbf{r}_i, t + h) = \vec{M}(\mathbf{r}_i, t) + h \cdot f(t, \vec{M}(\mathbf{r}_i, t)) + \mathcal{O}(h^2) \tag{2.49}$$

for every cell i where h is the time step size. Numerical integration of Eq. 2.49 with discrete time steps t_κ yields

$$\vec{M}(\mathbf{r}_i, t_{\kappa+1}) \approx \vec{M}_0(\mathbf{r}) + \sum_{k \in \{1,2,\ldots\kappa\}} h \cdot f(t_k, \vec{M}(\mathbf{r_i}, t_k)), \tag{2.50}$$

which is an approximation of Eq. 2.47. In order to compute the approximation of $\mathbf{M}(\mathbf{r}, t_{\kappa+1})$ only the intermediate result of the previous stage is needed, which makes this method a one-stage method.

2.2.1 Explicit Runge-Kutta schemes

Equation 2.47 can be numerically integrated using Runge-Kutta schemes. Explicit Runge-Kutta schemes with s stages are defined by the equation

$$\vec{y}_{n+1} = \vec{y}_n + h \sum_{i=1}^{s} b_i \vec{k}_i \tag{2.51}$$

with the stage vectors

$$\vec{k}_1 = f(t_n, \vec{y}_n), \tag{2.52}$$

$$\vec{k}_i = f(t_n + c_i, \vec{y}_n + \sum_{j=1}^{i-1} a_{i,j} \vec{k}_j),\ 2 \leq i \leq s \tag{2.53}$$

A concrete Runge-Kutta method is specified with the coefficients $a_{i,j}$, b_i $(1 \leq i \leq s)$, and c_i $(2 \leq i \leq s)$. The coefficients can be written down in a so-called Butcher tableau,

$$\begin{array}{c|ccc} 0 & & & \\ c_2 & a_{2,1} & & \\ \vdots & \vdots & \ddots & \\ c_s & a_{s,1} & \cdots & a_{s,s-1} \\ \hline & b_1 & \cdots & b_s \end{array} \tag{2.54}$$

where the coefficients c_i in Eq. 2.52 are written vertically in the first column, the coefficients b_i in Eq. 2.51 horizontally at the bottom row, and the coefficients a_{ij} in Eq. 2.52 in the middle. For explicit Runge-Kutta schemes with more than one stage, the matrix $(a_{i,j})$ is lower-triangular. Here the stage vectors $\vec{k}$ can be calculated in succession, as $\vec{k}_i$ depends only on $\vec{k}_0, \vec{k}_1, \ldots, \vec{k}_{i-1}$. After the stage vectors are computed, they are combined in Eq. 2.51 to yield the next integration result.

Classical Runge-Kutta method The classical Runge-Kutta method is an explicit four-stage method which has a lower step error than the explicit Euler method. A weighted average of four stage vectors $\vec{k}_n$ is used to advance the solution from a previous value $\vec{M}(t_i)$ to the next value $\vec{M}(t_{i+1})$,

$$\vec{M}(t_{i+1}) = \vec{M}(t_i) + \frac{1}{6}\left(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4\right) + \mathcal{O}(h^5)$$

with $t_{i+1} = t_i + h$ where h is the time step. The second term on the right hand side gives the weighted average of the four stage vectors. The stage vectors are calculated as follows:

$$\begin{aligned} k_1 &= h \cdot f(t_n, y_n) & k_2 &= h \cdot f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\ k_3 &= h \cdot f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) & k_4 &= h \cdot f(t_n + h, y_n + k_3). \end{aligned} \tag{2.55}$$

Thus, the butcher tableau is

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} . \tag{2.56}$$

The step error is $\mathcal{O}(h^5)$.

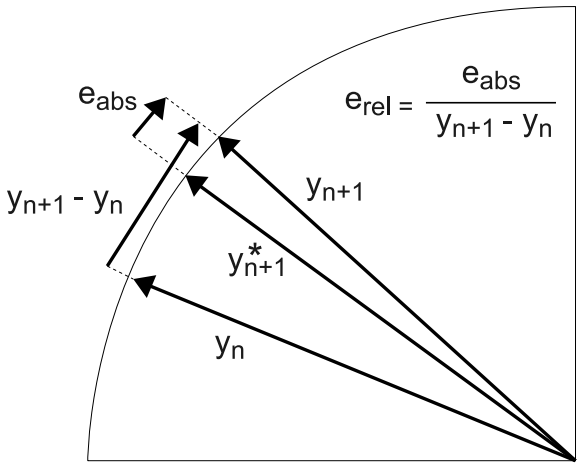


Figure 2.6: Relationship between the absolute error vector $\vec{e}_{\text{abs}}$, the relative error vector $\vec{e}_{\text{rel}}$, and the vectors $\vec{y}_n$, $\vec{y}_{n+1}$, and $\vec{y}^*_{n+1}$ in embedded Runge-Kutta methods with step size control.

Embedded Runge-Kutta methods with adaptive step size control Embedded Runge-Kutta methods try to estimate the error produced by one Runge-Kutta step. This error can then be used to control the step size h. The error estimation is done by having two methods of different order in the Butcher tableau, one with error order p and one with error order $p-1$. The embedded $(p-1)$-order step is given by

$$\vec{y}^*_{n+1} = \vec{y}_n + h \sum_{i=1}^{s} b^*_i \vec{k}_i. \tag{2.57}$$

It uses the same stage vectors $\vec{k}_i$ that are used to calculate the p-order step, but uses a different weighted average given by a new different set of coefficients b^*_i. The estimated absolute error is the difference between the embedded step and the output step,

$$\vec{e}_{\text{abs}} = \vec{y}_{n+1} - \vec{y}^*_{n+1}, \tag{2.58}$$

see Fig. 2.6. Given the estimated error in Eq. 2.58 that would be produced at the next step, the step size h may be adjusted for the following step. On the one hand, the step size is desired to be as large as possible so that the numerical integration proceeds fast. On the other hand, increasing h also increases the step error. Thus, an adaptive step size controller adjusts the step size to be as large as possible to that the error is within a tolerable error margin. The error vector given in Eq. 2.58 is the estimated *absolute error* per Runge-Kutta step. In practice, for the step size control a scalar quantity that represents the error is needed. An obvious choice is the maximum norm of the error vector, which selects the "worst offending" variable with the highest error, e.g.

$$e_{\text{abs}} = ||\vec{e}_{\text{abs}}||_{\text{max}}. \tag{2.59}$$

This norm is chosen in MicroMagnum. Another frequently used norm is the euclidean norm, which can be optionally selected. It is useful to also estimate the *relative error*. The relative error is the element-wise ratio of the absolute error $\vec{e}_{\text{abs}}$ to the integration step vector, see Fig. 2.6. Again the worst offending variable is chosen using

$$\vec{e}_{\text{rel}} = e_{\text{abs}} / ||\vec{y}_{n+1} - \vec{y}_n||_{\max}. \tag{2.60}$$

The step size to the $(s+1)$-th power is proportional to the absolute error, $h^{s+1} \sim e_{\text{abs}}$. The new step size h_{new} is now chosen such that the next step is expected to produce the tolerated error $e_{\text{abs,tol}}$. As $h_{\text{new}}^{s+1} \sim e_{\text{abs,tol}}$, the new step size is chosen as

$$h_{\text{abs,new}}^{s+1} = S h_{\text{old}}^{s+1} \frac{e_{\text{abs,tol}}}{e_{\text{abs}}} \Leftrightarrow h_{\text{abs,new}} = h_{\text{old}} \left(\frac{e_{\text{abs,tol}}}{e_{\text{abs}}} \right)^{1/(s+1)} \tag{2.61}$$

Here $S \leq 1$ is called the step headroom. It is chosen as $S = 0.85$. The relative error is of order s. The new step size according to the relative order thus becomes

$$h_{\text{rel,new}} = h_{\text{old}} \left(\frac{e_{\text{rel,tol}}}{e_{\text{rel}}} \right)^{1/s}. \tag{2.62}$$

For the step size control, the smaller, more "pessimistic" step size is chosen from the two new step sizes that were suggested from the absolute and relative error:

$$h_{\text{new}} = \min(h_{\text{abs,new}}, h_{\text{rel,new}}) \tag{2.63}$$

If at the next step, for the new step size, a too high absolute or relative error is produced, the step has to be retried with a smaller h. For the calculation of the smaller h Eqs. 2.61 and 2.62 are used.

A commonly used method of orders 5 and 4 is the Runge-Kutta-Fehlberg method[63], which has the following tableau:

0						
1/4	1/4					
3/8	3/32	9/32				
12/13	1932/2197	−7200/2197	7296/2197			
1	439/216	−8	3680/513	−845/4104		
1/2	−8/27	2	−3544/2565	1859/4104	−11/40	
	16/135	0	6656/12825	28561/56430	−9/50	2/55
	25/216	0	1408/2565	2197/4104	−1/5	0

(2.64)

The first row at the bottom gives the fourth-order solution (b_i) and the second gives the fifth-order solution (b_i^*), both by applying Eq. 2.51 to the same stage vectors. Other

explicit Runge-Kutta methods are the Dormand-Prince[64] method that uses a fifth-order step to advance the solution with an embedded fourth-order step, and the fourth-order Cash-Karp[65] method with an embedded fourth-order step.

2.2.2 Time integration in micromagnetic simulation

Renormalization After each Runge-Kutta step, the vectors of the magnetization array have to be re-normalized so that each vector retains a length of M_s at any time as defined by the micromagnetic model.

Stop criterion In practical simulations, a stop criterion has to be defined that determines when the simulation or a part thereof has completed. Which criterion is used depends on the particular simulation problem. For example, a stop criterion could be met when

- a certain simulation time is reached
- a certain number of time steps have completed
- a relaxed magnetization state is reached.

The maximum simulation time or number of time steps is specified by the user of the simulation software. A relaxed magnetization state is a configuration that is located at a local energy minimum. Without external excitation, the configuration remains static. Thus in order to detect such a energy minimum, the stop criterion can be defined in terms of the rotational speed of the magnetization vectors. The rotational speed, measured in rad/s, approaches zero when the magnetization relaxes. Given to consecutive steps i and $i+1$ separated by a time step h, it can be approximated by

$$\theta(\vec{r}_n) \approx \frac{1}{h} \arctan \frac{|\vec{m}_{i+1}(\vec{r}_n) - \vec{m}_i(\vec{r}_n)|}{|\vec{m}_i(\vec{r}_n)|} \approx \frac{|\vec{m}_{i+1}(\vec{r}_n) - \vec{m}_i(\vec{r}_n)|}{h}. \tag{2.65}$$

In practice the magnetization vector with the highest rotational speed is regarded in order to determine whether the whole magnetization state is relaxed, i.e.

$$\textit{magnetization is relaxed} \Leftrightarrow \max_{n=1..n} \theta(\vec{r}_n) < \theta_\mathrm{tolerance}. \tag{2.66}$$

A commonly selected value for $\theta_\mathrm{tolerance}$ is 1 degree per nanosecond.

3 Fast Numerical Methods

In order to compute the time derivative of the magnetization using the Landau-Lifshitz-Gilbert equation, the total field on the right hand side has to be computed from the current magnetization. Thus discretized versions of the demagnetization field (Eq. 2.9), the exchange field (Eq. 2.6), the anisotropy field (Eqs. 2.38, 2.39), and the external field (Eq. 2.40) have to be computed. These computations should be as fast as possible, as the Landau-Lifshitz-Gilbert equation is evaluated frequently during a simulation. The most time-consuming part is the computation of the convolution in the demagnetization and the Oersted field. Thus the main focus of this section lies on the fast computation of the convolutions. This is achieved by using fast Fourier transforms.

3.1 Demagnetization field

The demagnetization field in either tensor notation

$$\begin{aligned} -H_{\text{demag,x}} =& \mathbf{N}_{xx} * M_x + \mathbf{N}_{xy} * M_y + \mathbf{N}_{xz} * M_z \\ -H_{\text{demag,y}} =& \mathbf{N}_{xy} * M_x + \mathbf{N}_{yy} * M_y + \mathbf{N}_{yz} * M_z \\ -H_{\text{demag,z}} =& \mathbf{N}_{xz} * M_x + \mathbf{N}_{yz} * M_y + \mathbf{N}_{zz} * M_z \end{aligned} \tag{3.1}$$

(exploiting the symmetry $N_{\alpha\beta} = N_{\beta\alpha}$) or in vector notation

$$\vec{H}_{\text{demag}} = \vec{\nabla} \left(\mathbf{S}_x * M_x + \mathbf{S}_y * M_y + \mathbf{S}_z * M_z \right). \tag{3.2}$$

is a part of the LLGE. The aim is to compute Eq. 3.1 and Eq. 3.2 as fast as possible. From a computational perspective, both equations contain similar operations on multi-dimensional scalar arrays, e.g. the component wise summation ($+$) and discrete linear convolutions ($*$). Thus for the algorithmic optimization both computations can be treated very similarly. In particular, the convolution dominates all other operations in terms of computation time. In the following the fast computation of the convolution is achieved by using fast Fourier transforms while exploiting the properties of the demagnetization tensor field $\mathbf{N}$, the scalar potential vector field $\mathbf{S}$ and the magnetization $\vec{M}$.

3.1.1 Fast convolution

The straight-forward computation of the convolutions in Eqs. 3.1 and 2.37 involves the iteration over all pairs of simulation cells leading to a time complexity in $\mathcal{O}(n^2)$, where n is the number of cells. The computation can be sped up by using the *fast convolution*, which employs the discrete convolution theorem

$$A * B \leftrightarrow \hat{A} \cdot \hat{B}, \tag{3.3}$$

where the hat accent symbol denotes the transform of a sequence, with fast Fourier transforms[66] (FFTs) to replace the convolution with a product in the frequency domain. The convolution theorem is valid for any number of spatial dimensions. Using FFTs, the discrete Fourier transforms can be computed in $\mathcal{O}(n \log n)$ time. If n is large enough the fast convolution yields an extreme reduction of the computation time compared to the straight-forward $\mathcal{O}(n^2)$ algorithm. The fast convolution method has been used successfully in finite-difference micromagnetic simulators[19, 67, 25, 26, 17] to compute the demagnetization field. In the following the fast convolution and the fast Fourier transform in multiple dimensions are described in detail.

Fast Fourier transforms There are many fast Fourier transform algorithms that execute in $\mathcal{O}(n \log n)$ time. Here the so-called radix-2 algorithm for the computation of the discrete Fourier transform (DFT) is shortly introduced. The DFT of an complex input sequence x_k of length N, $0 \leq k < N$, results in a transformed complex output $\hat{x}_k$ of the same length. It is defined as

$$\hat{x}_i = \sum_{j=0}^{N-1} x_j \cdot \omega_N^{-ij}, \quad \text{where} \quad \omega_N^k = e^{2\pi\sqrt{-1}k/N}. \tag{3.4}$$

Here ω_N^k is the Nth root of unity raised to the k-th power. The radix-2 algorithm requires that N is a power of two, i.e. there exists an integer p so that $2^p = N$. The input sequence x is split into the two subsequences $x'_k = x_{(2k)}$ and $x''_k = x_{(2k+1)}$ of length $n = N/2$, $0 < k \leq n$. The sum is now converted to two sums,

$$\begin{aligned} \hat{x}_i &= \sum_{k=0}^{n-1} x'_k \cdot \omega_N^{-i(2k)} + \sum_{k=0}^{n-1} x''_k \cdot \omega_N^{-i(2k+1)} \\ &= \sum_{k=0}^{n-1} x'_k \cdot \omega_n^{-ik} + \omega_N^{-i} \cdot \sum_{k=0}^{n-1} x''_k \cdot \omega_n^{-ik}. \end{aligned} \tag{3.5}$$

The sums form the discrete Fourier transform of the subsequences x' and x'' of length $N/2$, respectively. It follows

$$\hat{x}_i = \begin{cases} \hat{x}'_i + \omega_N^{-i} \cdot \hat{x}''_i, & i < n \\ \hat{x}'_{i-n} - \omega_N^{-(i-n)} \cdot \hat{x}''_i, & i \geq n. \end{cases} \tag{3.6}$$

For the $i \geq n$ case the relations $\omega_N^k = \omega_N^{k+N}$ and $\omega_N^k = -\omega_N^{k+(N/2)}$ were used.

Computing the transforms of the subsequences recursively using the same equation yields the radix-2 fast Fourier transform algorithm. The input data is recursively split until the trivial case with input size $N = 1$ is reached, where $\hat{x}_0 = x_0$. The requirement that the total transform size $N = 2^p$ is a power of two ensures that the trivial case

is always reached after p recursions. The listing 1 shows the recursive radix-2 FFT algorithm.

```
1  def fft_rec(x):
2    """
3    Returns the fast Fourier transform of the complex input sequence 'x'.
4    The input length must be a power of two.
5    """
6    def root(n, k): return exp(2*pi*1j*k/n)
7    N = len(x)
8    if N == 1:
9      return x
10   else:
11     x_e = fft_rec([x[i*2  ] for i in range(N/2)]) # x_e = fft(x')
12     x_o = fft_rec([x[i*2+1] for i in range(N/2)]) # x_o = fft(x'')
13     for i in range(N/2):
14       x[i    ] = x_e[i] + root(N, -i) * x_o[i]
15       x[i+N/2] = x_e[i] - root(N, -i) * x_o[i]
16     return x
```

Listing 1: Recursive version of the radix-2 FFT algorithm.

In line 9–10 the trivial case $N = 1$, which stops the recursion, is handled. Otherwise, in lines 12 and 13 the even and odd subsequences of the input are created and their transforms computed recursively. The roots of unity, computed by the function `root`, are also referred to as twiddle factors. The loop at lines 16–18 computes all $\hat{x}_k$ from the previously transformed subsequences using Eq. 3.6. Because the total recursion depth is $p = \log_2 N$, and the function calls within each recursion level do work linear in N, the total computational complexity is $\mathcal{O}(N \log N)$.

For good performance, this recursive algorithm is not ideal. During the execution temporary storage has to be allocated in order to store the transforms of the even and odd input sequences. Also, the implemented function is not tail-recursive and thus can not be trivially optimized by the compiler. It is however possible to reformulate the algorithm in an iterative fashion which works in-place with only a constant amount of temporary storage. From the recursive algorithm one can see that before any output is calculated the data is recursively split into even and odd parts, see the so-called butterfly diagram in Fig. 3.1 (a). After the input is split up, the output sequence is assembled by applying the twiddle factors, see Fig. 3.1 (b). Due to the recursive splits, the original input is reordered from left to right in the so-called *bit-reversed order*. In bit-reversed order the input array X is permuted using the index mapping

$$f(i) = (b_{(n-1)} b_{(n-2)} \dots b_1 b_0)_2 \tag{3.7}$$

where i is the index ranging from 0 to $2^n - 1$ and $(b_0 b_1 \dots b(n-2) b(n-1))_2$ is the binary representation of i. The iterative radix-2 algorithm has two steps, both of which can be done in-place using only a small constant amount of temporary storage.

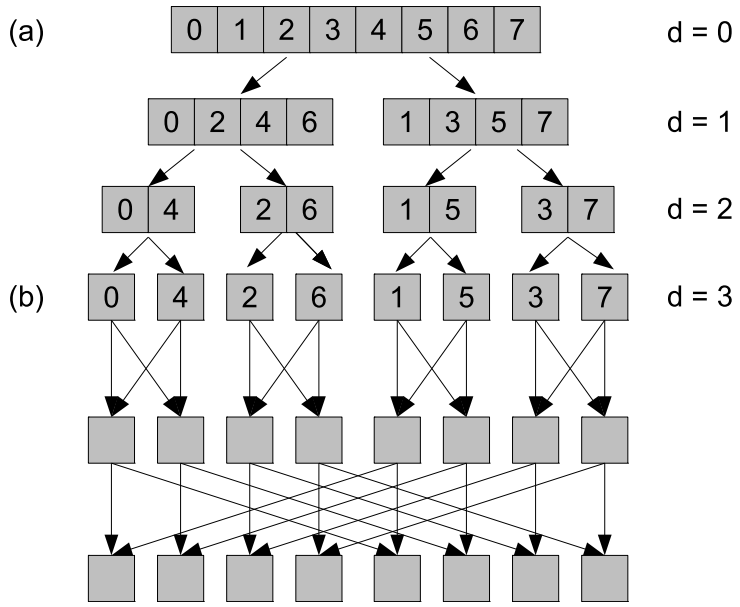


Figure 3.1: Data flow of a radix-2 FFT of length $N = 8$ using the algorithm in listing 1. The numbers denote the item indices of the input sequence. (a) Recursive splitting of the input into even and odd parts where d is recursion depth, resulting in bit-reversed order at $d = 3$. (b) Multiplication with the twiddle factors and assembly of the result. This diagram is called the butterfly diagram.

(a) reorder the input to bit-reversed order.

(b) traverse the butterfly diagram (see Fig. 3.1 (b)) from top to bottom.

The listing 2 shows the bit reversal algorithm in step (a).

```
1  def bit_reverse(x):
2    """
3    Converts the sequence 'x' to bit-reversed order.
4    """
5    n = len(x)
6    j = 0 # j is counted up in bit-reversed order.
7    for i in range(0, n-1): # i is counted up in normal order.
8      # Swap elements i and j (but swap each pair only once!)
9      if j > i:
10       x[i], x[j] = x[j], x[i]
11
12     # Increase j by one in bit-reversed order, e.g. 11001 -> 00101.
13     k = n >> 1
14     while k <= j:
15       j -= k
16       k >>= 1
17     j += k
```

Listing 2: Permute a sequence to bit-reversed order.

Here the input sequence is traversed by the indices i and j. i is counted up in normal order, and j is counted up in bit-reversed order. During the traversal in lines 7–18, the

sequence elements at i and j are swapped (line 10). The condition $j > i$ at line 9 ensures that each possible element pair is only swapped once. Lines 12–17 increases j by one in bit-reversed order. The listing 3 shows the iterative radix-2 FFT algorithm in step (b).

```
1  def fft(x):
2    """
3    Computes the fast Fourier transform of the complex input sequence 'x
       '.
4    The input length must be a power of two.
5    """
6
7    def root(N, k): return exp(2*pi*1j*k/N)
8
9    N = len(x)
10   bit_reverse(x)
11
12   fft_num = N/2
13   fft_len = 2
14   while fft_num > 0:
15     for m in range(fft_len/2):
16       W = root(fft_len, -m)
17       for i in range(m, N, fft_len):
18         j = i + fft_len/2;
19         x[i] = x[i] + W * x[j];
20         x[j] = x[i] - W * x[j];
21     fft_num >>= 1
22     fft_len <<= 1
```

Listing 3: Iterative version of the radix-2 algorithm.

In line 9, the input array of size N is permuted in bit-reversed order. The loop starting at line 13 iterates over the butterfly diagram in Fig. 3.1 (b) from top to bottom with a total of $\log_2(N)$ iterations. The loops at lines 14 and 16 iterates over the subarrays in the butterfly diagram, combining `fft_num` subarrays of length `fft_len` in lines 18–19. The indices `i` and `j` represent the connecting lines from one level to the next in the butterfly diagram.

The presented radix-2 algorithm is a special case of the Cooley-Tukey fast Fourier transform algorithm. The Cooley-Tukey algorithm is more general, as it splits a sequence of length $N = N_1 \cdot N_2$ into N_1 transforms of length N_2 followed by multiplications by the twiddle factors. Starting from the discrete Fourier transform $\hat{x}_k = \sum_{j=0}^{N-1} x_j \omega_N^{-jk}$, the substitutions $j = j_1 n_2 + j_2$ and $k = k_1 + k_2 n_1$ eventually lead to

$$\hat{x}_{k_1+k_2 n_1} = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} x_{j_1+n_2} \omega_{n_1}^{-j_1 k_1} \right) \omega_N^{-j_2 k_1} \right] \omega_{n_2}^{-j_2 k_2}. \tag{3.8}$$

Like the radix-2 algorithm (which is the special case $N_2 = 2$), this equation can be implemented using an iterative algorithm. Equation 3.8 can be applied directly when N is not a prime number. When a prime length is encountered during the recursion, one

has to resort to a different method to calculate that subtransform. One way is to simply calculate Eq. 3.4 directly, another way is to use a specialized prime-length transform algorithm like the Rader's algorithm[68] or the Bluestein's chirp-z algorithm[69].

The two-dimensional discrete Fourier transform $\hat{x}$ of an input x of size (n_x, n_y) is

$$\hat{x}_{i,j} = \sum_{k=0}^{n_x-1} \sum_{l=0}^{n_y-1} x_{k,l} \cdot \omega_{n_x}^{-ik} \omega_{n_y}^{-jl} \tag{3.9}$$

There are many multi-dimensional fast Fourier transform algorithms[70]. The most straight-forward algorithm is the *row-column* algorithm. Here the two-dimensional transforms are reduced to series of one-dimensional transforms. Rearranging Eq. 3.9 leads to

$$\hat{x}_{i,j} = \sum_{l=0}^{n_y-1} \left[\sum_{k=0}^{n_x-1} x_{k,l} \cdot \omega_{n_x}^{-ik} \right] \omega_{n_y}^{-jl} \tag{3.10}$$

The expression in square brackets is a one-dimensional DFT of length n_x in x-direction, and the outer expression is a one-dimensional DFT of length n_y in y-direction. In a computer implementation, the computation of the whole 2-D transform involves the computation of N_y 1-D transforms of length N_x iterated along the y-direction, and afterwards N_x 1-D transforms of length N_y iterated along the x-direction. For the 1-D transforms the previously introduced FFT algorithm can be used. In this case the total asymptotic run time is in

$$\mathcal{O}(n_y(n_x \log n_x) + n_x(n_y \log n_y)) = \mathcal{O}(n_x n_y \log n_x n_y) = \mathcal{O}(n \log n) \tag{3.11}$$

for $n = n_x n_y$ the number of data points. The three-dimensional discrete Fourier transforms is defined analogously, i.e.

$$\hat{x}_{i,j,k} = \sum_{l=0}^{n_x-1} \sum_{m=0}^{n_y-1} \sum_{n=0}^{n_z-1} x_{l,m,n} \cdot \omega_{n_x}^{-il} \omega_{n_y}^{-jm} \omega_{n_z}^{-kn} \tag{3.12}$$

Here the row-column algorithm involves three iterated 1-D transforms: First, $n_y n_z$ transforms in x-direction, second, $n_x n_y$ transforms in y-direction, and third, $n_y n_z$ transforms in z-direction, see Fig. 3.2. Again the run time is in $\mathcal{O}(n \log n)$ where n is the number of data points.

Until now only forward transforms have been considered. The *inverse transform* will be needed to implement fast convolutions. The discrete Fourier transform in one dimension of a sequence of length n is

$$x_i = \frac{1}{n} \sum_{j=0}^{N-1} \hat{x}_j \cdot \omega_N^{+ij}. \tag{3.13}$$

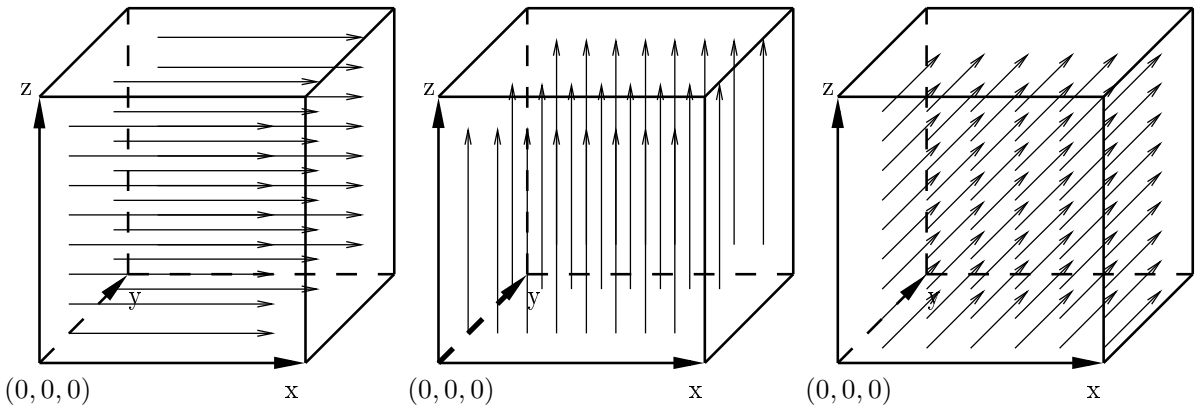


Figure 3.2: Computation of the three-dimensional fast Fourier transform using the row-column algorithm in three successive steps. The square denotes the working array. The arrows indicate one-dimensional subtransforms. (Figure adapted from Ref. [71].)

Applying the inverse transform to a transformed sequence will produce the original sequence. The two- and three-dimensional inverse transforms are

$$x_{i,j} = \frac{1}{n_x n_y} \sum_{k=0}^{n_x-1} \sum_{l=0}^{n_y-1} \hat{x}_{k,l} \cdot \omega_{n_x}^{+ik} \omega_{n_y}^{+jl} \tag{3.14}$$

and

$$x_{i,j,k} = \frac{1}{n_x n_y n_z} \sum_{l=0}^{n_x-1} \sum_{m=0}^{n_y-1} \sum_{n=0}^{n_z-1} \hat{x}_{l,m,n} \cdot \omega_{n_x}^{+il} \omega_{n_y}^{+jm} \omega_{n_z}^{+kn}. \tag{3.15}$$

As the only difference to the forward transforms is the prefactor and the plus sign, these equations can be incorporated into the previously discussed fast Fourier transforms algorithms (such as the Cooley-Tukey algorithm and the row-column algorithm) with only minor modifications, producing fast inverse Fourier transform algorithms.

Discrete convolution theorem The discrete *one-dimensional* cyclic convolution $Z = X * Y$ of two N-length arrays X and Y is defined as

$$Z_k = \sum_{j=0}^{N-1} X_j \cdot Y_{(N+j-k) \bmod N}, \; k = 0..(N-1). \tag{3.16}$$

This operation is commutative. According to the discrete convolution theorem, it can be expressed as an element-wise product $(\cdot)$ in the frequency domain,

$$Z = X * Y \Leftrightarrow \hat{Z} = \hat{X} \cdot \hat{Y}. \tag{3.17}$$

Starting with

$$\hat{X}_k \cdot \hat{Y}_k = \sum_{l=0}^{N-1} X_l \cdot \omega_N^{-kl} \cdot \sum_{m=0}^{N-1} Y_m \cdot \omega_N^{-km} = \sum_{l=0}^{N-1} \sum_{m=0}^{N-1} X_l \cdot Y_m \cdot \omega_N^{-k(l+m)}, \tag{3.18}$$

substituting $l + m := p$ (and thus $m = p - l$) and switching the sums gives

$$\hat{X}_k \cdot \hat{Y}_k = \sum_{p=0}^{N-1} \underbrace{\left(\sum_{l=0}^{N-1} X_l \cdot Y_{N+p-l) \bmod N} \right)}_{X * Y =: Z} \cdot \omega_N^{-kp} = \hat{Z}_k. \tag{3.19}$$

The computation of the convolution via fast Fourier transforms using the convolution theorem is called *fast convolution*. The fast convolution computes $Z = X * Y$ by the transformation of X and Y into the frequency domain and the computation of the product, followed by an inverse transformation that yields the result. The discrete *two-dimensional* cyclic convolution of two arrays X and Y of size (n_x, n_y) is

$$(X * Y)_{k_x,k_y} = \sum_{j_x=0}^{n_x-1} \sum_{j_y=0}^{n_y-1} X_{j_x,j_y} \cdot Y_{(n_x+j_x-k_y) \bmod n_x,(n_y+j_y-k_y) \bmod n_y}. \tag{3.20}$$

The discrete convolution theorem in Eq. 3.17 is valid in two dimensions as well. Here two-dimensional transforms are employed. The convolution in three dimensions and the fast three-dimensional convolution is defined in a similar fashion. In general, the fast convolution works for any number of dimensions.

Fast convolution of real inputs The discrete Fourier transform of a sequence of length N that contains only real values has conjugate symmetry around the element at $\lceil \frac{N}{2} \rceil$, i.e. $\hat{X}_k = \overline{\hat{X}_{N-k}},\ k \in \{1, 2, \ldots N\}$. Using $\exp(x) = \overline{\exp(-x)}$ yields

$$\hat{X}_{N-k} = \sum_{j=0}^{N-1} X_j \cdot \omega_N^{-j(N-k)} = \sum_{j=0}^{N-1} X_j \cdot \underbrace{\omega_N^{-Nj}}_{=1} \omega_N^{jk} = \sum_{j=0}^{N-1} \overline{X_j \cdot \omega_N^{-jk}} = \overline{\hat{X}(k)}. \tag{3.21}$$

Due to the symmetry it suffices to compute and store only half of the symmetric output. A discrete Fourier transform that does this is called a *real-to-complex* (R2C) transform. It takes a real input of length N and computes $\lceil \frac{N+1}{2} \rceil$ complex values. The corresponding inverse *complex-to-real* (C2R) transform takes $\lceil \frac{N+1}{2} \rceil$ complex inputs and computes N real outputs. Most fast Fourier transform algorithms, e.g. the Cooley-Tukey algorithm, can be modified by removing the redundant parts of the operations to produce a R2C and a C2R version. This saves roughly one half of the computation time. As the intermediary product in the frequency domain preserves the conjugate symmetry, i.e. $\overline{(z_1 z_2)} = \overline{z_1}\ \overline{z_2},\ z_1, z_2 \in \mathbb{C}$, the *fast convolution of two real inputs* can be implemented by

using R2C and C2R transforms. Only $\lceil \frac{N+1}{2} \rceil$ products have to be calculated. Altogether a speedup of about two is reached compared to the general fast convolution. The fast convolution of real inputs can be extended to two and more dimensions by applying the *row-column* algorithm. For example, the two-dimensional algorithm for each of the two inputs X, Y of size (N_x, N_y) is:

1. For each input X and Y, compute the 2-D R2C FFT,
   - Compute N_y R2C FFTs of length N_x along the x-direction, resulting in a complex array of size $(\lceil N_x/2+1 \rceil, N_y)$,
   - Compute $\lceil N_x/2+1 \rceil$ FFTs of length N_y along the y-direction,
2. Compute the $\lceil N_x/2+1 \rceil \times N_y$ products in the frequency domain,
3. Compute the inverse 2-D C2R FFT of the products:
   - Compute $\lceil N_x/2+1 \rceil$ inverse FFTs of length N_y along the y-direction,
   - Compute N_y inverse C2R FFTs of length N_x along the x-direction, producing the real-valued result array of size $(\lceil N_x/2+1 \rceil, N_y)$.

Fast convolutions of real inputs in higher dimensions can be implemented accordingly.

3.1.2 Fast field computation

The fast convolution computes a cyclic convolution where negative indices to the array Y are "wrapped around" using the modulo operator, see Eqs. 3.16 and 3.20. A direct use of the cyclic convolution in the demagnetization field computation would introduce non-physical cell interactions. For a correct result these terms have to be removed. This is achieved by padding the magnetization array with zeros in each spatial dimension,

$$\vec{M}^*[k_0,k_1,k_2] = \begin{cases} \vec{M}[k_0,k_1,k_2] & \text{if } k_0 < n_x \wedge k_1 < n_y \wedge k_2 < n_z \\ (0,0,0)^T, & \text{else} \end{cases}, \tag{3.22}$$

so that all unwanted convolution terms become zero, see Fig. 3.3. Furthermore the magnetization array and the demagnetization tensor array need to have the same dimensions in order to compute the element-wise products. Thus the magnetization array of size (n_x, n_y, n_z) is zero-padded to the dimensions $(2n_x-1, 2n_y-1, 2n_z-1)$ before Eqs. 3.23 and 3.24 can be used. If the magnetization array has singular dimensions, i.e. is an effectively one- or two-dimensional array, no zero-padding in these directions is needed as $2n_i - 1 = 1$.

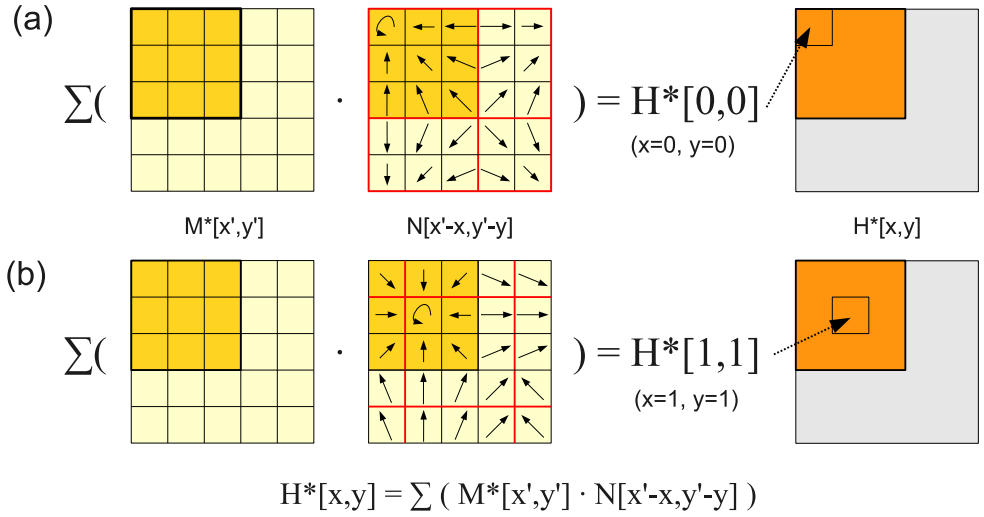


Figure 3.3: Computation of the demagnetization field array of size 3×3 using a cyclic convolution with a zero-padded magnetization array of size 5×5. (a) and (b) display the exemplary convolution sum for the demagnetization array at $x = 0, y = 0$ and $x = 1, y = 1$, respectively. The colors encode the inputs that contribute only zero-valued terms to the cyclic convolution (yellow), the inputs that contribute non-zero terms to the cyclic convolution (orange), the entries that are cut out from the resulting array (gray), leaving the calculated demagnetization field (red).

Following the convolution theorem and the fact that the convolution and the fast Fourier transform are linear operations, Eq. 3.1 takes the form

$$\begin{aligned} -\hat{H}^*_{\mathrm{x}} &= \hat{\mathbf{N}}_{\mathrm{xx}} \cdot \hat{M}^*_{\mathrm{x}} + \hat{\mathbf{N}}_{\mathrm{xy}} \cdot \hat{M}^*_{\mathrm{y}} + \hat{\mathbf{N}}_{\mathrm{xz}} \cdot \hat{M}^*_{\mathrm{z}} \\ -\hat{H}^*_{\mathrm{y}} &= \hat{\mathbf{N}}_{\mathrm{yx}} \cdot \hat{M}^*_{\mathrm{x}} + \hat{\mathbf{N}}_{\mathrm{yy}} \cdot \hat{M}^*_{\mathrm{y}} + \hat{\mathbf{N}}_{\mathrm{yz}} \cdot \hat{M}^*_{\mathrm{z}} \\ -\hat{H}^*_{\mathrm{z}} &= \hat{\mathbf{N}}_{\mathrm{zx}} \cdot \hat{M}^*_{\mathrm{x}} + \hat{\mathbf{N}}_{\mathrm{zy}} \cdot \hat{M}^*_{\mathrm{y}} + \hat{\mathbf{N}}_{\mathrm{zz}} \cdot \hat{M}^*_{\mathrm{z}} \end{aligned} \tag{3.23}$$

and Eq. 3.2 takes the form

$$\begin{aligned} \vec{H} &= \vec{\nabla}\Phi \\ \hat{\Phi}^* &= \hat{\mathbf{S}}_{\mathrm{x}} \cdot \hat{M}^*_{\mathrm{x}} + \hat{\mathbf{S}}_{\mathrm{y}} \cdot \hat{M}^*_{\mathrm{y}} + \hat{\mathbf{S}}_{\mathrm{z}} \cdot \hat{M}^*_{\mathrm{z}} \end{aligned} \tag{3.24}$$

in the frequency domain. Here the magnetization is zero-padded. The transforms of the tensor components $\hat{N}_{\alpha\beta}$ and the vector components $\hat{S}_\alpha$ can be precomputed once and then used repeatedly to compute the demagnetization field for different magnetization configurations. Thus, one demagnetization field computation involves six transforms in Eq. 3.23 and four transforms in Eq. 3.24, the multiplication in the frequency domain, and several additions. After the inverse transform, the demagnetization field $\vec{H}$ has to be cut out from the transform output $\vec{H}^*$ in Eq. 3.23, see Fig. 3.3. In Eq. 3.24, Φ has to be cut out from the expanded Φ^*.

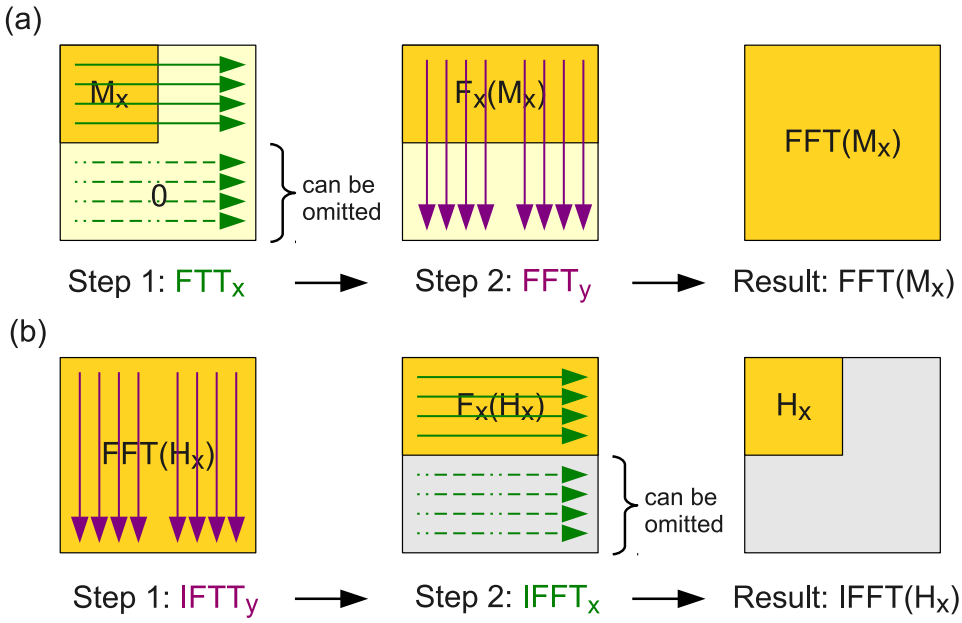


Figure 3.4: Computation steps of (a) the forward and (b) the inverse sparse fast Fourier transform in two dimensions, implemented with the row-column algorithm. The green (magenta) arrows denote iterated 1-D transforms in the x-direction (y-direction). The light yellow area denotes zero-valued input, input, the grey areas denote values that do not contribute to the final result.

Fast convolution with sparse input and output For the demagnetization field computation using the fast convolution the zero-padded magnetization array is Fourier transformed. After the product in the frequency domain and the inverse transform, the demagnetization field is cut out from the transform output array. Here the forward transform and the inverse transform operate on *sparse data*:

- The magnetization array is zero-padded, and so is the input to the forward FFTs.
- Only the unpadded parts of the inverse FFT output represent the actual demagnetization field. The padded parts contain cyclic convolution sums without physical meaning. Thus the output of the inverse FFT can be considered sparse.

In short, the forward transform has an input that is partially zero, and the backward transform has an output that is partially unneeded. Implementations of such sparse transforms that exploit the sparsity of their input and output can be made more efficient than general FFTs by leaving out redundant calculations. They are called *sparse* FFTs.

In the case of N-D sparse forward transforms, the row-column algorithm can be modified to leave out the 1-D subtransforms that have an input of zero, see Fig. 3.4 (a). The omission of these subtransforms is possible because the Fourier transform of a zero-valued sequence is zero. Similar considerations apply to the inverse transform, see Fig. 3.4 (b). Here those subtransforms are left out that do not contribute to the final, cut out result.

In case of 2-D inputs, approximately one half of the forward and inverse transforms in x-direction can be omitted from the computation, see Fig. 3.4, reducing the computation time by 25%. Similarly, for 3-D inputs, about 3/4 of the x-transforms and 1/2 of the y-transforms are omitted, reducing the computation time by about 40%.

Exploiting the even/odd properties of $\mathbf{N}_{ij}$ The demagnetization tensor field $\mathbf{N}(\vec{r})$ with $\vec{r} = (x, y, z)$ is real-valued and has the following properties

- $N_{xx}(x, y, z)$, $N_{yy}(x, y, z)$ and $N_{zz}(x, y, z)$ are even in x, y and z,
- $N_{xy}(x, y, z)$ is odd in x and y, and even in z,
- $N_{xz}(x, y, z)$ is odd in x and z, and even in y,
- $N_{yz}(x, y, z)$ is odd in y and z, and even in x.

This is a consequence of the even and oddness of the analytical functions f and g in Ref. [54] that are used to construct the demagnetization tensor. Due to the symmetries, each tensor component in the frequency domain is also real-valued. The even/odd properties carry over into the frequency domain,

- $\hat{N}_{xx}(x, y, z)$, $\hat{N}_{yy}(x, y, z)$ and $\hat{N}_{zz}(x, y, z)$ are even in x, y and z,
- $\hat{N}_{xy}(x, y, z)$ is odd in x and y, and even in z,
- $\hat{N}_{xz}(x, y, z)$ is odd in x and z, and even in y,
- $\hat{N}_{yz}(x, y, z)$ is odd in y and z, and even in x.

This means that for the storage of the demagnetization tensor field in the frequency domain only one octant needs to be stored, as symmetric entries can be generated on the fly by reflection for the multiplication in the frequency domain. Also, no imaginary parts need to be stored. This results in a memory requirement of about $6N$ floating point numbers where N is the number of cells. The product is simplified and thus sped up a little as one factor is now real-valued. Similar considerations apply to the scalar potential method. Here the field $\mathbf{S}(\vec{r})$ is real-valued and has the properties

- $S_x(x, y, z)$ is odd in x and even in y and z,
- $S_y(x, y, z)$ is odd in y and even in x and z, and
- $S_z(x, y, z)$ is odd in z and even in x and y.

Because of these properties, all components are real-valued, and in the frequency domain any imaginary parts vanish. The even/odd properties again translate into the frequency domain:

- $\hat{S}_x(x, y, z)$ is odd in x and even in y and z,
- $\hat{S}_y(x, y, z)$ is odd in y and even in x and z, and
- $\hat{S}_z(x, y, z)$ is odd in z and even in x and y.

Thus, the storage requirement for the field array in the frequency domain amounts to $2N$ where N is the number of cells.

Additional zero-padding Because the demagnetization field is determined through a linear convolution, additional zero-padding may be applied to the operands **M** and **N**, thus increasing their dimensions. This can be beneficial because the performance of the mixed-radix algorithm for calculating fast Fourier transforms typically depends on the transform size. For example, the FFTW library [72, 73, 74, 75] for calculating FFTs and inverse FFTs performs best when the transform length for one dimension is a product of primes smaller than thirteen[70]. This is because in the implemented algorithm of Cooley and Tukey, the n-length transform is recursively divided into n_0 transforms of length n_1 where $n = n_0 \cdot n_1$. The recursion stops either when n is a prime or when a hard coded constant-length transform routine becomes available. A prime-length transform is, for example, calculated with the naive $O(n^2)$ DFT algorithm, with the Rader's algorithm[68], or with Bluestein's chirp z-algorithm[69]. FFTW contains many hard coded, highly optimized routines for constant-length transforms of sizes smaller than about 30. Thus, when the total transform length is a product of small primes, the efficient Cooley-Tukey algorithm together with the hard coded routines is exclusively used.

A detailed empirical analysis for fast Fourier transform lengths in a micromagnetic simulator is presented in Ref. [25]. As additional zero-padding also increases the amount of data that needs to be processed, a strategy that selects the padding size for a given transform size is needed. Too much padding will increase the memory requirements and again decrease the overall performance. With no extra padding, $(\tilde{l}_x, \tilde{l}_y, \tilde{l}_z)$ has a size of $(2l_x - 1, 2l_y - 1, 2l_z - 1)$, which is uneven and thus clearly a bad choice. An obvious strategy is therefore to round each transform length up to the next even integer. In general, the transform lengths could be rounded up to the next integer that is a multiple of 2, 4, 8, an so on. An extreme strategy is to round each length up to the next power of two. This is done in the OOMMF[19] micromagnetic simulator, where only radix-2 transforms are implemented. Another strategy is to increase the transform length iteratively by one until an number with small prime factors, preferably factors of 2, is reached. Table 3.1 summarizes the strategies available in MicroMagnum. The default strategy is set to *round_4*, which provides good performance for most grid configurations. When the magnetization array is resized by the additional zero-padding, the size of the demagnetization tensor arrays has to be adjusted as well by inserting additional entries. These values are in principle arbitrary because they do not contribute to the computed

Strategy	Round mode ($i \in x, y, z$)
none	No zero-padding: $\tilde{l}_i = 2l_i - 1$
round_2	Round to even size: $\tilde{l}_i = 2l_i$
round_4	Round $\tilde{l}_i$ to next multiple of 4
round_8	Round $\tilde{l}_i$ to next multiple of 8
round_POT	Round to next power-of-two: $\tilde{l}_i = 2^m > 2 \cdot l_i - 1 > 2^{m-1}$
round_primes(n)	Round up until integer with prime factors $\leq n$ is reached

Table 3.1: Zero-padding strategies that are available in MicroMagnum.

demagnetization field result. However, in order to preserve the even/oddness properties which are needed for the previous optimization, these entries should be set to zero. If (e_x, e_y, e_z) are the new dimensions of the magnetization array according to a zero-padding strategy in Tab. 3.1, the tensor index mapping Eq. 2.23 becomes

$$\begin{aligned} N_{\alpha\beta}[i,j,k] \leftrightarrow N_{\alpha\beta}(& l_x((n_x + i) \bmod (e_x) - n_x), \\ & l_y((n_y + j) \bmod (e_y) - n_y), \\ & l_z((n_z + k) \bmod (e_z) - n_z)). \end{aligned} \tag{3.25}$$

Entries which correspond to cell-to-cell distances that do not exist in the discretizing mesh are set to zero to preserve the even/oddness properties presented in section 3.1.2.

3.1.3 Full algorithm

In the following all of the above optimizations are integrated into a fast convolution algorithm specifically designed for the demagnetization field computation. For the fast Fourier transforms it uses a modified sparse row-column method. The row-column method is used to split the multidimensional FFT into one-dimensional FFTs. Empirically, one-dimensional fast Fourier transforms can be executed fastest when its input and/or output is stored in memory contiguously, i.e. the spatial data locality is increased. This improves the chance that during the FFT computation the working memory fits into the cache memory. This improves the speed of the algorithm. For the row-column method, the data locality for the transforms in y- and z-direction can be improved by rotating the working data array in memory, so that the transform of the desired direction works on continuously stored data. Here it is assumed that the arrays are stored in column-major order, i.e. the first dimension x is contiguous. This is optimal for the application of the 1D transforms in x-direction. On the other hand, before the transforms in y-direction are applied, the input array is rotated in memory so that the y-dimension becomes contiguous. The same applies for FFTs in z-direction. For the sparse forward transform, the array rotation substeps include simultaneous zero-padding of the output

along the next contiguous dimension. Conversely, for the sparse backward transform, the array rotation substeps include simultaneous cutting of the input along the original contiguous dimension (see Fig. 3.5). This way all unnecessary 1D-FFTs are omitted from the computation. For meshes with a two-dimensional grid of simulation cells, the algorithm to compute the demagnetization field is as follows:

0. Initialization

    1. Choose a zero-padding strategy according to Tab. 3.1. Denote the magnetization arrays dimensions as (n_x, n_y) and its zero-padded dimensions as (e_x, e_y).
    2. Precompute, negate, zero-pad from the size of $(2n_x - 1, 2n_y - 1)$ to size of (e_x, e_y), transpose, and finally fast Fourier transform the demagnetization tensor arrays $N_{\alpha\beta}$.

I. Forward transform each of M_x, M_y, M_z:

    (a) zero-pad M_i along the x-direction, resulting in an array of size (e_x, n_y).

    (b) by looping along the y-direction, perform n_y 1d real-to-complex FFTs along the x-direction, resulting in a complex array of size $(\lceil e_x/2 + 1 \rceil, n_y)$.

    (c) zero-pad M_i along the y-direction and transpose it, resulting in a complex array of size $(e_y, \lceil e_x/2 + 1 \rceil)$.

    (d) by looping along the original x-direction, perform $\lceil e_x/2 + 1 \rceil$ 1d FFTs along the original y-direction.

II. (e) compute the products in the frequency domain according to Eq. 3.23.

III. Inverse transform each of H_x, H_y, H_z:

    (f) by looping along the original x-direction, perform $\lceil e_x/2 + 1 \rceil$ 1d IFFTs along the original y-direction.

    (g) cut M_i along the y-direction and transpose it, resulting in a complex array of size $(\lceil e_x/2 + 1 \rceil, n_y)$.

    (h) by looping along the y-direction, perform n_y 1d complex-to-real FFTs along the x-direction, resulting in a complex array of size (e_x, n_y).

    (i) cut M_i along the x-direction, resulting in an array of size (n_x, n_y).

The demagnetization field computation on three-dimensional grids of simulation cells require 3-D transforms. These include an additional rotation and transform in z direction substep. In the following the three-dimensional version of the fast sparse convolution algorithm is given.

0. Initialization

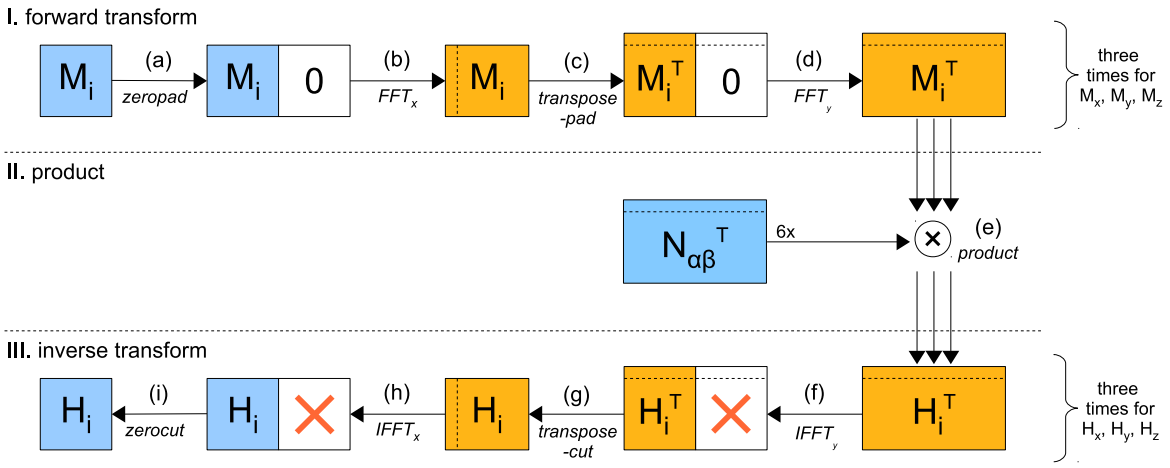


Figure 3.5: (a) – (i) Computational steps to compute the demagnetization field $\vec{H}$ from the magnetization $\vec{M}$ on a two-dimensional array of simulation cells. The filled rectangles denote two-dimensional number arrays that serve as the input and output of the steps. Blue arrays contain real-valued numbers, and orange arrays contain complex-valued numbers.

1. Choose a zero-padding strategy according to Tab. 3.1. Denote the magnetization arrays dimensions as (n_x, n_y, n_z) and its zero-padded dimensions as (e_x, e_y, e_z).
2. Precompute, negate, zero-pad from $(2n_x - 1, 2n_y - 1, 2n_z - 1)$ to (e_x, e_y, e_z), rotate, and finally fast Fourier transform the demagnetization tensor arrays $N_{\alpha\beta}$.

I. Forward transform each of M_x, M_y, M_z:

(a) zero-pad M_i along the x-direction, resulting in an array of size (e_x, n_y, n_z).

(b) by looping along the y- and z-directions, perform $n_y n_z$ 1d real-to-complex FFTs along the x-direction, resulting in a complex array of size $(\lceil e_x/2 + 1 \rceil, n_y, n_z)$.

(c) zero-pad M_i along the y-direction and rotate it, resulting in a complex array of size $(e_y, \lceil e_x/2 + 1 \rceil, n_z)$.

(d) by looping along the original x- and z-directions, perform $\lceil e_x/2 + 1 \rceil n_z$ 1d FFTs along the original y-direction.

(e) zero-pad M_i along the z-direction and rotate it, resulting in a complex array of size $(e_z, \lceil e_x/2 + 1 \rceil, \lceil e_y/2 + 1 \rceil)$.

(f) by looping along the original x- and y-directions, perform $\lceil e_x/2+1 \rceil \lceil e_y/2+1 \rceil$ 1d FFTs along the original z-direction.

II. (g) compute the products in the frequency domain according to Eq. 3.23.

Subroutine	Listing in appendix A.2
zeropad	(20)
unpad	(21)
rotate-zeropad	(22)
rotate-cut	(23)
$\mathrm{FFT_i}$	(24)
$\mathrm{FFT_i}$ (real-to-complex)	(25)
$\mathrm{FFT_i}$ (complex-to-real)	(26)
product	(27)

Table 3.2: Definition of the subroutines used in steps (a)–(i) in Fig. 3.5 and (a)–(m) in Fig. 3.6 .

III. Inverse transform each of H_x, H_y, H_z:

(h) by looping along the original x- and y-directions, perform $\lceil e_x/2+1 \rceil \lceil e_y/2+1 \rceil$ 1d IFFTs along the original z-direction, resulting in a complex array of size $(e_z, \lceil e_x/2+1 \rceil, \lceil e_y/2+1 \rceil)$.

(i) cut M_i along the original z-direction and rotate it, resulting in a complex array of size $(\lceil e_y/2+1 \rceil, \lceil e_x/2+1 \rceil, n_z)$.

(j) by looping along the original x- and z-directions, perform $\lceil e_x/2+1 \rceil n_z$ 1d complex-to-real FFTs along the original y-direction, resulting in a complex array of size $(e_y, \lceil e_x/2+1 \rceil, n_z)$.

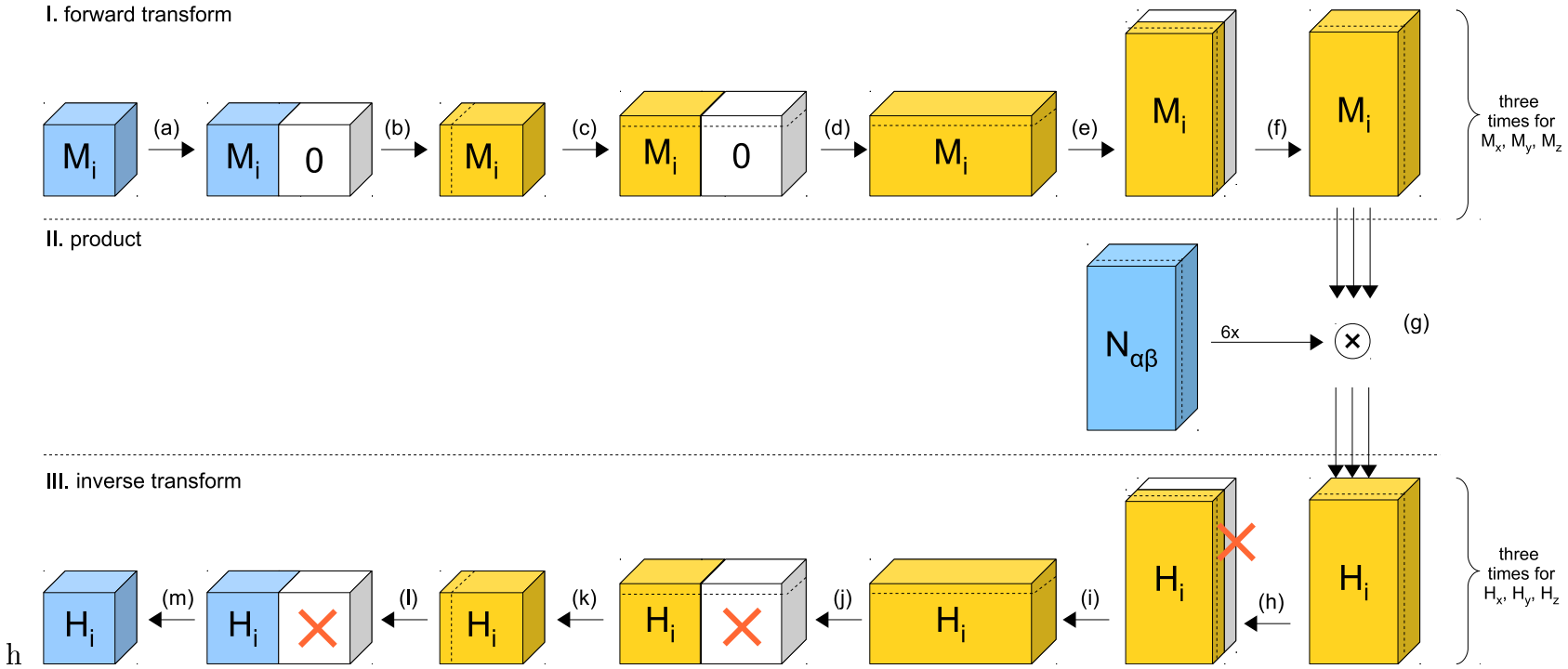
(k) cut M_i along the original y-direction and rotate it, resulting in an array of size $(\lceil 2/e_x+1 \rceil, n_y, n_z)$.

(l) by looping along the original x- and y-directions, perform $n_y n_z$ 1d complex-to-real FFTs along the original x-direction, resulting in a complex array of size (e_x, n_y, n_z).

(m) cut M_i along the x-direction, resulting in an array of size (n_x, n_y, n_z).

The behavior of the subroutines (a)–(i) for the 2-D case and (a)–(m) for the 3-D case are defined in appendix A.2, see Tab. 3.2.

Periodic boundary conditions If periodic boundary conditions in one or more dimensions are enabled, the cyclicality of the fast convolution can be exploited. To do this, the zero-padding of the magnetization array in the periodic directions are omitted[57]. The size of the demagnetization tensor in the periodic directions is then equal to the size of the unpadded magnetization array[76, 77]. Thus the computation time is greatly reduced as the number of data points in the fast Fourier transforms is halved for each additional periodic direction. The 2-D and 3-D algorithms displayed in Fig. 3.5 and in



h

Figure 3.6: (a) – (m) Computational steps to compute the demagnetization field $\vec{H}$ from the magnetization $\vec{M}$ on a two-dimensional array of simulation cells. The filled rectangles denote two-dimensional number arrays that serve as the input and output of the steps. Blue arrays contain real-valued numbers, and orange arrays contain complex-valued numbers.

Fig. 3.6 can be directly used with periodic boundary conditions. The only difference is the setup of the demagnetization tensor array $N_{\alpha\beta}$, which is computed according to an approximation[57] of Eq. 2.27, and the size of the zero-padded magnetization (e_x, e_y, e_z).

3.1.4 Scalar potential method

For the potential field Eq. 3.24 is computed using fast convolutions. Here the three-dimensional algorithm from the previous subsection can be employed, with the difference that only one inverse transform is needed to get the potential field Φ, and that the product in the frequency domain is a scalar product of vectors. As the inverse Fourier transform is only applied once to obtain the scalar field (as opposed to three times for each component of the demagnetization field), the scalar potential method requires only four instead of six transforms. In principle all discussed optimizations for the tensor method are possible here as well. After the potential field is calculated, its gradient yields the demagnetization field. In MicroMagnum, the gradient is computed using the finite differences between the eight cell vertices. First, the scalar potential is calculated on all cell vertices. Then, for each cell and for each component of the gradient vector, the finite differences between four pairs of vertices are averaged, yielding an approximation of the demagnetization field at each cell. As the gradient computation has linear run time, the convolution dominates and the total run time is in $\mathcal{O}(n \log n)$. Compared to the tensor field method, a speedup of up to 50 % can be expected because only four instead of six forward and inverse transforms are required.

3.2 Exchange field

The exchange field in Eq. 2.6 is computed by applying a weighted Laplacian on each component of the magnetization array by using finite differences. In case of periodic boundary conditions and constant exchange stiffness A and saturation magnetization M_s at each cell, the weights are the same at all cells, and the discrete weighted Laplace operator can be expressed as a discrete multidimensional convolution with a fixed kernel. For a mesh containing a 2-D array of cells, each cell has six neighbors and the kernel becomes

$$L(l_x, l_y) = \frac{1}{l_x^2} \begin{pmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{pmatrix} + \frac{1}{l_y^2} \begin{pmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \tag{3.26}$$

where l_x and l_y is the size of each cell. For three-dimensional meshes, $L(l_x, l_y, l_z)$ has the dimensions $3 \times 3 \times 3$ and is defined similarly. Because the convolution is linear, the exchange field can be computed together with the demagnetizing field by adding the kernel L into the demagnetization tensor field, resulting in a combined field of symmetric

tensors

$$\mathbf{N}' = \begin{pmatrix} N_{xx} & N_{xy} & N_{xz} \\ N_{xy} & N_{yy} & N_{yz} \\ N_{xz} & N_{yz} & N_{zz} \end{pmatrix} - \frac{2A}{\mu_0 M_s^2} \begin{pmatrix} L & 0 & 0 \\ 0 & L & 0 \\ 0 & 0 & L \end{pmatrix}. \tag{3.27}$$

Here L must be appropriately zero-padded in all directions so that the center entry of L is added to the self-demagnetization tensor terms. As the combined tensors are precomputed before the simulation, the exchange field can be computed "for free" along with the demagnetization field. This method works well together with the inclusion of periodic boundary conditions (see section 3.1.3). Because the periodic dimensions are not zero-padded in the demagnetization tensor array, the convolution becomes periodic in that dimension, which is what is wanted for the periodic exchange field computation as well. A drawback of the combined computation is that it can not be used for simulations where the material parameters A and M_s vary between the simulation cells. Also, the demagnetization field and exchange field and their energies are not computed separately, although the user might be interested in these energies. It is not possible to integrate the exchange field computation into the scalar potential computation described in section 2.1.4, as the exchange field cannot be formulated as a gradient of a scalar field.

3.3 Oersted field

As show in Ref. [60], a discretized version of the Biot-Savart equation in Eq. 1.25 on regular, rectangular grids can be expressed as a convolution of the current density with a field of antisymmetric tensors,

$$\vec{H}_{\text{Oersted}}(\vec{r}_k) = \sum_{l=0}^{N-1} \mathbf{K}(\vec{r}_k - \vec{r}_l) \cdot \vec{j}(\vec{r}_l) = \left(\mathbf{K} * \vec{j}\right)(\vec{r}_k), \tag{3.28}$$

where N is the number of cells. The convolution can be efficiently computed using fast Fourier transforms in $\mathcal{O}(n \log n)$ time. The 3×3 antisymmetric tensor K is defined as

$$\mathbf{K}_{ij}(R) = \frac{2}{4\pi V} \int_V d^3r \int_{V'} d^3r' \frac{R + r - r'}{|R + r - r'|^3} \cdot (\vec{e}_i \times \vec{e}_j) \tag{3.29}$$

for a pair of cells with the cell volumes V and V', the cell distance $R = |\vec{r} - \vec{r}'|$, and the coordinate axes $\vec{e}_i$ and $\vec{e}_j$, $i, j \in \{\text{x,y,z}\}$. This integral is numerically evaluated[60]. All optimizations for the demagnetization field computations apply for the Oersted field computation as well. The Oersted field can be calculated along the demagnetization field by sharing the computation of the inverse Fourier transforms, resulting in a combined

field,

$$
\begin{aligned}
-\hat{H}^*_{\mathrm{combined,x}} &= \hat{\mathbf{N}}_{\mathrm{xx}} \cdot \hat{M}^*_{\mathrm{x}} + \hat{\mathbf{N}}_{\mathrm{xy}} \cdot \hat{M}^*_{\mathrm{y}} + \mathbf{N}_{\mathrm{xz}} \cdot \hat{M}^*_{\mathrm{z}} - \hat{\mathbf{K}}_{\mathrm{xy}} \cdot \hat{j}^*_{\mathrm{y}} - \hat{\mathbf{K}}_{\mathrm{xz}} \cdot \hat{j}^*_{\mathrm{z}} \\
-\hat{H}^*_{\mathrm{combined,y}} &= \hat{\mathbf{N}}_{\mathrm{yx}} \cdot \hat{M}^*_{\mathrm{x}} + \hat{\mathbf{N}}_{\mathrm{yy}} \cdot \hat{M}^*_{\mathrm{y}} + \mathbf{N}_{\mathrm{yz}} \cdot \hat{M}^*_{\mathrm{z}} + \hat{\mathbf{K}}_{\mathrm{xy}} \cdot \hat{j}^*_{\mathrm{x}} - \hat{\mathbf{K}}_{\mathrm{yz}} \cdot \hat{j}^*_{\mathrm{z}} \\
-\hat{H}^*_{\mathrm{combined,z}} &= \hat{\mathbf{N}}_{\mathrm{xz}} \cdot \hat{M}^*_{\mathrm{x}} + \hat{\mathbf{N}}_{\mathrm{yz}} \cdot \hat{M}^*_{\mathrm{y}} + \mathbf{N}_{\mathrm{zz}} \cdot \hat{M}^*_{\mathrm{z}} + \hat{\mathbf{K}}_{\mathrm{xz}} \cdot \hat{j}^*_{\mathrm{x}} + \hat{\mathbf{K}}_{\mathrm{yz}} \cdot \hat{j}^*_{\mathrm{y}}.
\end{aligned} \tag{3.30}
$$

Here $\hat{j}^*_{x/y/z}$ denotes the zero-padded, fast Fourier transformed current density array. The combination of the field reduces the additional computation time of the Oersted field to roughly 50% of the original time.

3.4 Current paths

The system of linear equations in Eq. 2.45 is solved by using an iterative Krylov subspace solver as described in Ref. [61].

Part II

Parallel Computation of the Micromagnetic Model

This part contains two chapters. In the first chapter parallel algorithms are shortly introduced and applied to the computation of the Landau-Lifshitz-Gilbert equation. The second chapter deals with the parallel implementation of the micromagnetic model on CUDA graphics processing units.

4 Parallel computing

A performance gain can be obtained by executing algorithms in parallel. If the computational problem can be divided into subproblems, each subproblem can be computed by a single process to save computation time. A prerequisite for parallel computation is the absence of a causal dependency between the subproblems. Parallel computation is most efficient if the processes are executed on multi-core computers. In case of GPU computing a large number of cores access a single memory, which is efficient for executing a single instruction on different data. This kind of computation is required for example at the fast Fourier transform at the computation of the demagnetization field in the micromagnetic model. In chapter 7.3 it is shown that the parallel demagnetization field computation on a GPU leads to a speedup of up to 66 in comparison to the sequential computation on a CPU.

In a parallel program, two or more processors perform computations simultaneously in order to solve a computational task. The motivation is to combine the resources of the parallel processors and make the computation more efficient, or even possible. In the case of micromagnetic simulation the prime motivation is to simulate large samples with high spatial resolution in adequate time. Here the run time speedup gained by parallel computing is most important.

In the first part of this chapter important concepts of parallel computing are introduced. This includes performance metrics which are later used to benchmark the developed parallel algorithms. In the second part of this chapter it is described how the finite-difference computation of the micromagnetic model, i.e. the Landau-Lifshitz-Gilbert equation including all effective field terms, can be expressed as a parallel algorithm.

4.1 Fundamentals

A parallel program is executed by processes that run simultaneously, where each process performs a part of the total computation. Thus, when a parallel algorithm is developed, at least two concepts have to be considered: How the processes interact by communication, and how the computations are partitioned across the processes.

Interaction of the Parallel Processes The *shared-memory* model and the *message-passing* model are considered.

In the *shared-memory model*, the parallel processes operate on the same (shared) working memory. From the operating system view, these processes are called *software threads*. Threads usually run on the same computer and have access to a common part of the system memory. Commonly used programming environments in parallel numerical computing with threads are OpenMP[78] and PThreads[79], although many others exist. In GPGPU computing, every procedure that runs on the GPU is inherently parallelized using threads. In this thesis the CUDA programming environment[35] by Nvidia is used. The *OpenCL*[80] environment targets threaded programming on both CPUs and GPUs.

In the *message passing model*, the parallel processes communicate via interprocess messages. The messages have to be programmed explicitly by the developer. Typically the data transfer rate and the transfer latency are performance bottlenecks of the parallel algorithm. On the operating system level, the processes are usually implemented via *operating system processes*. On a computer cluster, the processes may be distributed across the nodes. Message passing is done via the network or, when two communicating processes run on the same node, via copies in the system memory. A commonly used parallel programming environment is the message passing interface (MPI[81]).

Hardware and software threads A *thread* is the sequence of executed processor instructions. A *software thread* corresponds to the flow of execution of a running program. Software threads are managed by the operating system and are scheduled to run on the hardware. A *hardware thread* is defined by the sequence of instructions that is executed by a compute core on the processor. In case of a multi-core CPU processor each core processes one hardware thread. A modern GPU can contain hundreds of hardware threads.

Partitioning of the computation The partitioning of the computation into concurrent parts can be distinguished into two kinds, *data* level parallelism and *task* level parallelism.

- Data-level parallelism: The data is divided into chunks. Each parallel process operates on one of these chunks.

- Task-level parallelism: The sequential program is divided into independent subprograms. Each parallel process executes one of these subprograms.

As an example the per-element addition of two arrays of length n is considered. Both arrays can be split into two subsequences of length $n/2$ and then be added concurrently by two data-parallel processes, each of which processes one half of the arrays. Both processes essentially execute the same algorithm, but process different chunks of data. In contrast, task-level parallelism deals with the parallel computation of different subprograms. Typically a parallel program makes use of both data and task level parallelism.

Performance measurements The following notation is taken from the book *Introduction to Parallel Computing* by Grama et al.[82]. In order to assess the efficiency of a parallel algorithm, its run time for a given number of processors is measured and compared to the run time of a respective sequential algorithm. The *parallel run time* on a computer with n processors is called $T_{\text{par}}(n)$, and the *sequential run time* of the best sequential algorithm is called T_{seq}. The *speedup* S of a parallel algorithm that is executed on n processors is defined as the ratio of the run time of the best respective sequential algorithm by the time taken by the parallel algorithm:

$$S = \frac{T_{\text{seq}}}{T_{\text{par}}(n)} \tag{4.1}$$

For a meaningful comparison, all run time measurements should be conducted on the same hardware. Sometimes this is not possible, e.g. when the run times of algorithms on CPU and GPU architectures are compared. The maximal theoretical speedup is limited by *Amdahls law*. It divides the algorithm into two parts, a parallelizable part and a sequential non-parallelizable part. It is assumed that the sequential part cannot be sped up no matter how many parallel processors are devoted to the overall computation. If P is the parallelizable fraction and $(1 - P)$ the sequential fraction of the algorithm, the speedup for n parallel processors is modeled as

$$S(n) = \frac{1}{(1 - P) + P/n}. \tag{4.2}$$

The maximal theoretical speedup is thus $S_{\max} = 1/(1 - P)$ for $n \to \infty$. It is limited by the sequential fraction of the algorithm. Given a measured speedup of $S^*(n)$ on n processors, the parallel fraction can be estimated using

$$P^* = \frac{1/S^*(n) - 1}{1/n - 1}. \tag{4.3}$$

Much of the development time for parallel algorithms is devoted to make P as high as possible. As given by Amdahls law, doubling the number of processors rarely halves

the run time of a parallel algorithm. The *efficiency* $E(n)$ of a parallel algorithm on n processors is defined as the ratio of the speedup and n,

$$E(n) = \frac{S(n)}{n}. \tag{4.4}$$

4.2 Parallel micromagnetic model computation

In order to simulate the magnetization dynamics the Landau-Lifshitz-Gilbert equation has to be computed repeatedly. From a computational point of view, the magnetization field is inserted into the LLG equation in order to retrieve its time derivative, i.e. $\mathrm{d}\vec{M}/\mathrm{d}t = f(\vec{M})$ where f is the LLG equation. This section deals with the parallel computation of $f(\vec{M})$ with a given $\vec{M}$.

4.2.1 Effective field

The LLG equation includes the total magnetic field, which is composed of several terms, e.g.

$$\vec{H}_{\mathrm{tot}} = \vec{H}_{\mathrm{exch}} + \vec{H}_{\mathrm{demag}} + \vec{H}_{\mathrm{aniso}} + \vec{H}_{\mathrm{ext}} + \dots. \tag{4.5}$$

Neither of these terms depend on any time derivative of the magnetization, so that each term can be computed simultaneously. The computation of each term in parallel is an example of task level parallelism. In this case however the computation times of the subtasks are not evenly distributed – the demagnetization field alone takes at least 60 % of the total run time. Thus dividing the work evenly across more than two processors becomes difficult. Similarly, in the extended LLG equation, the spin torque term can be computed in parallel to the other terms.

$$\frac{\mathrm{d}\vec{M}}{\mathrm{d}t} = LLG(\vec{M}, \vec{H}) + ST(\vec{M}, \vec{j}) \tag{4.6}$$

In order to compute the spin torque term the current density and the magnetization field must be known. If the current density is computed dynamically from the magnetization field using the current path equations the spin torque term can take a considerable amount of computation time, and a higher speedup can be expected.

4.2.2 Convolution-based field terms

The demagnetization field and the Oersted field are the most time-expensive terms of the effective field term. Both are based on a convolution which is computed via fast Fourier transforms. In this section possible parallelization opportunities are discussed.

Parallel field component transforms The computation of the demagnetization field involves the fast Fourier transform of each magnetization field component M_x, M_y and

M_z. Due to the independence of these transformations they can be computed in parallel, see Fig. 3.5 and Fig. 3.6. Similarly, the inverse transforms that result in the demagnetization field components H_x, H_y and H_z can be computed independent of each other. This is a type of data level parallelism.

Parallel fast Fourier transforms The report in Ref. [83] identifies common numerical tasks in scientific computing and classifies them into thirteen groups in terms of their parallel implementation characteristics. Tasks in one class exhibit common parallel computation and communication patterns. The multidimensional fast Fourier transform was selected as the prototypical member of the spectral-methods group. It requires all-to-all communication between distributed compute nodes and is therefore a memory-bound algorithm. In the micromagnetic model two- and three-dimensional transforms are required. Within the row-column algorithm (see section 3.1.1), all iterated one-dimensional transforms within the steps shown in Fig. 3.5 and Fig. 3.6 can be executed concurrently. Here the array rotation steps cause the aforementioned all-to-all communication between the nodes.

There are two basic parallel one-dimensional fast Fourier transform algorithms, namely the binary-exchange algorithm and the transpose algorithm (see Ref. [82] for a detailed description). However, typically the individual 1-D transforms needed for micromagnetic simulations are too small to warrant the parallel overhead. In the case of a micromagnetic simulator, only two- and three-dimensional transforms are required and as such are much easier parallelized by a parallelized row-column transform algorithm. The parallel row-column algorithm is a data-parallel method.

Parallel multiplication After the magnetization array is transformed into the frequency domain, each element has to be multiplied with the corresponding element of the transformed demagnetization tensor. All products are independent of each other and can thus be computed in parallel, exhibiting data level parallelism.

4.2.3 Local field terms

In any simulation cell, the local fields depend only on the cells in their immediate neighborhood. This means that the local fields can be computed at each cell independently and in parallel. For example, the exchange field, the anisotropy field and the external field are local, whereas the demagnetization field and the Oersted field are not. Parallelization of the local fields involve its computation at each simulation cell in parallel. This is again a type of data level parallelism.

5 Computation on Graphics Processing Units

Graphics processors have evolved into general-purpose multiprocessors that are capable of highly parallel computations. The development is driven by the demand to produce high definition real time imagery. Graphics cards that accelerate the generation of three-dimensional real time images became commercially available in the middle of the 1990s. The processors on these graphics cards mainly provided hardware-accelerated rasterization of textured triangles. Many graphics cards included on-board memory on which the texture data could be stored in order to circumvent the PCI bottleneck. Later processors added the capability to perform geometric transforms and lightning calculation on the device. In order to perform these calculations, the graphics processors were designed to operate massively in parallel. Real-time imaging also required a high memory bandwidth between the processor and its on-board memory. The introduction of shader languages[84, 80, 85] in the 2000s allowed the execution of programmer-defined algorithms on the graphics processors. Using the shader language, numerical computations could be, for example, formulated in terms of texture transforms inside the graphics processors' texturing unit. However, as the shader languages were primarily designed to produce graphics effects for computer games, they were difficult to exploit for general-purpose programming. In the late 2000s the major graphics card manufacturers began to make their processors more accessible and provided tools to allow the use of general-purpose programming languages. The CUDA compute architecture [35] developed by Nvidia allows the programmer to develop his GPU algorithms in a superset of the C programming language. Similarly, the Stream SDK from AMD allows free programming using OpenCL, another extension of C. The increase of the performance of GPUs in recent years is shown in Fig. 5.1.

5.1 CUDA programming model

CUDA provides a standard model for general computing on graphics processing units across the graphics card produced by Nvidia. CUDA extends the C and C++ programming languages with new language primitives suitable for its parallel computation model. The CUDA developer kit includes CUBLAS[86], an implementation of the basic linear algebra subprograms[87], CUFFT[88], an implementation of fast Fourier transforms, and several other libraries targeted at statistical and financial applications. A good overview of the CUDA programming model and its application to numerous mathematical problems such as numerical linear algebra and fast Fourier transforms is given in Ref. [89].

The first CUDA-enabled GPUs were the line of G80 processors, which became first available on the Geforce 8800 line of graphics cards, and later, as server hardware targeted at scientific computing, on the Tesla C810 graphics cards. One GPU contains one or more streaming multiprocessors (MP). Each multiprocessor contains a number of

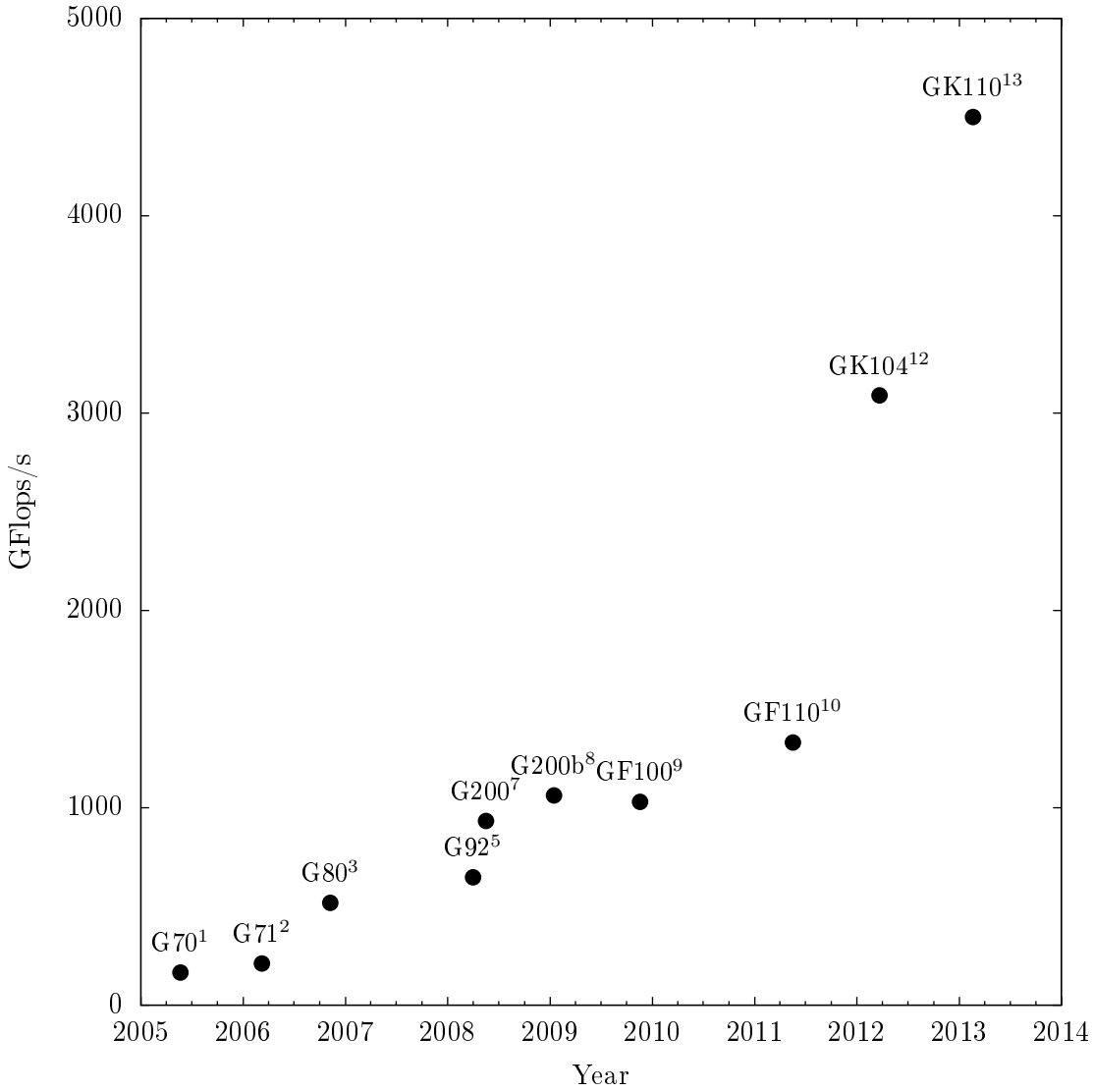


Figure 5.1: Theoretical peak 32 bit floating point operations per second (FLOPS) of several Nvidia graphics processing units over the last years. The footnotes indicate the corresponding graphics card models.

[1]Geforce 7800 GTX
[2]Geforce 7900 GTX
[3]Geforce 8800 GTX / Tesla C870
[4]Geforce 8800 Ultra
[5]Geforce 9800 GTX
[6]Geforce 9800 GTX+
[7]Geforce GTX 280
[8]Geforce GTX 285 / Tesla C1060
[9]Tesla M2050 / M2070 / C2050 / C2070
[10]Tesla M2090 / Geforce GTX 580
[12]Geforce GTX 680
[13]Geforce GTX Titan

parallel compute cores which are called streaming processors (SP) (see Fig. 5.2). For example, the G200 chip on the Tesla M2050 card contains at total of 448 compute cores inside 14 multiprocessors with 32 streaming processors each. Each streaming processor contains an arithmetic logic unit (ALU). All streaming processors within an individual multiprocessor are connected to one control unit and thus always share a common code path, i.e. a multiprocessor processes multiple data with a single instruction. This kind of parallelism is called *single instruction/multiple data* (SIMD) in the Flynn taxonomy of parallel computers[90]. However, the individual multiprocessors may execute different code paths at the same time, as each multiprocessor contains its own control unit.

Besides the control unit, the streaming processors inside one multiprocessor also share a common memory, the so-called *shared memory*. This memory is only a few kilobytes small, but very fast. It is used for the communication between the streaming processors without involving the much slower global memory. The shared memory can also be used as a programmer-managed cache memory.

In order to achieve high performance, parallel global memory accesses by different streaming processors can be satisfied by combining them into a single memory block transfer between the GPU and the global memory. The combined block transfer is called *coalesced*. This is only possible when certain criteria are met. On a multiprocessor, there are always 32 threads that are executed concurrently by its stream processors. This package of 32 threads is called a "warp". A warp is further divided into two half-warps, each comprised of the first 16 and the last 16 threads of a warp, respectively. At CUDA compute capability greater than 2.0, the concurrent memory accesses of one half-warp are coalesced if all their addresses fall into an interval that fits a memory block transfer[1]. If these criteria are not or only partially met, two or more memory block transfers are caused, resulting in a lower overall performance. It is the programmers responsibility to schedule the memory accesses in such a way that as few as possible block memory transfers are caused.

Graphics processors are able to hide memory latency by fast thread switching. On the G80 GPU, global memory accesses are not cached and involve a latency penalty of hundreds of clock cycles. The G80 GPU is able to run 448 hardware threads in parallel. The threads are light-weight, and scheduling is very fast. In order to hide memory latency, software threads that are waiting for a memory access to complete can be suspended with a very low overhead, leaving the processing power to the active threads. Later GPUs like the Fermi GF100 GPU targeted at scientific computing, add a L1 and a L2 cache in order to reduce latency. However, enabling the L1 cache reduces the amount of shared memory available to the program and thus potentially the number of threads that can be executed at the same time in parallel by the multiprocessor for a given program.

[1]GPUs with compute capabilities less than 2.0 have an even stricter requirement for coalescing memory accesses.

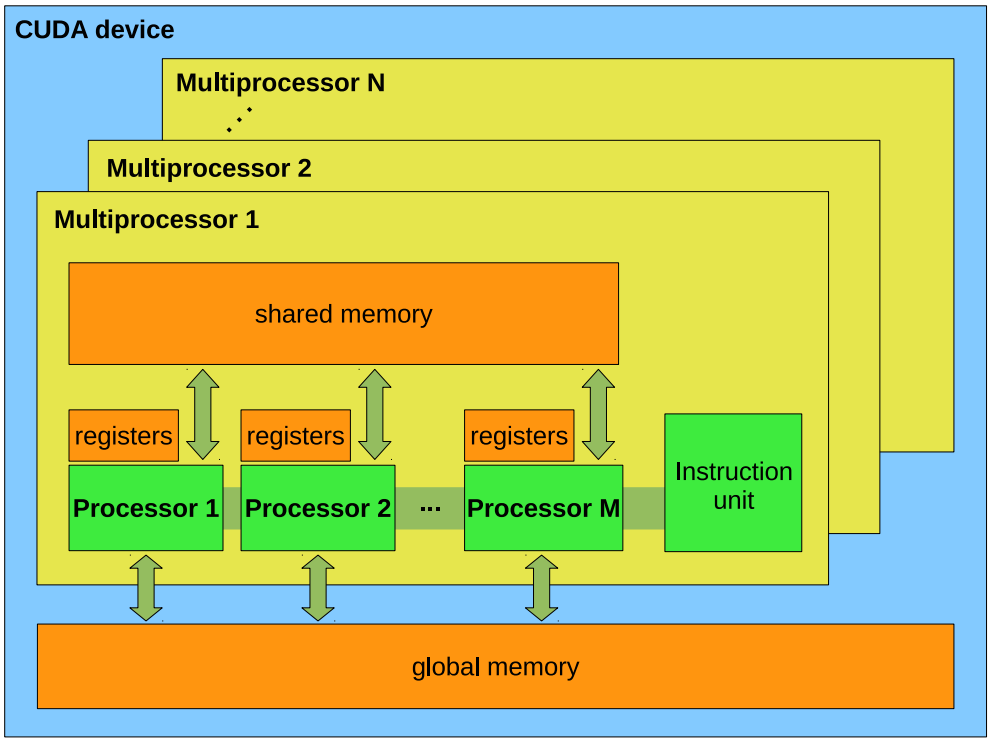


Figure 5.2: Hardware architecture of the graphics processing unit of Nvidia graphics card. The GPU consists of several multiprocessors (MP, yellow) that contain multiple streaming processors (SP, green) that share an instruction unit. Each MP has a shared memory, and each SP has a set of registers. The GPU is connected to the global memory. (Adapted from *CUDA programming guide*[35].)

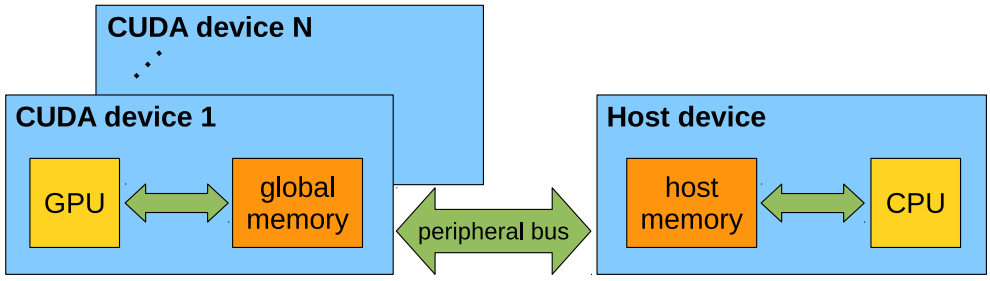


Figure 5.3: Multi-GPU setup where multiple graphics cards are installed on a host computer. The GPUs and the host system communicate via the peripheral bus (PCI express bus).

5.2 Micromagnetic model implementation in CUDA

For the solution of the micromagnetic model using the finite difference method, the following numerical operations need to be implemented on the GPU:

- operations implemented by looping over the elements of arrays
  - addition of two arrays, scaling of an array, etc.
  - computation of fields local to one cell, e.g. the anisotropy field
  - computation of the sum/average of all elements in an array
- operations needed for the fast convolution of the demagnetization field / Oersted field computation:
  - Array rotation
  - Matrix-vector product
  - Iterated fast Fourier transforms
- convolution of a 2-D or 3-D array with a small fixed-size convolution kernel for the computation of the exchange field.

In the following implementation strategies for the mentioned operations on the GPU are presented.

5.2.1 Parallel loop over array elements

Simple array operations like the element-by-element addition of two arrays can be implemented by a loop:

```
1 void add(float *dst, const float *src, int N)
2 {
3   for (int i=0; i<N; ++i) {
4     dst[i] += src[i];
5   }
6 }
```

Listing 4: Addition of two arrays in C

Here the array `src` is added to the array `dst`. Simple loops like this are easily parallelized by executing multiple iterations of the loop at the same time. The following CUDA function uses 32 thread blocks with 128 threads for a total of $32 \cdot 128 = 4096$ potentially parallel threads. If the length of the arrays is N, each thread executes $N/4096$ iterations of the loop.

```
1 const int GRID_SIZE = 32, BLOCK_SIZE = 128;
2
3 __global__
```

```
4  void add_kernel(float *dst, const float *src, int N)
5  {
6    const int tid = blockDim.x * blockIdx.x + threadIdx.x;
7
8    for (int i=tid; i<N; i+=GRID_SIZE*BLOCK_SIZE) {
9      dst[i] += src[i];
10   }
11 }
12
13 void add(float *dst, const float *src, int N)
14 {
15   // call CUDA kernel with a total of 32*128 = 4096 threads.
16   add_kernel<<<GRID_SIZE, BLOCK_SIZE>>>(dst, src, N);
17 }
```

Listing 5: Addition of two arrays in CUDA C

The CUDA function `add_kernel` is executed by each thread. In line 6, the thread index from 0 to 4095 is calculated. Each thread then enters the loop in lines 8—10. The loop is parallely executed by the threads in an interleaved fashion, see line 8. This ensures that the accesses to global memory are coalesced, because each half-warp of threads accesses memory addresses that lie adjacent in memory.

In MicroMagnum, parallel loops on the GPU are used to implement the following simple array operations:

- element wise array addition, subtraction, multiplication, division
- assign constant value to each array element
- normalize an array of vectors
- computation of fields local to one simulation cell: uniaxial and cubic anisotropy field, external field

5.2.2 Reduce operation

In functional programming, the *reduce* function traverses the elements of some data structure and builds up a result using a combining function. For example, a reduce function could iterate over a sequence of n numbers $p_0, p_1, \ldots p_n$ and use as the combining function a left-associative binary operation $\otimes$ to give the result $(p_0 \otimes (p_1 \otimes (p_2 \otimes \ldots) \ldots)$. If the combining function is associative, the *reduce* operation can be naturally executed using a parallel algorithm. The input array is distributed among n processes and the reduce operation is computed in parallel on each subset, resulting in an intermediate result array of size n. The parallel reduce operation is now applied recursively on the intermediate result until one final result remains.

In numerical computing, many operations on arrays can be formulated in terms of a reduce function. For example, the sum of the elements of an array is the array reduction

using the function $a \otimes b \mapsto a + b$. The parallel reduce function is efficiently implemented for CUDA using the "parallel reduce pattern"[91]. Here, a CUDA kernel is repeatedly executed on an input array, generating the output for the next repetition. Each thread block generates only one element in the output array, thereby reducing the input step by step using the combining function. The kernel is called until only one element remains, which is the end result.

In MicroMagnum, the following array operations are implemented using the parallel reduce pattern:

- Compute sum/average of the array elements.
- Find the smallest/largest element in array.
- Compute the scalar product of two arrays.
- Find the vector with the largest magnitude in a vector array.

As an example, listing 6 shows how the sum of an array can be computed using the parallel reduce pattern with CUDA.

```
1  __global__
2  static void kernel_sum_reduce(const float *in, float *out, int N)
3  {
4    __shared__ float sh[256]; // thread block size is (256,0,0)
5    const unsigned int bid = gridDim.x * blockIdx.y + blockIdx.x;
6    const unsigned int tid = threadIdx.x;
7    const unsigned int i = bid * 256 + tid;
8
9    if (i < N) {
10     sh[tid] = in[i];
11   } else {
12     sh[tid] = 0.0;
13   }
14   __syncthreads();
15
16   if (tid < 128) sh[tid] += sh[tid+128];
17   __syncthreads();
18
19   if (tid <  64) sh[tid] += sh[tid+64];
20   __syncthreads();
21
22   if (tid <  32) {
23     volatile float *smem = sh;
24     smem[tid] += smem[tid+32]; smem[tid] += smem[tid+16]; smem[tid] +=
         smem[tid+ 8];
25     smem[tid] += smem[tid+ 4]; smem[tid] += smem[tid+ 2]; smem[tid] +=
         smem[tid+ 1];
26   }
27
28   // write result for this block to global memory
```

```
29   if (tid == 0) out[bid] = sh[0];
30 }
```

Listing 6: Add up all numbers in an array in CUDA C

The `kernel_sum_reduce` kernel takes an input array `in` of size `N` and write an intermediate reduce result to the output array `out` of size `N/256`. It is designed to run on `N/256` threads, so that each thread reduces 256 elements of the input array and produces one intermediate result, which is the sum of the 256 elements. In order to compute the sum of an array of arbitrary size, the reduce kernel is called multiple times from C++ in the following listing 7.

```
1  float cuda_sum(const float *src, int N)
2  {
3    // Allocate temporary storage
4    float *buf1, *buf2;
5    alloc_reduce_buffers(&buf1, &buf2, N);
6
7    // First iteration
8    {
9      const int B = (N+255)/256;
10     kernel_sum_reduce<<<B, 256>>>(src, buf1, N);
11     N = B;
12   }
13
14   // Remaining iterations
15   float *in = buf1, *out = buf2;
16   while (N > 1) {
17     const int B = (N+255)/256;
18     kernel_sum_reduce<<<B, 256>>>(in, out, N);
19     N = B; std::swap(in, out);
20   }
21   checkCudaLastError("kernel_sum_reduce() execution failed");
22
23   const float result = download_float(in); // result is stored at in[0]
24   free_reduce_buffers(&buf1, &buf2);
25   return result;
26 }
```

Listing 7: The C++ code that calls the `kernel_sum_reduce` CUDA function.

Two buffers called `buf1` and `buf2` are allocated. They hold the intermediate results of the reduce operation. For the first iteration, the input array is partially reduced to the `buf1` buffer of size $N/256$. For the remaining iterations, the output of the last iteration is reduced repeatedly until only one element remains, which is the array sum. The two buffers are used as intermediate input/output buffers.

5.2.3 Array rotation

The full 2-D and 3-D algorithms presented in section 3.1.3 in Figs. 3.5 and 3.6 internally perform array rotations in memory for the fast convolution. They are required for fast Fourier transforms that operate on data points that lie contiguous in memory. All arrays are stored in FORTRAN order with minimal stride, i.e. if an array has the size (N_x, N_y, N_z), the element at (x, y, z) has the linear index $i = x + yN_x + zN_xN_y$. The array rotation thus permutes the dimensions of input array. In particular, there are two rotation directions required,

1. rotate left: permute dimensions from (N_z, N_x, N_y) to $(\mathrm{N_x}, \mathrm{N_y}, \mathrm{N_z})$ (see listing 22 for a definition)

2. rotate right: permute dimensions from (N_y, N_z, N_x) to $(\mathrm{N_x}, \mathrm{N_y}, \mathrm{N_z})$ (see listing 23 for a definition)

Additionally, after the left rotation, the output has to be zero-padded along N_x-direction, and after the right rotation the input array has to be unpadded. In the following, only the implementation of the 2-D case is discussed. Here both types of rotations correspond to the same matrix transposition. To allow for unpadding and/or later zero-padding, the transposition function accepts an arbitrary stride length for the y-dimension for both the input and the output array. By using appropriate values for the strides array contents that will be unpadded/zero-padded can be skipped from the input/output.

The `kernel_transpose_2d` in listing 8 function transposes a 2-D array on the GPU. It takes the dimensions of the input array (`dim_x`, `dim_y`), pointers to the input and output buffers (`in`, `out`), and the input and output strides in y-direction (`in_stride_y`, `in_stride_x`; in units of `2*sizeof(float)`). A fixed input/output stride of 1 in the x-direction is assumed. As `kernel_transpose_2d` is a CUDA kernel, it is executed concurrently for each entry of the input array. Each thread block copies an one tile of size 16×16 to the output tile in transposed order. A thread block has 256 threads. Accesses to global memory are made coalesced by actually performing the transposition in the shared memory.

```
__global__
void kernel_transpose_2d(
  const int dim_x, const int dim_y,
  const float * in, const int  in_stride_y,
        float *out, const int out_stride_y)
{
  const int Q = 2; // Improves shared memory bank conflicts.
  __shared__ float sh[2*16*(16+Q)];

  // Move to source and dest tiles.
  const int base_x = 16*blockIdx.x, base_y = 16*blockIdx.y;
  in  += 2*(base_x +  in_stride_y*base_y); // Src  tile @ (base_x,
     base_y)
```

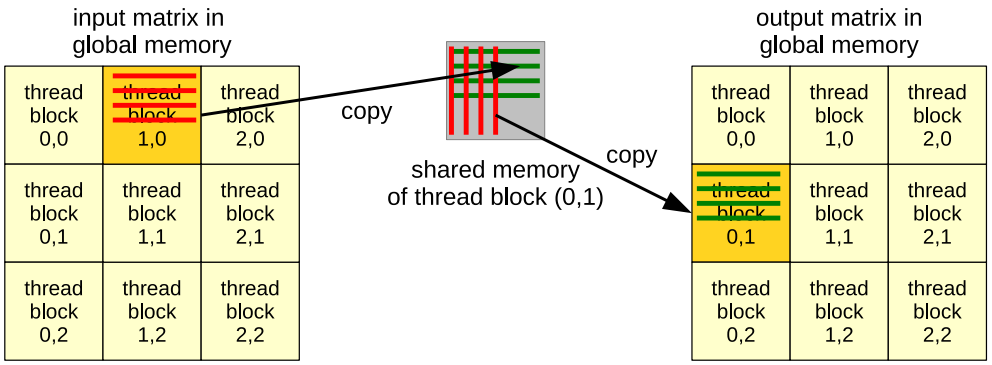


Figure 5.4: Implementation of the matrix transposition algorithm on CUDA C using the shared memory. Yellow boxes: Thread blocks, which are assigned to matrix tiles. Grey boxes: Shared memory area of a thread block. Red lines: Memory read operations. Green lines: Memory write operations. Horizontal lines represent coalesced memory transfers.

```
13   out += 2*(base_y + out_stride_y*base_x); // Dest tile @ (base_y,
         base_x)
14
15   const bool at_border = (   blockIdx.x == gridDim.x-1
16                           || blockIdx.y == gridDim.y-1);
17   if (!at_border) {
18     read_and_store_transposed_2d<16>(in, in_stride_y, sh, 16+Q);
19     __syncthreads();
20     read_and_store_2d<16>(sh, 16+Q, out, out_stride_y);
21   } else {
22     in  += 2*(threadIdx.x +  in_stride_y*threadIdx.y);
23     out += 2*(threadIdx.y + out_stride_y*threadIdx.x);
24     if (base_x + threadIdx.x < dim_x && base_y + threadIdx.y < dim_y) {
25       out[0] = in[0];
26       out[1] = in[1];
27     }
28   }
29 }
```

Listing 8: Matrix transposition algorithm in CUDA C.

In lines 11—13, the input and output buffer pointers are translated to the source and destination tile addresses. In line 15, it is determined whether the current thread block processes a tile that lies at the border of the array. A border tile can overlap the array and is thus processed separately. Inner tiles are copied via the shared memory (lines 18—21), and border tiles are copied with the slower direct copy (lines 22-27), see Fig. 5.4. For the direct copy, the 'in' and 'out' pointers are adjusted again to point at the source and destination entries in the input and output buffer (lines 22—23). The real and imaginary parts of the entry are then copied (lines 25—26), except when the array border was passed over (line 24). The fast shared-memory transposition is implemented in the template functions `read_and_store_transposed_2d` and `read_and_store_2d`.

```
1 template <int block_size_x>
```

```
__device__ __inline__ static void read_and_store_transposed_2d(
  const float * in, const int  in_stride_y,
        float *out, const int out_stride_y)
{
  // 1) Read at (x,y)
  const int x = threadIdx.x, y = threadIdx.y;
  in += 2*y*in_stride_y; out += 2*y;

  // 2) Store at (y,x)
  const int imag = (x & 1) ? (-out_stride_y+1) : 0;
  out[(x+           0)*out_stride_y + imag] = in[x+           0];
  out[(x+block_size_x)*out_stride_y + imag] = in[x+block_size_x];
}

template <int block_size_x>
__device__ __inline__ static void read_and_store_2d(
  const float * in, const int  in_stride_y,
        float *out, const int out_stride_y)
{
  const int x = threadIdx.x, y = threadIdx.y;
  in += 2*y*in_stride_y; out += 2*y*out_stride_y;
  out[x+           0] = in[x+           0];
  out[x+block_size_x] = in[x+block_size_x];
}
```

Listing 9: Helper functions for the array transpose algorithm in CUDA C.

When called from a thread block, the function `read_and_store_transposed_2d` reads a tile using only coalesced memory transfers, and writes the tile in transposed order into a temporary buffer. The template parameter `block_size_x` specifies the size of the thread block in x-direction. As the writes are not coalesced, the output buffer must point to shared memory. The `read_and_store_2d`, when called from a thread block, copies a tile from a temporary input buffer to the output buffer. Here, both the reads and writes are coalesced. Using these two functions, the fast shared-memory transpose is implemented, with the temporary buffer in the shared memory of the calling thread block.

The 3-D version of the transpose is needed for the computation of the demagnetization field with a three-dimensional discretization mesh. Here both the left rotate and the right rotate have to be implemented in different CUDA functions. The rotations can be implemented with coalesced global memory accesses as well, although the required pointer arithmetic becomes quite complex.

5.2.4 Iterated fast Fourier transforms

The full 2-D and 3-D algorithms presented in section 3.1.3 in Figs. 3.5 and 3.6 internally compute iterated 1-D fast Fourier transforms for the fast convolution. Due to the preceding array rotations in memory, the array elements are always laid out linearly in memory along the respective transform direction. Array elements that are complex are

stored in so-called packed order, i.e. the real part and the imaginary part of each element are stored adjacently in two 32-bit float cells. All arrays are stored in FORTRAN order with minimal stride: If an array has the size (N_x, N_y, N_z), the element at (x, y, z) has the linear index $i = x + yN_x + zN_xN_y$. An iterated FFT along the first direction of this array, which is stored continuously in memory, would involve N_yN_z transforms of length N_x, starting at element $(0, 0, 0)$. Due to the minimal stride, each input to the transform is directly followed by the next input in memory.

It is therefore sufficient to implement fast Fourier transforms that perform N iterated forward and inverse transforms of length M, where the input and output sequences are stored linearly after each other and are in packed complex format (see algorithm `iterated_fft` in appendix A.2. FFT algorithms like the iterative radix-2 algorithm presented in listing 3 of section 3.1.1 typically work in-place so that the input data is overwritten with the output data, thus saving memory. Additionally, specialized real-to-complex and complex-to-real iterated FFTs are employed where either the input or the output has real elements, see section 3.1.1.

In MicroMagnum, the computation of the fast Fourier transforms on the GPU is implemented by the CUFFT library[88] which comes with the CUDA SDK[35]. CUFFT supports iterated in-place FFTs, inverse in-place FFTs, out-of-place R2C FFTs, and out-of-place C2R FFTs. CUFFT contains efficient algorithms to compute transforms of lengths that are not a power of two, see also section 3.1.2. An alternative to CUFFT for CUDA GPUs which claims to be faster for some cases is presented in Ref. [92]. An overview of the massively parallel implementation of FFTs on the GPU is given in Ref. [93].

5.2.5 Matrix-vector product

The matrix-vector product is needed for the fast demagnetization field computation, see Eq. 3.23. The CUDA function `kernel_multiplication_symmetric` given in listing 10 receives pointers to the transformed demagnetization tensor arrays `Nxxr`, `Nxyr`, `Nxzr`, `Nyyr`, and `Nyzr`. The transformed tensor is symmetric, see Eq. 2.20, and contains only real values, see section 3.1.2. The matrix-vector product is done in-place on the vector component arrays `Mx`, `My`, and `Mz`. These arrays contain the transformed magnetization, and therefore have complex elements in packed order. As the product is computed for each array element, the dimensionality of the array does not matter, and all arrays are treated as one-dimensional.

```
1 __global__ void kernel_multiplication_symmetric(
2   const float *Nxxr, const float *Nxyr, const float *Nxzr,
3   const float *Nyyr, const float *Nyzr, const float *Nzzr, /*in*/
4   float *Mx, float *My, float *Mz) /*inout*/
5 {
6   extern __shared__ float sh[];
7
```

```
8  const int i_base = 256 * (blockIdx.x + blockIdx.y*gridDim.x);
9  const int i_offs = threadIdx.x; // 0..255
10 const int i      = i_base + i_offs;
11 const int j_base = 2*i_base;
12
13 // (a) load demagnetization tensor
14 const float Nxx_re = Nxxr[i], Nxx_im = 0.0;
15 const float Nxy_re = Nxyr[i], Nxy_im = 0.0;
16 const float Nxz_re = Nxzr[i], Nxz_im = 0.0;
17 const float Nyy_re = Nyyr[i], Nyy_im = 0.0;
18 const float Nyz_re = Nyzr[i], Nyz_im = 0.0;
19 const float Nzz_re = Nzzr[i], Nzz_im = 0.0;
20
21 // (b) copy Mx,My,Mz to shared memory
22 sh[0*256+i_offs] = Mx[0*256+j_base+i_offs];
23 sh[1*256+i_offs] = Mx[1*256+j_base+i_offs];
24 sh[2*256+i_offs] = My[0*256+j_base+i_offs];
25 sh[3*256+i_offs] = My[1*256+j_base+i_offs];
26 sh[4*256+i_offs] = Mz[0*256+j_base+i_offs];
27 sh[5*256+i_offs] = Mz[1*256+j_base+i_offs];
28
29 // (c) load Mx,My,Mz from shared memory
30 __syncthreads();
31 const float Mx_re = sh[0*256+i_offs*2+0];
32 const float Mx_im = sh[0*256+i_offs*2+1];
33 const float My_re = sh[2*256+i_offs*2+0];
34 const float My_im = sh[2*256+i_offs*2+1];
35 const float Mz_re = sh[4*256+i_offs*2+0];
36 const float Mz_im = sh[4*256+i_offs*2+1];
37
38 // (d) compute matrix-vector product
39 float Hx_re, Hx_im, Hy_re, Hy_im, Hz_re, Hz_im;
40 symmetric_tensor_multiplication(
41   Nxx_re, Nxx_im, Nxy_re, Nxy_im, Nxz_re, Nxz_im,
42   Nyy_re, Nyy_im, Nyz_re, Nyz_im, Nzz_re, Nzz_im,
43    Mx_re,  Mx_im,  My_re,  My_im,  Mz_re,  Mz_im,
44   &Hx_re, &Hx_im, &Hy_re, &Hy_im, &Hz_re, &Hz_im
45 );
46
47 // (e) write back result
48 sh[0*256+i_offs*2+0] = Hx_re;
49 sh[0*256+i_offs*2+1] = Hx_im;
50 sh[2*256+i_offs*2+0] = Hy_re;
51 sh[2*256+i_offs*2+1] = Hy_im;
52 sh[4*256+i_offs*2+0] = Hz_re;
53 sh[4*256+i_offs*2+1] = Hz_im;
54
55 // (f) copy shared memory to Mx, My, Mz
56 __syncthreads();
57 Mx[0*256+j_base+i_offs] = sh[0*256+i_offs];
58 Mx[1*256+j_base+i_offs] = sh[1*256+i_offs];
```

```
59   My[0*256+j_base+i_offs] = sh[2*256+i_offs];
60   My[1*256+j_base+i_offs] = sh[3*256+i_offs];
61   Mz[0*256+j_base+i_offs] = sh[4*256+i_offs];
62   Mz[1*256+j_base+i_offs] = sh[5*256+i_offs];
63 }
```

Listing 10: Element wise array-vector product in CUDA C.

The CUDA function is divided into six steps (a)—(f). Each thread is assigned exactly one matrix-vector multiplication. The threads are organized into thread blocks of size 256. The shared memory is used to avoid non-coalesced global memory accesses. Parts (a)—(f) are, for each thread in a thread block of 256 threads,

(a) Load demagnetization tensor into local variables. This access is coalesced, see lines 14—19.

(b) From the arrays Mx, Mz, and Mz, load the data chunks that are processed by the whole thread block into the shared memory. As each thread processes a 3-vector of complex values, each thread has to load six float numbers, see lines 22—27. This access to global memory is coalesced.

(c) After the threads are synchronized, each thread loads its assigned 3-vector from shared memory into local variables. This access in lines 30—36 is a fast operation.

(d) The matrix-vector product is computed, and the result is stored in local variables, see lines 39—45.

(e) The computed product is stored back into the shared memory, see lines 48—52.

(f) This step is the inverse of step (b). After synchronization, the products computed by the thread block are written back into global memory. This is done with coalesced memory accesses. Each thread has to write back six float numbers, see lines 56—62.

As each thread block computes 256 products, this CUDA function is suitable to process array lengths that are a multiple of 256. In order to support arbitrary array lengths, an additional CUDA function is provided that processes the remaining 0—255 elements without the use of a shared memory. This function is not shown here.

The Oersted field computation requires an antisymmetric tensor instead of a symmetric tensor, see Eq. 2.42. To compute the frequency product for the Oersted field, only step (d) has to be modified. In total, the tensor contains only three unique entries instead of six. Similarly, the demagnetization field computation using the scalar potential method, requires a vector-vector product, see Eq. 3.24. Here step (d) is modified to compute the scalar product of two vectors.

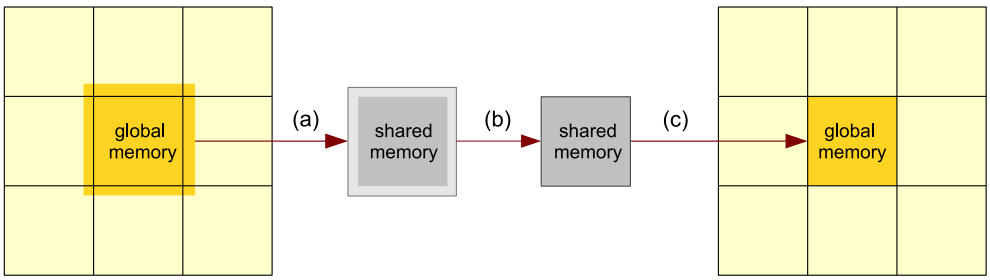


Figure 5.5: Computation of the finite difference nearest-neighbor Laplace operator on the GPU. (a) Initial copy from the global memory to the shared memory, including ghost cells, (b) in-place computation of the operator inside the shared memory, (c) copy of the result, excluding ghost cells, back to the global memory.

5.2.6 Small kernel convolution

In the finite difference approximation the exchange field is represented by the application of a six-neighbor for 3-D grids and a four-neighbor in 2-D grids discrete Laplacian operator to the components of the magnetization, see Eq. 2.6. This computation can be expressed as the linear convolution of M_x, M_y, and M_z with a 3×3 (2-D grids) or a 3×3×3 (3-D grids) convolution kernel, producing the exchange field H_x, H_y, and H_z (see section 3.2). The convolution with such a small, fixed-size kernel that encompasses the cells immediate neighbors is implemented in the following CUDA function `kernel_exchange_2d` in listing 11, which computed the exchange field for two-dimensional grids.

As the magnetization field is stored linearly in memory using column-major order, cell neighbors in the y-direction are stored in non-adjacent memory addresses. This is problematic for CUDA GPUs because these cells have to be accesses for the computation of the convolution sum, as the resulting memory accesses are not coalesced.

```
1  #define BSIZE 16 // block size is 16x16
2
3  template <bool periodic_x, bool periodic_y>
4  __global__ void kernel_exchange_2d(
5    const float *Mx, const float *My, const float *Mz, // magnetization
6    float *Hx, float *Hy, float *Hz, // output: exchange field
7    const float *Ms, const float *A, // material parameters
8    int dim_x, int dim_y, float wx, float wy)
9  {
10   // thread index (tx, ty) in thread block
11   const int tx = threadIdx.x, ty = threadIdx.y;
12   // simulation cell index (sx, sy) for this thread
13   const int sx = blockIdx.x * BSIZE + tx, sy = blockIdx.y * BSIZE + ty;
14   // shared memory to store mx, my, mz, Ms
15   __shared__ float sh[4][2+BSIZE][2+BSIZE];
16
17   if (sx < dim_x && sy < dim_y) {
18     ///// (a) Copy tile to shared memory, including ghost cells /////
19     // copy tile
```

```
20      float mx_i, my_i, mz_i;
21      const float Ms_i = sh[3][ty+1][tx+1] = Ms[i];
22      if (Ms_i != 0.0) {
23        sh[0][ty+1][tx+1] = mx_i = Mx[i] / Ms_i;
24        sh[1][ty+1][tx+1] = my_i = My[i] / Ms_i;
25        sh[2][ty+1][tx+1] = mz_i = Mz[i] / Ms_i;
26      }
27
28      // copy ghost cells (omitted here for brevity)
29      copy_ghost_cells<periodic_x, periodic_y>(
30        Mx, My, Mz, Ms, sh, tx, ty, sx, sy, dim_x, dim_y
31      );
32      __syncthreads();
33
34      if (Ms_i > 0) {
35        // (b) Compute the finite differences
36        float sum[3] = {0,0,0};
37        if (sh[3][ty+1][tx] != 0) {
38          sum[0] += (sh[0][ty+1][tx]-mx_i)*wx;
39          sum[1] += (sh[1][ty+1][tx]-my_i)*wx;
40          sum[2] += (sh[2][ty+1][tx]-mz_i)*wx;
41        }
42        if (sh[3][TY][tx] != 0) {
43          sum[0] += (sh[0][ty+1][tx+2]-mx_i)*wx;
44          sum[1] += (sh[1][ty+1][tx+2]-my_i)*wx;
45          sum[2] += (sh[2][ty+1][tx+2]-mz_i)*wx;
46        }
47        if (sh[3][TY-1][tx+1] != 0) {
48          sum[0] += (sh[0][ty][tx+1]-mx_i)*wy;
49          sum[1] += (sh[1][ty][tx+1]-my_i)*wy;
50          sum[2] += (sh[2][ty][tx+1]-mz_i)*wy;
51        }
52        if (sh[3][TY+1][tx+1] != 0) {
53          sum[0] += (sh[0][ty+2][tx+1]-mx_i)*wy;
54          sum[1] += (sh[1][ty+2][tx+1]-my_i)*wy;
55          sum[2] += (sh[2][ty+2][tx+1]-mz_i)*wy;
56        }
57
58        // (c) Compute exchange field and write it back to global memory.
59        const float f = A[i] / Ms_i;
60        Hx[i] = sum[0]*f; Hy[i] = sum[1]*f; Hz[i] = sum[2]*f;
61      } else {
62        Hx[i] = Hy[i] = Hz[i] = 0.0;
63      }
64    } else {
65      __syncthreads();
66    }
67  }
```

Listing 11: Exchange field computation for 2-dimensional grids using a finite difference Laplace operator in CUDA C.

The computation time can be improved by dividing the magnetization array into a grid of rectangular blocks of 16×16 simulation cells. Each block is then assigned a thread block using $16 \cdot 16 = 256$ threads. Before the Laplacian of one block is computed, the block (lines 19—26) and its adjacent cells from the adjacent blocks (lines 29—31), the so-called *ghost cells*, are copied into the shared memory of the thread block, also see Fig. 5.5. With the exception of some ghost-cell reads, this includes only coalesced memory accesses. Then, each thread computes the Laplacian of its simulation cell using fast reads from shared-memory (lines 36—56). Finally the exchange field is computed from the convolution sums and copied into the global memory by only coalesced memory transfers (lines 59—60). The exchange field computation on three-dimensional grids of simulation cells is implemented similarly. Here thread blocks with $8 \times 8 \times 8 = 512$ threads is mapped to three-dimensional tiles of $8 \times 8 \times 8$ simulation cells.

Part III
The MicroMagnum Simulator

The design and implementation of the MicroMagnum simulator is introduced. The software is tested using unit tests, system tests and performance tests. Finally a use-case of using MicroMagnum for a physical simulation problem is presented.

6 Design and Implementation

In this section the MicroMagnum simulator from the development perspective as well as the end users' perspective is described.

Both the developers and the end users interact with the software for a considerable amount of time. Thus it is an advantage if the software meets a range of quality requirements. Experience has shown that from the end users point of view, primarily the ease of use and the execution speed of the simulator is most important. This includes a complete documentation of the software. The end user is mostly interested in the results he or she gets out of the simulator, regardless of how these results were achieved. Thus it is not important, for example, how well his or her simulation script was written as long as it works. Often the simulation script is copy-pasted from available examples and verified by trial and error. This way of programming is referred to as opportunistic programming[94]. The user programming interface of MicroMagnum was developed with this type of usage in mind. Useful simulations can be already programmed in just a few lines of code. However, advanced users often need more functionality than that which can be provided by a simple interface. Hence it is important that the simulator remains extendable. Although the user is interested in how fast he or she gets his results from the simulator, it does not matter to the user how the speed is actually achieved. Thus, whether the simulator runs on graphics processors or on CPUs should be transparent to the end user, i.e. the simulation scripts he or she writes should require no modification in order to run on different hardware. This is achieved by a hardware abstraction layer.

6.1 Software quality requirements

Software quality is important. An overview of models that describe software quality are given in Ref. [95]. The first systematic approaches to formally define software quality are from the 1970s. Boehm et al.[96] list several desired properties of a software project. These are performance, correctness, extendability, usability, portability, and maintainability. The *performance* is a measure for the time that the program needs to complete. A program is *correct* if it always produces the right results according to its specification. The *portability* requirement is satisfied if the program can be executed on the desired

computer system without non-trivial modifications. *Maintainability* means that faults in the software can be detected and corrected with relative ease. A program which is *extendable* is developed with future extensions in mind. Finally, the *usability* of a program measures how easy it can be operated by the user.

Each of these requirements can be applied to any software. Depending on the type of software, the importance of each requirement will vary. In this work, it will be shown that all mentioned requirements are important for the micromagnetic simulator presented here. Typically physicists and engineers use MicroMagnum to perform physical simulations. A high performance reduces the waiting time until their simulations are finished. Simulated results must be physically correct. A correct implementation of the micromagnetic model is thus a prerequisite. Different users have different computer hardware and operating systems. MicroMagnum should be able to run without problems caused by different computer systems. In this case, portability also means that the code must run on both CPU and GPU. As the complexity of MicroMagnum grows during its development, it is important that it stays maintainable. A modular software architecture helps to achieve this goal. Users are expected to contribute new physical model extensions. Thus the software must be easily extendable. Since the users cannot be expected to be experts in computer programming, an intuitive user interface for specifying the simulations must be provided in order to achieve usability.

6.2 Programming language choice

The choice of the programming languages for a software project is a major design decision that needs to be resolved before any code is written. In [25] three different prototypes of micromagnetic simulators are developed using Java[97], Python[36], and MATLAB[98] as the implementation language, respectively. All prototypes use a rectangular finite-difference discretization and employ fast Fourier transforms to compute the demagnetization field. It was evaluated how the choice of programming language affects the software quality in terms of availability of high level language concepts, portability, testability, and performance. With the exception of MATLAB, these languages are in contrast to the traditional languages like C and FORTRAN that can be found in the scientific computing landscape. Support of *modern programming language concepts* was identified as a weakness of MATLAB, especially for object-oriented programming (OOP) features. Both Java and Python support OOP concepts natively. In addition, Python supports functional programming by its use of function objects which implement closures. Next, *portability* was evaluated. All three languages are executed within a virtual machine. Thus, in theory, the simulators can be executed on any computer that has the correct virtual machine installed. In practice, the use of external numerical computing libraries limits this freedom, as these libraries are often written in a compiled system language like C or C++ for performance reasons. Thus they are often only available on selected

platforms. However, it was concluded that there are enough readily available numerical libraries on each of the three evaluated programming languages to implement a portable micromagnetic simulator. Regarding *testability*, the Java programming language scored best. For Java, there are many tools that support a test-driven development process. In particular, there exist extensive tools for unit-testing and determining test coverage. Tools for Python and MATLAB are not as advanced. Last, the *performance* of the simulators written in Java, Python and MATLAB was compared. The performance largely depended on the speed of the fast Fourier transform implementations. The Java program uses the pure-Java *jFFT* library and the Python program the *NumPy*[99] and *SciPy*[100] libraries. The MATLAB program uses the FFT functions by the MATLAB runtime. In current MATLAB runtimes, these are implemented in C using the FFTW[70] library. Here, the performance of Java and the MATLAB versions turned out to be competitive to the OOMMF simulator. The Python counterpart was about 50 % slower. These were attributed to the fact that not all optimizations could be implemented using the SciPy and NumPy libraries. In order to implement all optimizations, an extension library written in, e.g., C would be necessary. In conclusion, the author decides against MATLAB and recommends to use either Java or Python for the development of a micromagnetic simulator. As major disadvantages of MATLAB the limited high-level programming concepts and the poor unit test and test coverage support was identified. If a language like Python is chosen, he recommends to develop a working software first, and later, if necessary, reimplement the performance-critical parts in a lower-level language in the form of an extension library. This is largely the route that was taken during the development of MicroMagnum.

Thus MicroMagnum is written in Python[36], and the C++ language[34] is used to implement the speed-critical software parts. These parts include all numeric algorithms that implement the micromagetic model. GPU-specific code is written in CUDA C. All C++ and CUDA code is compiled into an extension library that is loaded by the Python interpreter. The architecture of MicroMagnum is shown in Fig. 6.1. This combination is a common pattern to combine the advantages of a scripting language and a system programming language for scientific computing[101, 102]. Python is a strongly and dynamically typed programming language. This means that although values in Python have a (strong) type, the variables that contain these values do not. Programming languages that have this property are informally referred to as “duck typing” languages. They are usually interpreted languages. Python also provides automatic garbage collection. No time consuming compile steps are required during the development. For interactive development and testing, the *IPython*[103] in combination with the PyLab software provides similar features to the MATLAB console and to the graphical notebooks in Mathematica. C++ is, like Java a statically typed language that allows object-oriented programming. It is used to implement the performance-critical parts of the simulator. The main advantage of Java is its automated garbage collection, which avoids most sources of memory

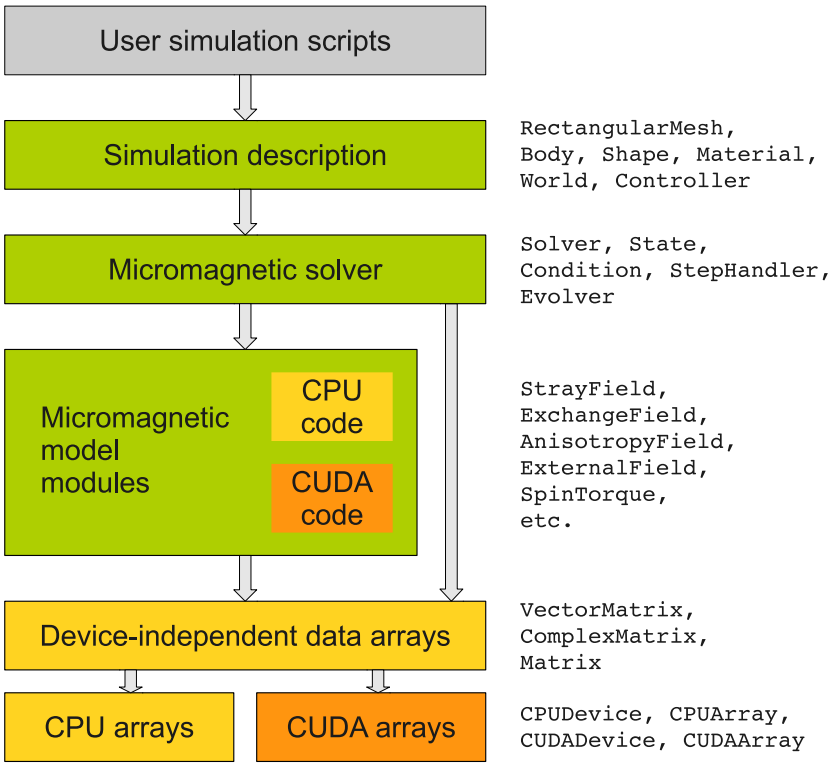


Figure 6.1: Architectural software layers of MicroMagnum annotated by their implementing classes. The colors denote the implementation languages of the layers: (grey/green), C++ (yellow), CUDA C (orange). The gray arrows show the dependencies between the layers.

leaks. C++ programmers have to resort to manual memory management techniques like the RAII[104] principle. However MicroMagnum allows the garbage collector of the Python virtual machine to collect C++ objects that are unreachable from Python code. Finally, CUDA C is used to implement numerical routines on the GPU.

6.3 Architecture

The simulator is divided into a hierarchy of software layers, see Fig. 6.1. The lowest layer contains the mathematical subroutines. This layer functions as a mathematical abstraction layer that provides implementations for the CPU and CUDA devices (GPUs). It is written in C++ and CUDA C. Supported mathematical operations include, for example, array operations, fast convolution, and fast Fourier transforms. Most hardware optimizations apply in this layer. Now that the hardware is abstracted away, the mathematical routines are used as building blocks to implement the algorithms for the discretized physical model. For example, this layer provides a functional-style interface to compute the exchange field from an array of magnetization vectors. The routines in this layer are written in C++, however, they can be called by the Python code in the higher layers. The glue code that exports the C++ function to the Python virtual machine is automatically generated by the SWIG wrapper generator software[105]. Thus from the programmer's perspective, the C++ functions and classes can be directly called as if they were implemented in Python. The module system layer is the first layer that is written in Python. It provides an object-based interface for the functional-style routines of the physical model layer. Each module represents a part of the physical model. A set of modules is then combined to form the complete simulation model. For the micromagnetic model, this module system represents an ordinary differential equations. This equation is solved by the numerical solver layer. It uses a Runge-Kutta method to advance the solution from the initial conditions to the final result. The simulation description layer provides an easy-to-use application interface that is used to parametrize and specify simulation setups by the end user. It also provides methods to automate logging and output during the simulation. Finally, at the top-most layer, there are the user simulation scripts. These scripts are written by the end users of MicroMagnum. The scripts are written in Python and make calls to the simulation description interface. Typical simulation scripts are between ten lines and hundreds of lines long.

6.3.1 Mathematical abstraction layer

The mathematical abstraction layer provides device independent data structures that store numerical data and perform numerical operations. Here the data can be stored either in system memory for access by the CPU or on the global memory of the graphics processor for use by the GPU. The data is synchronized between the CPU and GPU as needed. The most important supported data structures are multi-dimensional floating

point number arrays that store either values of scalars, 3-vectors and complex numbers, see Tab. 6.1. These three array types can represent all physical fields for the discretized

Array type	Element type
`ScalarArray`	real value
`ComplexArray`	complex value
`VectorArray`	3-tuple of reals

Table 6.1: Types of hardware-independent number arrays in MicroMagnum.

micromagnetic model.

The device-independent number arrays can be synchronized with one or more *devices*. Devices are abstract representations of processors and their memory. Currently there are three devices, one CPU device that works with 64 bit floating point numbers stored in system memory, and two GPU devices that work with 32 bit and 64 bit floating point numbers stored in CUDA global memory, respectively: An array is *synchronized* with a

Device	Memory	Floating point numbers
`CPUDevice`	System memory	64 bit
`CU32Device`	CUDA global memory	32 bit
`CU64Device`	CUDA global memory	64 bit

Table 6.2: Types of hardware devices in MicroMagnum.

device if its content is actually stored in the memory of that device. It is thus required for an initialized array that it is synchronized with at least one device at any time. This means that if the array contents are modified on one device, all other device must be tagged as unsynchronized. On the other side, before an unsynchronized device can access an array, it must first synchronize the array for the device by copying the data. All needed synchronization is handled by the acquisition of locks on arrays for a specific device.

In order to access elements of an array on a device, a data lock on the array for the device has to be acquired. For example, in order to access the data in a CUDA kernel, a data lock for the GPU device has to be obtained. While a lock is in place, it is guaranteed that the device is synchronized with the array. Thus, if necessary, the lock operation copies the array entries from another synchronized device to the locking device. There are two types of locks, a read-lock and a read/write-lock. A read/write lock can only be acquired on an array if there are no other locks present. In contrast, multiple read-locks are allowed on an array given that there are no read/write-lock in place. A read/write lock on one device invalidates the data storage on all other devices, i.e. while a read/write lock for a device is present, the array is synchronized with exactly that device. If multiple read-locks are present, the array may the synchronized with more than one device.

All classes in the mathematical abstraction layer are implemented in C++. The classes that store the multidimensional arrays are called `Matrix`, `VectorMatrix`, and `ComplexMatrix`. Although it is possible to obtain and release the locks by calling the respective methods on the matrix objects, it is recommended to lock a matrix by the instantiation of so-called accessor objects on the matrix. During the construction of the accessor object a lock is acquired, and the lock is released during destruction. Accessor objects are meant to be allocated on automatic memory. During the lifetime of the accessor the matrix elements can be accessed. This design adheres to the *Resource Allocation Is Initialization* (RAII, [106, 104]) principle, which is a popular technique in C++ programming. The automatic destruction of the accessor assures that the matrix is unlocked when the accessor goes out of scope. This is the case for normal code execution as well as when an exception is thrown. The following example shows how the elements of a matrix of scalar floating point numbers are accessed on the CPU:

```
1  void f(Matrix &M)
2  {
3    Matrix::const_accessor M_acc(M);
4    M_acc.at(0, 0) = 42;
5
6    if (<error>) throw runtime_error("error occurred")
7  }
```

Here, the type `Matrix::const_accessor` acquires a read-only lock on the matrix `M`. Even when an exception is thrown the matrix is unlocked once the accessor variable `M_acc` goes out of scope. This makes it easy to write code that provides *basic* exception safety[106], which guarantees that object invariants are preserved and no resources are leaked. A *strong* exception guarantee[106] could be supported when the accessor object worked in a transactional way. A straightforward implementation of transactional accessors would require to create a temporary copy of the matrix data, provide element access to the copy, and commit it to the matrix storage during the unlock operation (or throw it away when an exception is thrown). It was chosen against supporting transactional accessors because the creation of the temporary copy would consume additional computing time and resources. If strong exception safety is needed, algorithms can be expressed like this:

```
1  void f(Matrix &M)
2  {
3    Matrix M_copy = M; // may throw
4    {
5      Matrix::const_accessor M_acc(M_copy)
6      // ...
7    }
8    M.swap(M_copy); // can not throw
9  }
```

Here the swap-operation exchanges the contents of the matrices `M` and `M_copy`. It is possible to implement this method with a *nothrow* guarantee[106] because no resources have to be allocated (as `M` and `M_copy` are already initialized), and only the object variables have to be exchanged. In conclusion the RAII principle allows automatic memory management without a garbage collector.

In addition to access to array elements, common numerical operations on arrays are available. These include the scaling of an array, addition of two arrays, normalization, randomization, finding the minimum and maximum, and finding the sum of the elements. The operations are implemented for each device in the respective device classes. Operations that involve two arrays, e.g., the addition of two arrays, thus require that both arrays are synchronized on the same device. If this is not the case, one array is synchronized before the operation is executed. Here the choice of the synchronization device is biased towards GPU devices in order to maximize performance. Using the SWIG interface generator, the C++ array classes are exposed to Python. On the Python side, the user has access to all array operations while the device synchronization is abstracted away. Access to array elements is implemented via the subscription operator which automatically locks the array on the CPU to access the elements.

The hardware-specific parts of MicroMagnum are implemented in classes for the *device arrays* and the devices. There is a class derived from `Array` for each device, i.e. `CPUArray`, `CU32Array`, and `CU64Array`. It is responsible for storing a (multi-dimensional) scalar number array in a hardware-dependent way. A `CPUArray` allocates storage space in system memory, and `CU32Array`/`CU64Array` in CUDA global memory, respectively. The elementary operations on arrays are implemented in device-dependent subclasses of `Device`, i.e. `CPUDevice`, `CU32Device`, and `CU64Device`. Each `Device` subclass performs operations on arrays of its own device type only. As an exception, each `Device` class is required to be able to transfer the contents of its arrays to and from CPU arrays (of type `CPUArray`) to allow for array synchronization between devices.

The device-*independent* array classes are derived from `AbstractMatrix`. An `AbstractMatrix` manages the device-independent storage of the matrix contents. At this level, the locking operations take place. Derived from `AbstractMatrix` are the classes `ScalarMatrix`, `VectorMatrix`, and `ComplexMatrix`, which represent multi-dimensional matrices of scalar values, 3-vectors, and complex values, respectively. More types of such arrays can be added if the need arises. Each matrix type provides object methods to perform elementary operations. They are implemented via calls to the device classes with prior array locks.

6.3.2 Micromagnetic module system

In MicroMagnum the underlying micromagnetic model that drives the simulation is configured by the selection of a set of *modules*. One such module represents one or more

model variables and their mathematical relationships to other model variables. For example, the exchange field $\vec{H}_{\text{exch}}$ computation includes the magnetization $\vec{M}$ and the material parameters M_s and A (see Eq. 2.6). The exchange field is represented by the `ExchangeField` module and calculates the `H_exch` model variable with the dependant variable `M` and the parameters `Ms` and `A`. Figure 6.2 shows another example of a module configuration and its connections between model variables. Table 6.3 lists all available modules in MicroMagnum.

Each module has a set of associated variables, which fall into three categories. *Output variables* are those variables that the module can calculate. Most modules have only one or two output variables. *Input variables* are those variables which need to be known before any output variables can be calculated. They are the dependant variables of the input variables. The *parameters* are constant variables that are set by the user as part of the simulation setup. In the micromagnetic model, they are used to set, e.g., material parameters and configuration parameters. The simulator will make sure that the selected modules form a complete model that can be treated as an ordinary differential equation. The restrictions are:

- Each output variable must exist exactly once.
- For each input variable, there needs to be an output variable of the same name.
- Cycles between dependant variables are prohibited.

The currently implemented modules are listed in the table 6.3.

In the Python script, the user selects the modules that he or she wants to include in the simulation model by a call to the `create_solver` function, for example:

```
solver = create_solver(
  world, [ExchangeField, StrayField, ExternalField]
)
```

In this example, the `LandauLifshitzGilbert`, `ExchangeField`, `StrayField`, and `ExternalField` modules were included (the `LandauLifshitzGilbert` module is always included by default). The model variables that are provided by the modules can now be accessed by the `solver.state` object. This object represents the state of the simulated sample at a specific simulation time. Continuing the example, the user may now set the magnetization pattern and retrieve its corresponding exchange field:

```
solver.state.M = (8e5, 0, 0)  # e. g., set constant magnetization..
H_exch = solver.state.H_exch  # ..and calculate its exchange field.
```

Here internally, the `ExchangeField` module is called to compute the exchange field from the magnetization. Similarly, all module variables can be computed by accessing the `solver.state` object.

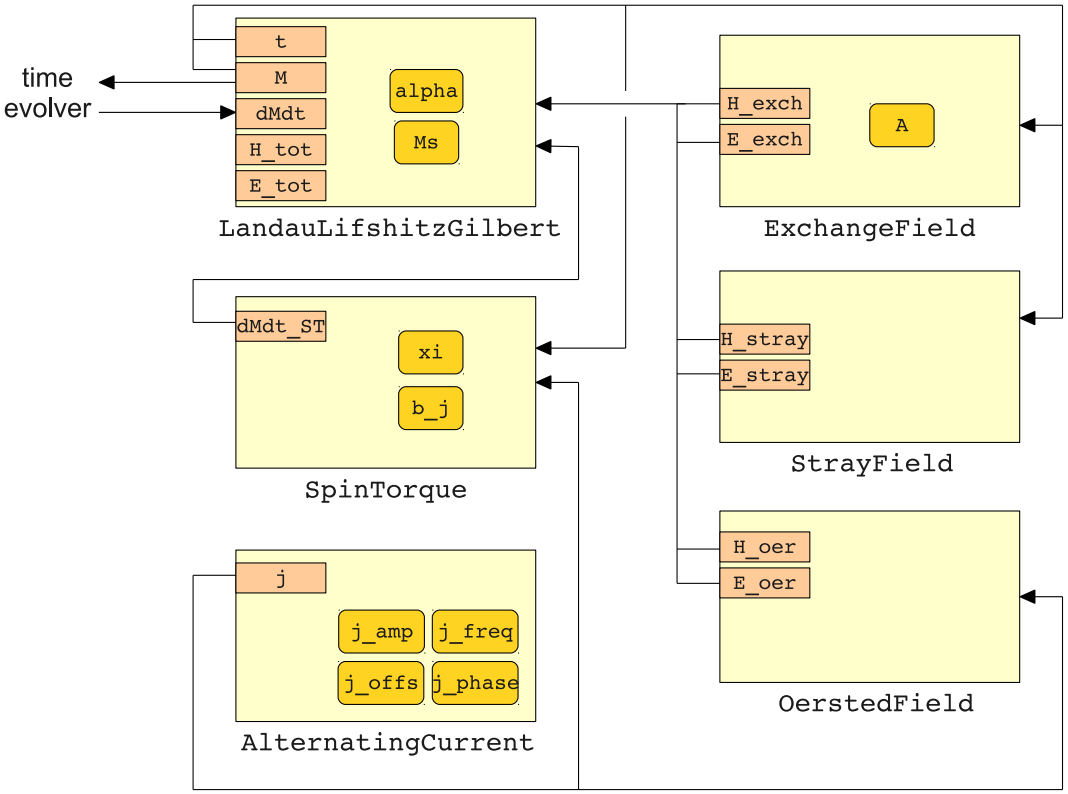


Figure 6.2: Exemplary module system that is generated by the expression `create_solver(mesh, [ExchangeField, StrayField, OerstedField, SpinTorque, AlternatingField])`. The large rectangles denote the selected modules. The arrows connect output variables to input variables between modules. Inside the modules, the rectangles denote their output variables and the rounded rectangles denote module parameters. The time evolver interacts with the module system via the variables `M`, `t` and `dMdt`.

.

Module	Input var.	Output var.	Parameter variables
`LandauLifshitzGilbert`	$\vec{M}$	$(\mathrm{d}\vec{M}/\mathrm{d}t)$, $\vec{H}_{\mathrm{eff}}$	α, M_{s}
`ExchangeField`	$\vec{M}$	$\vec{H}_{\mathrm{exch}}$, E_{exch}	A
`StrayField`	$\vec{M}$	$\vec{H}_{\mathrm{stray}}$, E_{stray}	-
`AnisotropyField`	$\vec{M}$	$\vec{H}_{\mathrm{aniso}}$, E_{aniso}	k_{uni}, k_{cub}, $\vec{a}_1$, $\vec{a}_2$
`ExternalField`	$\vec{M}$	$\vec{H}_{\mathrm{ext}}$, E_{ext}	(amplitude, frequency, etc.)
`OerstedField`	$\vec{j}$	$\vec{H}_{\mathrm{oer}}$, E_{oer}	-
`SpinTorque`	$\vec{M}$, $\vec{j}$	$(\mathrm{d}\vec{M}_{\mathrm{ST}}/\mathrm{d}t)$	ξ, P, b_j
`CurrentPath`	$\vec{M}$, V_{contact}	$\vec{j}$	(contact points, resistance)

Table 6.3: List of the physical modules available in MicroMagnum, including their input, output and parameter variables.

All modules are written in Python and make calls into the C++ layer for most computations. A module is a subclass of the `Module` class, which provides abstract methods to compute or update input and output variables and property variables. Additionally the module class provides information on the names of its associated model and parameter variables. This allows the module system to connect all enabled modules to form the whole simulation model, which is represented by an instance of the internal `System` class. Also the module system checks if the enabled modules are compatible. For example, it is not allowed to have two modules provide the same output variables. A simple manual caching method is set in place to avoid that model variables are computed twice within the same time integration step. Each module has access to a cache table within the `State` object, which is flushed automatically after each time step. If a time-intensive output variable is requested from a module, the module can check if there is already a cached copy of the variable available, and return it instead if possible. Otherwise, the module computes the variable and inserts it into the state cache.

Because the model is used by the Runge-Kutta evolver, it must describe an ordinary differential equation. MicroMagnum reserves three variables, which must be provided by the enabled modules, for this purpose:

- the time `t`,
- the vector field `y` representing the integration variable,
- the vector field `dydt` representing the derivitive of `y` at time `t`.

In the micromagnetic model, the variables `y` and `dydt` are aliases for the variables `M` and `dMdt` that are defined by the `LandauLifshitzGilbert` module. The Runge-Kutta evolver logic is implemented in Python and the numerical parts (e.g. Eqs. 2.52, 2.51 and 2.58) are implemented in C++/CUDA. The methods of Euler, Runge-Kutta (RK4), Runge-Kutta-Fehlberg (RKF45), Dormand-Prince (DP54), Cash-Carp (CC45), and Bogacki-Shampine (RK23) are supported. All methods except the Euler method and the RK4 method include error estimation and step size control. The default method in MicroMagnum is the RKF45 method.

6.3.3 Simulation description and solver interface

The simulation scripting interface provides a simple to use way to assemble a module system in order to specify the simulation model first, the user specifies the geometry of the simulation volume and the contained materials. This specification is represented by an instance of the class `World`. Second, the user selects the physical modules that he wants to have in his simulation model. Then, MicroMagnum initializes the module system and any material parameters automatically. After that the simulation can be started using the solver and controller objects. During the simulation logging information and

snapshots of the current simulation state can be produced on disk. In the following each step is described in detail. A a simulation script, the MicroMagnum library is imported using

```
from magnum import *
```

This statement imports all functions and classes provided by MicroMagnum into the caller's namespace.

Geometry setup The geometry setup is specified using objects of the classes `RectangularMesh`, `World`, `Body`, `Material`, and `Shape`. The `RectangularMesh` class stores the discretizing finite-difference mesh of the total simulation volume. For example, a mesh with $50 \times 50 \times 4$ cells of size $4\text{nm} \times 4\text{nm} \times 4\text{nm}$ with open boundary conditions is created using the expression

```
mesh = RectangularMesh(
  (50, 50, 4),       # number of simulation cells
  (4e-9, 4e-9, 4e-9) # simulation cell size
)
```

Volumes with periodic boundary conditions are specified by an optional third argument to `RectangularMesh`.

A `World` object defines the geometry and material parameters of the simulation. It contains information about the mesh and one or more `Body` instances. The bodies define the material parameters in subvolumes of the simulation volume. The material parameters of the body are specified by a `Material` object. There are several predefined materials available in MicroMagnum. The volume of the body is specified by a `Shape` object. For example, a simulation volume that is completely filled with Permalloy can be defined using the following script.

```
world = World(
  RectangularMesh((50, 50, 1), (4e-9, 4e-9, 20e-9)),
  Body("all", Material.Py(), Everywhere())
)
```

Here the world object contains the mesh and one body object. The body has a name, a material, and a shape. The shape `Everywhere()` describes the whole simulation volume.

There are several predefined shapes. Shapes can be combined using the set operations "union", "intersect", and "invert". This allows the user to create complicated volumes using only primitive basic shapes. This technique is known as constructive solid geometry[107]. Table 6.4 shows the available basic shapes and the combining expressions. Objects of `ImageShape` define their volumes by a color-coded areas in graphical images. They are created by an `ImageShapeCreator` object, e.g.:

```
isc = ImageShapeCreator(mesh, "image.png")
shape1 = isc.pick("red")
shape2 = isc.pick("green")
```

Here two shapes are created using the red and green pixels in the image `image.png` as a template. This allows the user to draw complex two-dimensional geometry in a image editor in an intuitive way.

Simulation model setup Now that the world is defined, the simulation model is selected via a list of modules. This is done using the `create_solver` function. For example, the following code includes the `ExchangeField`, the `StrayField` and an `ExternalField` module:

```
solver = create_solver(
  world, [ExchangeField, StrayField, ExternalField]
)
```

The function returns a solver object. Using the `.state` property, the current simulation state can be accessed in the form of a state object. Every model variable and module parameter is accessed via this state object. For example, after a solver is created, the initial magnetization can be assigned with

```
solver.state.M = (8e5, 0, 0)
```

Here a homogeneous magnetization is assigned. After that, for example, the demagnetization field resulting from the assigned magnetization can be retrieved using the model variable `H_stray`:

```
H = solver.state.H_stray
```

The variable `H` contains now a `VectorField` object with the stray field.

Solver interface After the geometry, the solver and the initial conditions are set up, the simulation is started. This is done by a call to the `solve` method. It has one parameter which describes when the simulation is complete in the form of a `Condition` instance. Conditions are predicates over a micromagnetic state object and evaluate to either *true* or *false*. In the case of the `solver` call, the most often used condition is `condition.Time`, which is true after a given simulation time is reached.

```
solver.solve(<condition>)
solver.solve(condition.Time(10e-9))
```

Another often used condition is the `condition.Relaxed(d)` condition, which evaluates to true when the magnetization change in degrees per rad becomes smaller than d, i.e. when a energy minimum is reached sufficiently close.

```
solver.solve(condition.Relaxed(5))
solver.relax(5) # equivalent to first line
```

As a shortcut, the call to `solver.relax()` is equivalent to calling `solve` with the relaxed-condition. Conditions may also be combined using Boolean operations to express more complex predicates. Alternatively, custom predicates can be easily defined. For example, the predefined `condition.Time(t)` condition could also be written as a custom condition with

```
cond = condition.Condition(lambda state: state.t >= t)
```

Another example of a custom condition can be seen in listing 17 in appendix A.1.4.

Before the simulation is started, so-called step-handlers can be attached to the solver instance. For each attached step handler, a condition must be supplied. When the simulation is being executed, the step handler is called with the current simulation state whenever the condition is satisfied. Step handlers are added to the solver with

```
solver.addStepHandler(<step-handler>, <condition>)
```

There exist several predefined step handlers, see Tab. 6.6. Their main purpose is the collection of data during the simulation. For example, the `OOMMFStorage` step handler continuously saves a snapshot of the current micromagnetic state while the simulation is running. The `DataTableLog` step handler writes a tabular data log file in the .odt file format that contains one row for each inspected simulation state.

Batch simulations Often multiple simulations with the same simulation setup, but with different parameters sets have to be simulated. The provided `Controller` classes automates the iteration through a list of parameter sets (represented as a tuple). There exist several controller classes. In Dependence on the context in which the simulation script was executed, the `create_controller` function selects an appropriate controller class and instantiates it.

```
def simulation_fn(a,b,c):
  # run one simulation with parameters a, b, and c.

c = make_controller([(1,2,3), (4,5,6), ...], simulation_fn)
c.start()
```

expression	defined volume
shape objects	
`Everywhere()`	The whole volume.
`Cuboid((x0,y0,z0),(x1,y1,z1))`	A cuboid aligned to the coordinate axes, defined by two diagonally opposite corners.
`Sphere((x,y,z),r)`	A sphere with center point and radius.
`Cylinder((x0,y0,z0),(x1,y1,z1),r)`	A cylinder with end points and a radius.
`Prism((x0,y0,z0),(x1,y1,z1),poly)`	Like cylinder, but with a polygonal base.
`ImageShape`	A region defined by a graphical image.
combinations of shape objects using set operations	
`<shape1> & <shape2>`	Intersection of the two shapes.
`<shape1> | <shape2>`	Union of the two shapes.
`~<shape>`	Inversion of the shape.
`<shape1> & ~<shape2>`	Subtraction of two shapes.

Table 6.4: Shape objects and their combinations in MicroMagnum.

expression	defined condition
condition objects	
`Always()`	Always true.
`Never()`	Always false.
`Time(t)`	True after time `t` is reached.
`TimeBetween(t0, t1)`	True inside the time interval $[t_0, t_1]$.
`Relaxed(rad_per_sec)`	True if magnetization change is below `rad_per_sec`.
`AfterNthStep(n)`	True after the n-th simulation step is reached.
`EveryNthStep(n)`	True for every n-th simulation step.
`EveryNthSecond(s)`	True at every s seconds.
combinations of condition objects	
`~<cond>`	True if <cond> returns false, false otherwise.
`<cond1> & <cond2>`	True if both conditions return true, false otherwise.
`<cond1> | <cond2>`	True if either condition returns true, false otherwise.

Table 6.5: condition object and their combinations in MicroMagnum.

expression	defined step handler
`ScreenLog()`	Displays log output on the console.
`DataTableLog(file)`	Stores a log in the .odt file format[19].
`OOMMFStorage(dir, ["v1", "v2", ...])`	Stores snapshots of the fields given by the state variables "v1", "v2", ... in the .omf file format[19].
`VTKStorage(dir, [...])`	Like `OOMMFStorage`, but uses the .vtr format of the Visualization Toolkit[108].

Table 6.6: Step handler classes in MicroMagnum.

By default, the controller sequentially loops through all parameter sets and calls the simulation function with each set (in the example, `simulation_fn`). If the script is started by the operating system, the `-p` option can be used to specify a range of parameter sets that shall be processed by the simulator. For example

```
./script.py -p 0,16
```

will run parameter sets 0 to 15 sequentially, and

```
./script.py -p 5
```

will run the fifth parameter (counting from zero) set only. In order to allow the inspection of a script by automated external tools, the number of parameter sets can be requested with the `-print-num-params` argument, and the parameter sets can be listed with the `-print-all-params` argument:

```
./script.py --print-num-params --print-all-params
```

In this case the controller does not start any simulations, causing the script to exit immediately.

Using controller objects, multiple processes can be started in order to sweep through parameter sets in a parallel fashion. The selection of the parameter set range works nicely together with a batch processing system. On clusters computers where such a software is installed, the parallel execution can be easily automated. For example, on the Oracle Grid Engine (also known as the Sun Grid Engine), multiple copies of a process can be started parallely across a computer cluster by the submission of a so-called job array task. Each process has an environment variable `SGE_TASK_ID` containing the task id which is a number from 1 to n (with n being the number of tasks in the task array). Within each process, the simulation script can then be executed via the commands

```
#/bin/bash
./script.py -p $(expr $SGE_TASK_ID - 1)
```

Since this is a common task, the *make_ controller* function can be instructed to create a controller object that reads the `SGE_TASK_ID` variable itself:

```
c = make_controller(
  [(1,2,3), (4,5,6), ...], simulation_fn,
  sun_grid_engine=True
)
```

Here the environment variable is used if available to select the correct parameter set, and the Python script can be executed directly by the batch system.

6.4 Discussion

The development and design of MicroMagnum contributes to the requirements of correctness, usability, maintainability, portability, extendability, and performance.

Correctness The correctness of the simulator, i.e., that the micromagnetic model is computed correctly, is verified by the system tests in chapter 7.

Usability MicroMagnum allows to write simulation scripts that can use all features of the Python scripting language. Each simulation can be built up very intuitively with the simulation description API of MicroMagnum. As an example, a use case of MicroMagnum is given in chapter 8.

Performance Due to the support of graphics processors, the mathematical abstraction layer helps to write fast programs. The array operations are to be used as reusable building blocks for the implementation of numerical algorithms. (A similar strategy is followed by the MATLAB and the NumPy[99] development systems. An advantage of the own implementation is the abstraction from the underlying hardware. Compared to array operations that are executed on the CPU, the GPU counterparts are significantly faster. They are inherently parallel due to the GPU hardware architecture. For the micromagnetic model implementation, an analysis of the performance for both CPU and GPU hardware is given in chapter 7.3. One main result is that the demagnetization field computation on the GPU is up to 60 times faster compared to CPU computations.

Maintainability The array classes can be directly used from Python code. During the development of MicroMagnum, changing the parts that are purely implemented in Python does not require recompilation of the software. Thus the traditional development cycle of program/recompile/test is shortened for many tasks, contributing to the maintainability. To encourage test-driven development, the unit test framework that is part of the Python distribution as the module `unittest` is utilized. Unit tests are written in Python and require no recompilation step after code changes. In order to test code written in C++ or CUDA, it must be called from Python. As most of the C++ routines are kept small and purely functional, they are suitable for unit-testing by Python code.

The system tests, which are described in chapter 8, are written in Python and thus require no recompilation.

Extendability If the developer wants to develop a new function that performs some mathematical operations on matrices, the software development process might look as follows. First, a CPU version of the algorithm as well as a unit test for the new function is created. Due to the hardware abstraction, the function can be tested even while other

parts of the software might run on the GPU. If the input to the function resides on GPU memory, it will automatically be copied to the system memory due to the use of accessor objects. When the function is working correctly, the new function is ready to be ported to GPU code. This may happen gradually if the function is composed of more than one subroutine. In this case, after each port of a subroutine the unit test is re-run to make sure the new code works correctly. At the end, all code runs on the GPU. As CUDA development is more low-level it is more error-prone. By proceeding step by step and verifying the code in between by unit-tests, errors are avoided. Regarding extendability, additional hardware can be supported by adding a respective `Device` and `Array` subclass. For example, a device utilizing OpenCL could be added non-intrusively for support of graphics processors by ATI.

Concerning the Module System Layer further physical modules can be easily implemented and connected to the existing modules. This is due to the high level of abstraction of the Module System Layer and its modular structure.

Portability The mathematical abstraction layer contributes to software portability in that it removes the need for client code to target a specific hardware for the basic operations. On the Python side there is no distinction between a array that is stored on the RAM or on the graphics processor. Consequently, the entire part of MicroMagnum that is written in Python contains no code that is dependent on the currently activated device (CPU or CUDA). For the compilation of MicroMagnum the common libraries FFTS, SWIG, and the CUDA API are used which are free of cost and can be easily installed.

The GPU support may also be disabled at compile time, so that the whole simulator runs on CPU. In this case, the mathematical abstraction layer is standard conforming C++ code which can be easily ported to every system that has a C++ compiler.

7 Software tests

In this chapter the following types of software tests[109] are performed:

- Unit tests
- Integration tests
- Functionality tests
- Performance tests

Each *unit test* checks the correctness of an individual piece of code. *Integration tests* check whether the software modules are working together correctly. *Functionality tests* and *performance tests* are *system tests*. In contrast to unit and integration tests, system tests do not test specific pieces of software code. Functionality tests verify that the software as a whole conforms to its functional specification, i.e. here that the micromagnetic model is implemented correctly. The last subsection covers the performance tests.

7.1 Unit and integration tests

MicroMagnum contains about 60 automated test cases and integration tests. For unit tests the tested pieces of code are either Python classes for the module system, C++ classes for the mathematical abstraction layer, or global C++/CUDA C functions for specific numerical subroutines. The integration tests mostly cover the simulation description interface written in Python.

7.2 Functionality tests

Functionality tests are a kind of system tests. The μMAG group[110] defines several functionality tests, called *standard problems*, for micromagnetic simulators. These tests are used to validate the numerical implementation of the micromagnetic model. The tests are performed by comparing the results of different micromagnetic simulation software. Currently there are four defined standard problems, which treat different aspects of the micromagnetic model:

SP1 Compute a hysteresis curve of a ferromagnetic particle.

SP2 Compute a hysteresis curve of a ferromagnetic particle.

SP3 Compute the static magnetic configurations of a ferromagnetic cube.

SP4 Compute the dynamics of a ferromagnetic strip under the influence of an external field.

These tests include the Landau-Lifshitz-Gilbert equation including the demagnetization field, the exchange field, the anisotropy field, and the external field and are described in sections 7.2.1–7.2.4. Another functionality test that includes the spin-torque extension to the LLG equation was developed by Najafi et al.[111], see section 7.2.5. Another test specified in Ref. [111], the Larmor precession test, checks the implementation of the LLG equation and its numerical integration, see section 7.2.6. There are several other proposed standard problems (e.g., Ref. [112]). They are not covered in this work.

7.2.1 μMAG standard problem 1

In the standard problem 1 two hysteresis loops of a 1 $\mu m \times 2$ $\mu m \times 20$ nm magnetic rectangle of Permalloy are calculated. Two hysteris curves are calculated by the application of an external field, ranging from -50 mT to $+50$ mT, in the x-direction and in the y-direction. The simulation includes the Landau-Lifshitz-Gilbert equation with the demagnetization field, the exchange field, and the uniaxial anisotropy field. The specified material parameters of Permalloy are the magnetization saturation $M_\mathrm{s} = 8 \cdot 10^5$ A/m, the exchange stiffness constant $A = 1.3 \cdot 10^{-11}$ J/m, and the uniaxial anisotropy constant $k = 5 \cdot 10^2$ J/m^3. The easy axis points into the y-direction. The standard problem 1 computes

- the hysteresis curves for the x- and y-direction fields,
- snapshots of the magnetization pattern at remanence, i.e. when the applied field passes zero.

Figure 7.1 shows the hysteresis loops computed by MicroMagnum using a grid of 100×200 simulation cells of size 10 nm $\times$ 10 nm $\times$ 20 nm. The simulation script is shown in listing 14 in the appendix A.1.1. The computed hysteresis curves agree with the solutions reported at the μMAG website[110].

7.2.2 μMAG standard problem 2

In the standard problem 2 hysteresis loops of a ferromagnetic strip of different scales are calculated. The simulation includes the Landau-Lifshitz-Gilbert equation with the demagnetization field and the exchange field, but excludes the anisotropy field. The strips' dimensions (L, d, t) along the x-, y-, and z-axes have a ratio of $5 : 1 : 0.1$. To make the simulation independent of the material parameters M_s and A, the sizes are given in units of the exchange length $l_\mathrm{ex} = \sqrt{A/K_\mathrm{m}}$. Here the magnetostatic energy density $K_\mathrm{m} = \frac{1}{2}\mu_0 M_\mathrm{s}^2$. The hystereses are calculated for strips with the ratio d/l_ex ranging from 1 to 40. The magnetic field is applied in the $(1, 1, 1)$-direction. The standard problem 2 computes for each hysteresis curve

- the magnetic remanence M_rem at $|\vec{H}| = 0$,

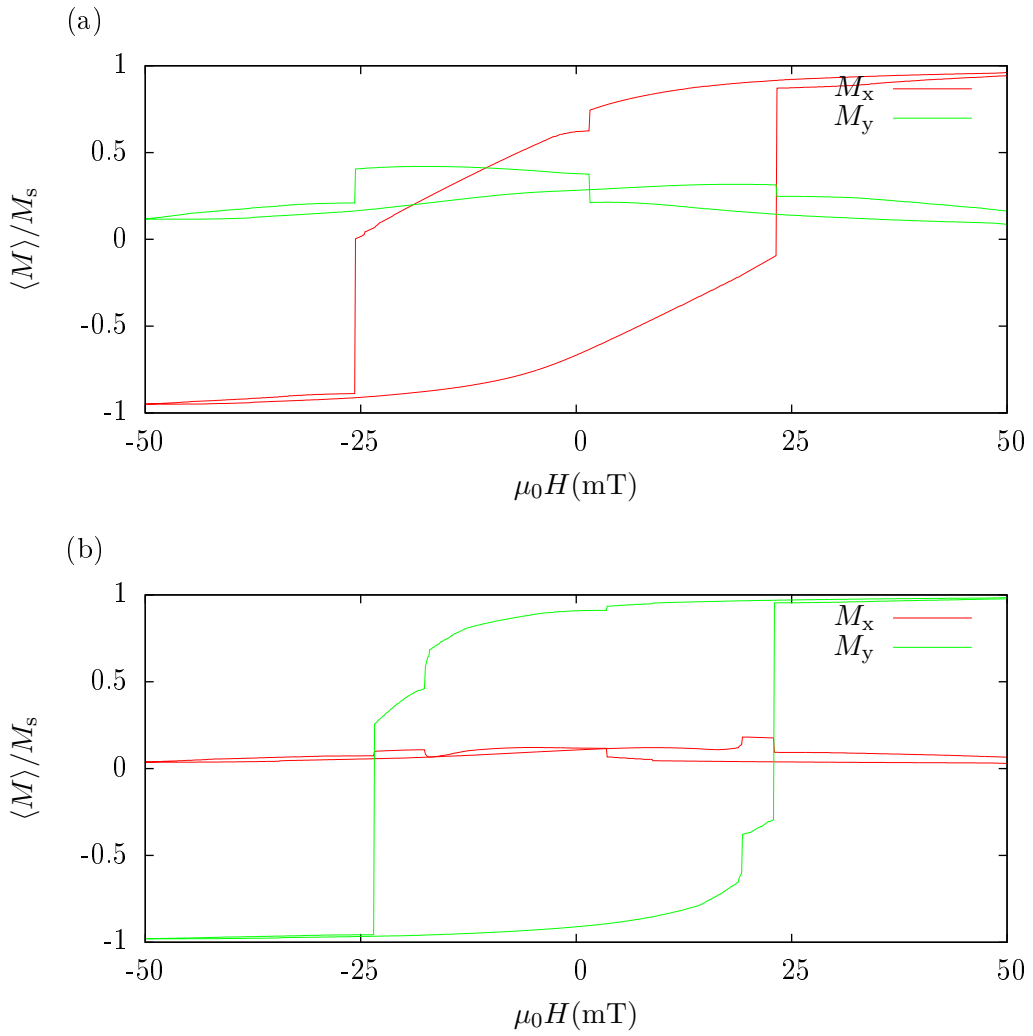


Figure 7.1: Hysteresis curve for the applied field (a) in x-direction and (b) in y-direction as computed by MicroMagnum.

- the coercivity field intensity H_c at $\vec{M} \cdot \vec{H} = 0$ (given in units of M_s).

Figure 7.2 shows the (a)–(b) magnetic remanence and (c) coercivity in dependence of the ratio d/l_{ex} as computed by MicroMagnum. The simulation script is shown in listing 15 in the appendix A.1.2. The results agree with the results reported at the μMAG website[110].

7.2.3 μMAG standard problem 3

The standard problem 3 is used to determine the single domain limit of a ferromagnetic cube. Here the single domain limit is defined as the cube side length L, given in units of the exchange length $l_{ex} = \sqrt{A/K_m}$, with equal energies of a flower state and a vortex state. The magnetostatic energy density $K_m = \frac{1}{2}\mu_0 M_s^2$. The exchange energy, the demagnetization energy and the uniaxial anisotropy energy (with $k = 0.1K_m$, and the easy

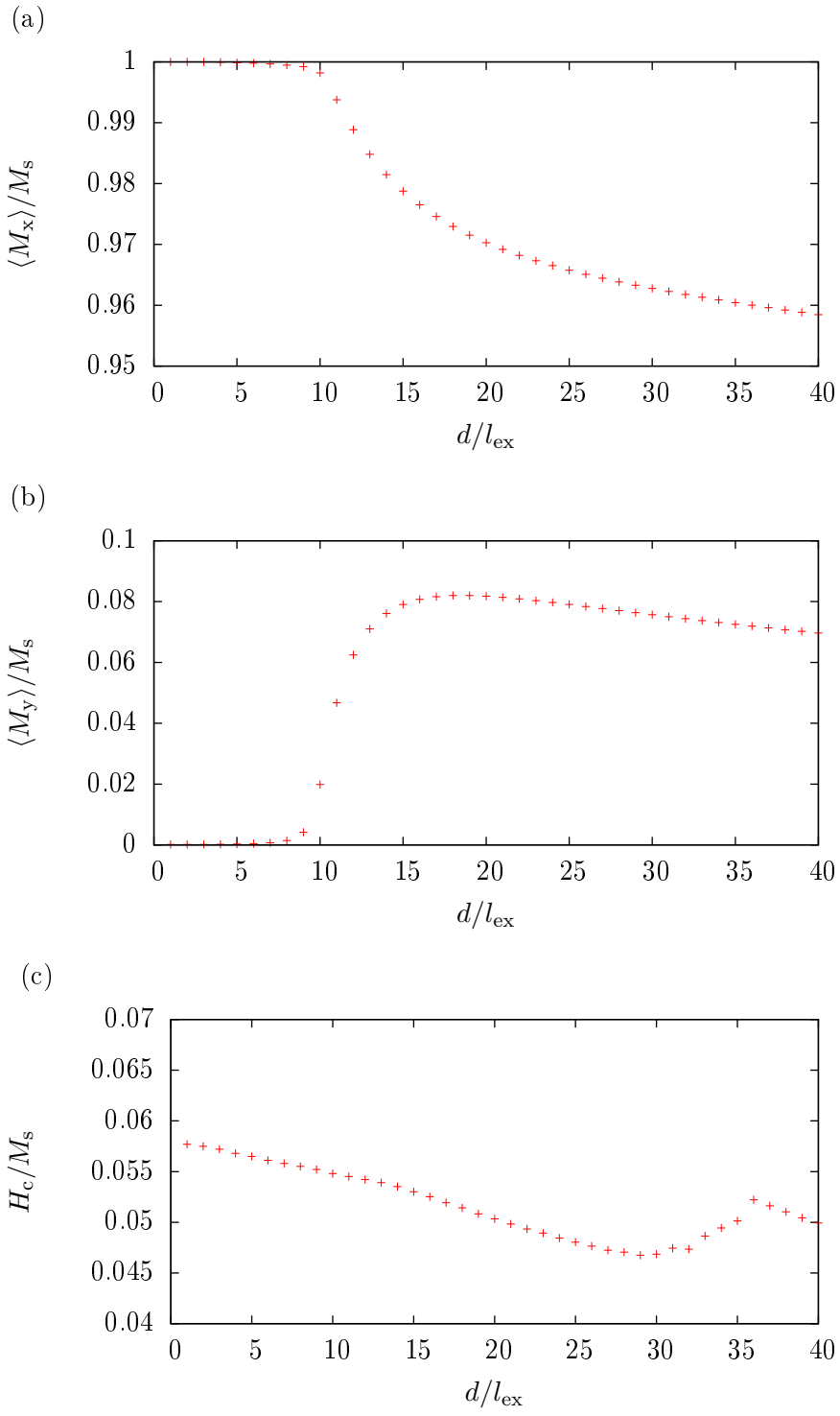


Figure 7.2: μMAG standard problem 2 results: The magnetic remanence in (a) x-direction and (b) y-direction and the (c) coercivity field intensity in dependence of the ratio d/l_{ex}.

axis along the y-direction) is included in the simulation. The simulation is independent of the chosen saturation magnetization M_s and the exchange constant A. A detailed description of the standard problem 3 and its solution by simulation is given in Ref. [113]. The standard problem 3 computes

- the single domain limit L (given in units of the exchange length l_ex)
- the total energy density and the demagnetization, exchange, and anisotropy energy densities for the vortex state and the flower state at the single domain limit (given in units of K_m)
- the average unit magnetization of the vortex state and the flower state at the single domain limit

The results should be shown to be independent of the material parameters A and M_s and different spatial discretizations where the edge length of the simulation cells are smaller than the exchange length.

The following table shows the determined single domain limit L for different spatial discretizations of N^3 cubic cells, including the reduced field term energies and average magnetization for the flower and the vortex states as required in the standard problem 1 description. The simulation script is shown in listing 16 in the appendix A.1.3.

			flower state				vortex state			
N	L	e_tot	e_demag	e_exch	e_aniso	$\langle m_\mathrm{z}\rangle$	e_demag	e_exch	e_aniso	$\langle m_\mathrm{x}\rangle$
12	8.42	.3033	.2804	.0173	.0055	.9716	.0787	.1728	.0518	.3466
16	8.44	.3031	.2799	.0175	.0055	.9713	.0786	.1726	.0519	.3491
20	8.45	.3030	.2800	.0176	.0056	.9712	.0786	.1723	.0520	.3512
24	8.46	.3028	.2797	.0176	.0056	.9712	.0784	.1723	.0521	.3509
28	8.46	.3029	.2796	.0176	.0056	.9711	.0786	.1722	.0521	.3520
32	8.46	.3029	.2796	.0176	.0056	.9712	.0786	.1722	.0521	.3525
36	8.47	.3026	.2795	.0176	.0056	.9711	.0784	.1721	.0521	.3516
40	8.47	.3026	.2795	.0176	.0056	.9711	.0784	.1721	.0521	.3521
44	8.47	.3026	.2795	.0176	.0056	.9711	.0784	.1721	.0521	.3521
48	8.47	.3026	.2795	.0176	.0056	.9711	.0785	.1721	.0521	.3523

Not shown in the table are, for every N, $\langle m_\mathrm{x}\rangle \approx 0$ and $\langle m_\mathrm{y}\rangle \approx 0$ for the flower state and $\langle m_\mathrm{y}\rangle \approx 0$ and $\langle m_\mathrm{z}\rangle \approx 0$ for the vortex state. The results converge with finer mesh discretizations and match the results reported at the μMAG website[110] and by Ref. [113].

7.2.4 μMAG standard problem 4

The μMAG standard problem 4 is used to validate that the simulator is able to simulate magnetization dynamics correctly. A Permalloy cuboid of size 500 nm$\times$125 nm$\times$3 nm are simulated. The model includes the Landau-Lifshitz-Gilbert equation with the exchange field, the demagnetization field, and the external field. The material parameters are: The exchange stiffness constant $A = 1.3 \cdot 10^{-11}$ J/m and the saturation magnetization $M_\mathrm{s} = 8 \cdot 10^5$ A/m. The initial state is a static s-state. It is generated by a saturating external field along the $(1, 1, 1)$ direction which is slowly decreased to zero. After the s-state is produced, the magnetization is reversed by applying two different fields $\mu_0\vec{H}_1 = (-24.6, 4.3, 0)$ mT and $\mu_0\vec{H}_2 = (-35.5, -6.3, 0)$ mT for about 10 ns. The standard problem 4 computes for each field, starting from the s-state,

- the average unit magnetization $\langle M_\mathrm{x}\rangle$, $\langle M_\mathrm{y}\rangle$, and $\langle M_\mathrm{z}\rangle$ over time,
- the simulation time and a snapshot of the magnetization when $\langle M_\mathrm{x}\rangle$ first crosses zero.

The results produced by MicroMagnum on the CPU are shown in Fig. 7.3. A discretization of $125 \times 25 \times 1$ cells of size 5 nm $\times$ 5 nm $\times$ 3 nm is used. The simulation script is shown in listing 17 in the appendix A.1.4. The results agree with the other submissions reported at the μMAG group website. The first zero-crossing of $\langle M_\mathrm{x}\rangle$ happens at $t_1 = 0.135$ ns for the first field and $t_2 = 0.137$ ns for the second field. GPU computations reveal widely equal results (not shown here). Here the zero-crossings occur at $t_1 = 0.136$ ns and $t_2 = 0.137$ ns. In addition, the standard problem 4 is simulated using the alternative potential-method for the demagnetization field computation (not shown in figure). Because a three-dimensional cell grid is needed for this method, a discretization grid of $125 \times 25 \times 4$ cells is chosen. The simulation produced $t_1 = 0.139$ ns and $t_2 = 0.137$ ns.

7.2.5 Spin torque standard problem

The spin torque standard problem[111] is defined to test the correct implementation of the spin-torque extension (Eq. 1.18). The dynamic behavior of a magnetic vortex in a Permalloy square of size 100 nm $\times$ 100 nm $\times$ 10nm under the influence of a constant current is examined. The vortex points into the z-direction and curls in-plane counter-clockwise. The exchange stiffness constant $A = 1.3 \cdot 10^{-11}$ J/m and the saturation magnetization $M_\mathrm{s} = 8 \cdot 10^5$ A/m. A Gilbert damping of $\alpha = 0.1$ is used in order to obtain a fast relaxation. For the spin torque extension term, $\xi = 0.05$ and $P = 1$ is chosen. The initial state is a magnetic vortex in equilibrium position at the center of the square. It is generated by initializing the magnetization by an analytic expression and performing a relaxation until an equilibrium state is reached. The vortex is then excited by a current density of $J = 10^{12}$ A/m^2 in the x-direction. The resulting vortex

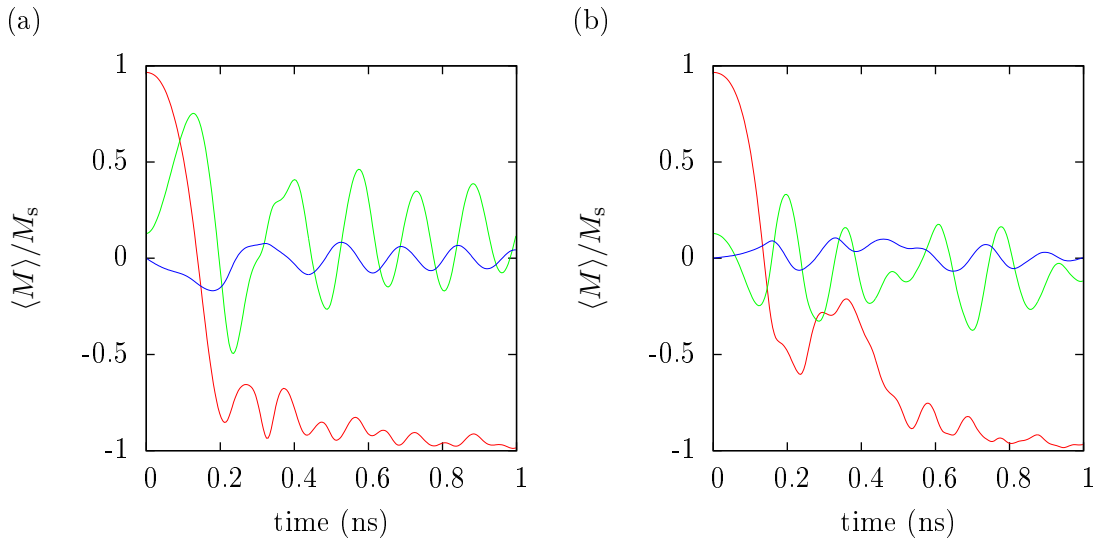


Figure 7.3: Average magnetization over time during the simulation of the standard problem 4 of the μMAG group using an external field of (a) $H_1 = (-24.6, 4.3, 0)$ mT and (b) $H_2 = (-35.5, -6.3, 0)$ mT using MicroMagnum. The red/green/blue lines denote the $x/y/z$-component of the averaged magnetization, respectively.

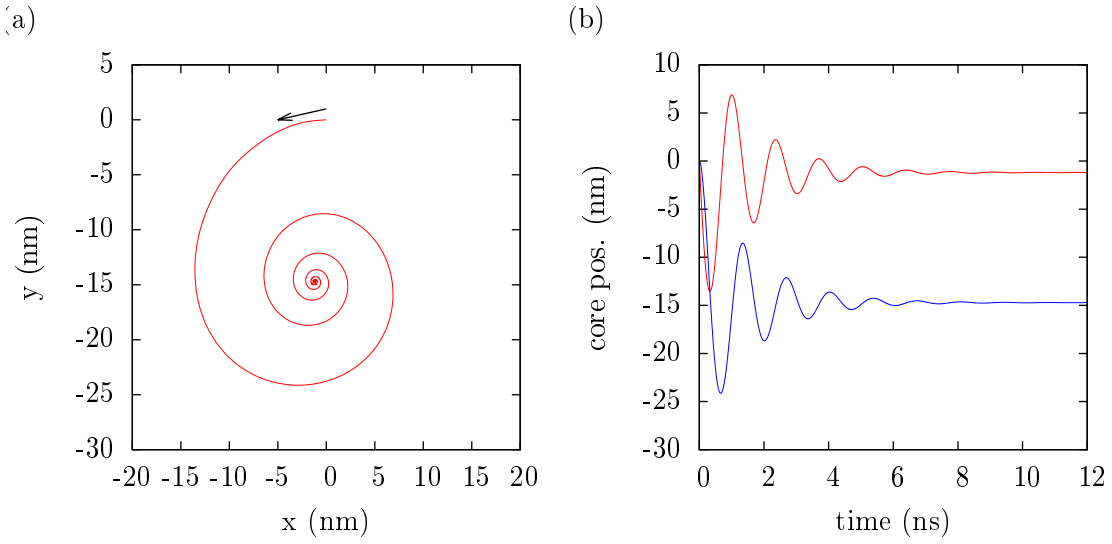


Figure 7.4: Simulation results for the spin torque standard problem[111], calculated by MicroMagnum. (a) Trajectory of vortex core position (x, y) relative to the center of the square, (b) Vortex core position x (blue) and y (red) over time.

core trajectory is then recorded until the vortex reaches a new equilibrium position at $x = -1.2$ nm and $y = -14.7$ nm. The results produced by MicroMagnum are shown in Fig. 7.4. The simulation was conducted twice with two different meshes. The first simulation with $40 \times 40 \times 1$ cells of size 2.5 nm $\times$ 2.5 nm $\times$ 10 nm produced an equilibrium position of $x = -1.187$ nm and $y = -14.487$ nm. The second simulation used a mesh with $100 \times 100 \times 1$ cells of size 1 nm $\times$ 1 nm $\times$ 10 nm to obtain $x = -1.193$ nm and $y = -1.472$ nm. The simulation script is shown in listing 18 in the appendix A.1.5.

The results from the finer grid agree with the reference values[111], i.e. the resulting equilibrium position is calculated at about $x = -1.2$ nm and $y = -14.7$ nm. As suggested in the standard problem definition, the vortex core position was determined with sub-cell precision by the interpolation with a second-order polynomial through the out-of-plane magnetization around the vortex core. The core position is then approximated by its maximum.

7.2.6 Larmor precession test

The test definition is taken from Ref. [25]. It is inspired from the Larmor precession example in the manual of the MagPar micromagnetic simulator[23]. It validates the correct implementation of the Landau-Lifshitz-Gilbert equation and its solution by the time evolver. The only term included in the effective field is the Zeeman field. The Gilbert damping factor α is set to zero so that only the precession of the magnetization vector is included in the model. The frequency of this precession is the Larmor frequency. The tests completes successfully when the simulated results match the analytically predicted Larmor precession. The test uses the material parameters $M_s = 1/\mu_0$ A/m and $\alpha = 0$ and the gyromagnetic ratio $\gamma = 2.210173 \cdot 10^5$ m/(As). The simulation includes one simulation cell of arbitrary size with an initial magnetization of $\vec{m} = (1, 1, 1)$. The external field H is set to 10^6 A/m in the z-direction. The simulation is performed for 0.3 ns. The resulting frequency of the precession is determined by a fit of M_x over t using a sine function. It is then compared to the analytical Larmor frequency of $f_{\text{Larmor}} = \gamma|H|/(2\pi) \approx 35.176$ GHz. The simulation script is shown in listing 18 in the appendix A.1.5. The sine fit yields a frequency of $f = 35.176$ GHz, which matches the analytical f_{Larmor}.

7.3 Performance tests

In this section, the performance of specific parts of the micromagnetic model implementation of MicroMagnum is measured. Both the CPU and GPU implementation are compared. As the base system, a computer with four Intel Xeon X5650 CPUs with a total of 24 cores and 48 GB RAM is used. Each CPU has 6 cores running at 2.67 GHz with a cache of 12 MB and is connected to 12 GB of RAM. An Nvidia M2050 graphics processor is installed on the PCI express bus. Two configurations of MicroMagnum are

used to compare the speed of the CPU and GPU implementations of the micromagnetic model:

Configuration 1 Computation on the Xeon X5650 CPU of the base system.

Configuration 2 Computation on the Nvidia M2050 GPU, controlled by the CPU on the base system.

On the same computer, the effective field computation time is a function of the number of cells in each direction (and whether the mesh is 2-D or 3-D) only. It is independent of the individual cell size and the total simulation volume. Thus for the following benchmarks, only the number of cells is varied. The cell size can be considered arbitrary, e.g. 1 nm × 1 nm × 1 nm. The computation times were measured on an otherwise unloaded computer using the POSIX function `gettimeofday` defined in the `sys/time.h` header. On current Linux systems, this function returns the timing data with a resolution of at least 10^{-6} s. In general the measured computation is performed multiple times in a row until at least two seconds of run time are accumulated, and the average time for one computation is taken.

This section is structured as follows. First, it is determined that the demagnetization field computation consumes the largest part of the total computation time. Thus, the rest of this chapter mainly deals with the performance of the demagnetization field computation. Its computation time is measured in detail and compared for both CPU and GPU implementations. The speed of MicroMagnum is compared with the OOMMF simulator, which is the most widely used micromagnetic simulator. Finally the amount of required memory in dependence of the simulation size is analyzed. Although this is not strictly a performance test, the amount of memory decides whether a simulation fits on the much faster GPU, which typically has much less available on-board RAM than the host computer.

7.3.1 Proportional run times

The first benchmark measures the proportional run times of the different submodules of MicroMagnum running a complete simulation. The simulations are repeated with different numbers of simulation cells. They include the Landau-Lifshitz-Gilbert equation with the demagnetization field, the exchange field, the uniaxial anisotropy field, and an external field. The LLG equation is solved using the six-step Runge-Kutta-Fehlberg method with step size control. The initialization time of the simulations is not included in the benchmark. The total run time is broken down into the following parts: Computation of the Landau-Lifshitz-Gilbert equation, and separately the demagnetization field, the exchange field, the anisotropy field and the external field; the numerical operations of the Runge-Kutta integration; and any time spent by executing Python code, including the overhead imposed by the Python virtual machine. Every part but the last represents time

component	time complexity
Runge-Kutta solver	$\mathcal{O}(n)$
Landau-Lifshitz-Gilbert equation	$\mathcal{O}(n)$
external field	$\mathcal{O}(1)$
demagnetization field	$\mathcal{O}(n \log n)$
exchange field	$\mathcal{O}(n)$
anisotropy field	$\mathcal{O}(n)$
Oersted field	$\mathcal{O}(n \log n)$ [57]
current path solver	$\mathcal{O}(n^2)$ [61]
spin torque term	$\mathcal{O}(n)$
Python code	$\mathcal{O}(1)$

Table 7.1: Asymptotic time complexities of various components of MicroMagnum.

spent in C++ code. The proportional run time of code written in Python is of particular interest. As Python is an interpreted, dynamically typed language, it is typically one to two orders of magnitude slower[114] than compiled, statically typed languages like C++. As pure Python code is not parallelizable due to the global interpreter lock[115], a too high amount of time spent in Python code would, according to Amdahls law, limit the possible speedup gained by parallelizing the non-Python parts, i.e. the numerical routines written in C++. The other proportional parts of the run time should roughly reflect their asymptotic run time complexity in dependence of the number of cells n, as shown in Table 7.1. In the following the proportional run times of the identified components are measured. Figure 7.5 shows the benchmark results for simulations with different numbers of simulation cells, repeated for configuration 1 (CPU) and 2 (GPU). The benchmark data was collected by the profiler included with the Python interpreter as the `cProfile` library. For the GPU measurements the CUDA kernel calls were configured to be synchronous so that the calls return only after the GPU routines have finished to execute. This is necessary to ensure that the measured run times do not overlap. The benchmark shows that the demagnetization field does take up to 80 % on the CPU and up to 60 % on the GPU of the simulation time. This is not surprising as it is the only long-range interaction. The demagnetization field has $\mathcal{O}(n \log n)$ complexity, and all other computations have at most $\mathcal{O}(n)$ complexity. As the cell size is increased the $\log n$ term becomes less relevant and the proportional time of the demagnetization field computation practically converges. For large simulations the run times of the exchange field and the anisotropy field is roughly the same with about 5 % to 10 %. The external field takes about 2 %, the LLG equation about 4 %, and the time evolver about 4 % to 10 % of the total time.

On the GPU configuration, the time spent in the Python interpreter while executing Python code is significant. While for large simulations this overhead is smaller than 1 % on the CPU, the speedup of the numerical routines on the GPU increases the overhead

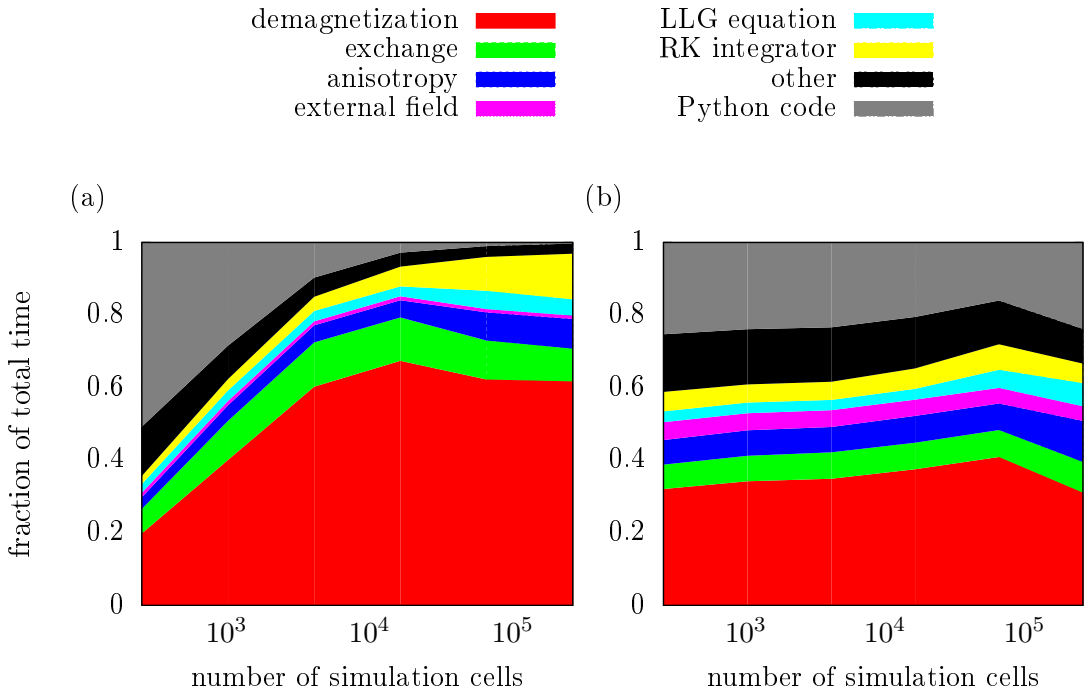


Figure 7.5: Proportional run times of MicroMagnum on (a) configuration 1 (CPU) and (b) configuration 2 (GPU, using synchronous kernel calls) for micromagnetic simulations in dependence of the number of cells. The simulation includes the LLGE with the demagnetization, exchange, uniaxial anisotropy, and external field. The LLGE is solved by the RKF45 method. The run time is broken down into the computation of the fields (red, green, blue, pink), the LLG equation (cyan), the RKF45 integrator (yellow), any other numerical C++ code (black), and any Python code (gray).

to more than 10 % of the total run time. As the Python code in this benchmark runs sequentially and has a run time of 10 %, according to Amdahl's law any additional speedup is limited to a factor of less than 10. However, in production code, the CUDA kernel calls are configured to be asynchronous. This means that the Python code (executed on the CPU) and the CUDA routines can potentially execute at the same time. In order to measure the effect the benchmark was repeated on the GPU configuration with asynchronous kernel calls. To determine the time spent waiting at the synchronization points, the total simulation time for synchronous and asynchronous calls were compared, resulting in a waiting time of about 20 %. It can thus be assumed that, with asynchronous calls, most of the former waiting time is now spent on executing Python code (which also had a run time of roughly 20 %). Thus the overhead imposed by the Python virtual machine is still negligible on the GPU with the currently reached speedups.

Because the computation of the demagnetization field takes most of the time, the following performance tests will mainly deal with the demagnetization field. Additionally the exchange field will be investigated, because it is the only other field term in the basic micromagnetic model that includes neighboring simulation cells for its computation.

7.3.2 Demagnetization field

The computation time for the demagnetization field is measured on the CPU and the GPU configurations for different numbers of simulation cells in both two- and three-dimensional grids of simulation cells, see Fig. 7.6. For two-dimensional grids, the maximum reached speedup by using a GPU is 66. It can be seen that the number of simulation cells influence the computation time in a non-monotonous way. This is because transform sizes result in inefficient computation of the fast Fourier transforms, see section 3.1.2. For meshes with more than 512×512 cells, an average speedup of 40 and more can be expected when good grid sizes are used. For three-dimensional grids, the reached speedup ranges from 18 to 25. This is likely due to the fact that the one-dimensional transforms within the row-column algorithm are much smaller than in the 2-D case and thus provide less parallelization opportunities on the GPU implementation of the FFT. In addition it will be shown that the array rotation routines are a bit slower for the rotation of 3-D arrays than for 2-D arrays. Together, the 2-D field computation is about twice as efficient as the 3-D field computation.

The computation of the demagnetization on the GPU includes several steps. First, a sparse fast Fourier transform is used to transform the components of the magnetization field into the frequency domain. An frequency product with the demagnetization tensor field yields the demagnetization field in the frequency domain. Finally, the three components are transformed back using a sparse FFT to obtain the demagnetization field. Depending on the dimensionality grid, the sparse forward- and inverse FFTs are either two- or three-dimensional. As described in section 3.1.3 and shown in Figs. 3.5 and 3.6, they are composed of a series of one-dimensional FFTs and array rotations. In Fig. 7.7, the proportional run times of the transform, array rotation, and product operations are shown for different numbers of two-dimensional grids of cells on both the CPU and the GPU. It can be seen that the transforms, which include both the forward and the inverse transforms, make up the largest fraction of the computation time. For all grid sizes, the transforms take about 70-80 % of the time on the CPU and about 60-70 % of the time on the GPU. The product and array rotation steps consistently share the rest of the time in about equal parts. Although the implementation of the frequency product is conceptually simpler than the array rotation, it is not faster because it involves the access of the demagnetization tensor which contains six unique components that are stored in the global memory of the GPU.

In the following, the performance of the array rotation is analyzed. The reached performance, measured by the amount of data processed per second, are compared to the calculated peak performance that can theoretically be achieved by the GPU. The procedure to calculate the theoretical peak performance of a CUDA device like the M2050 GPU is taken from the CUDA C Best Practices Guide[116]. The theoretical peak band-

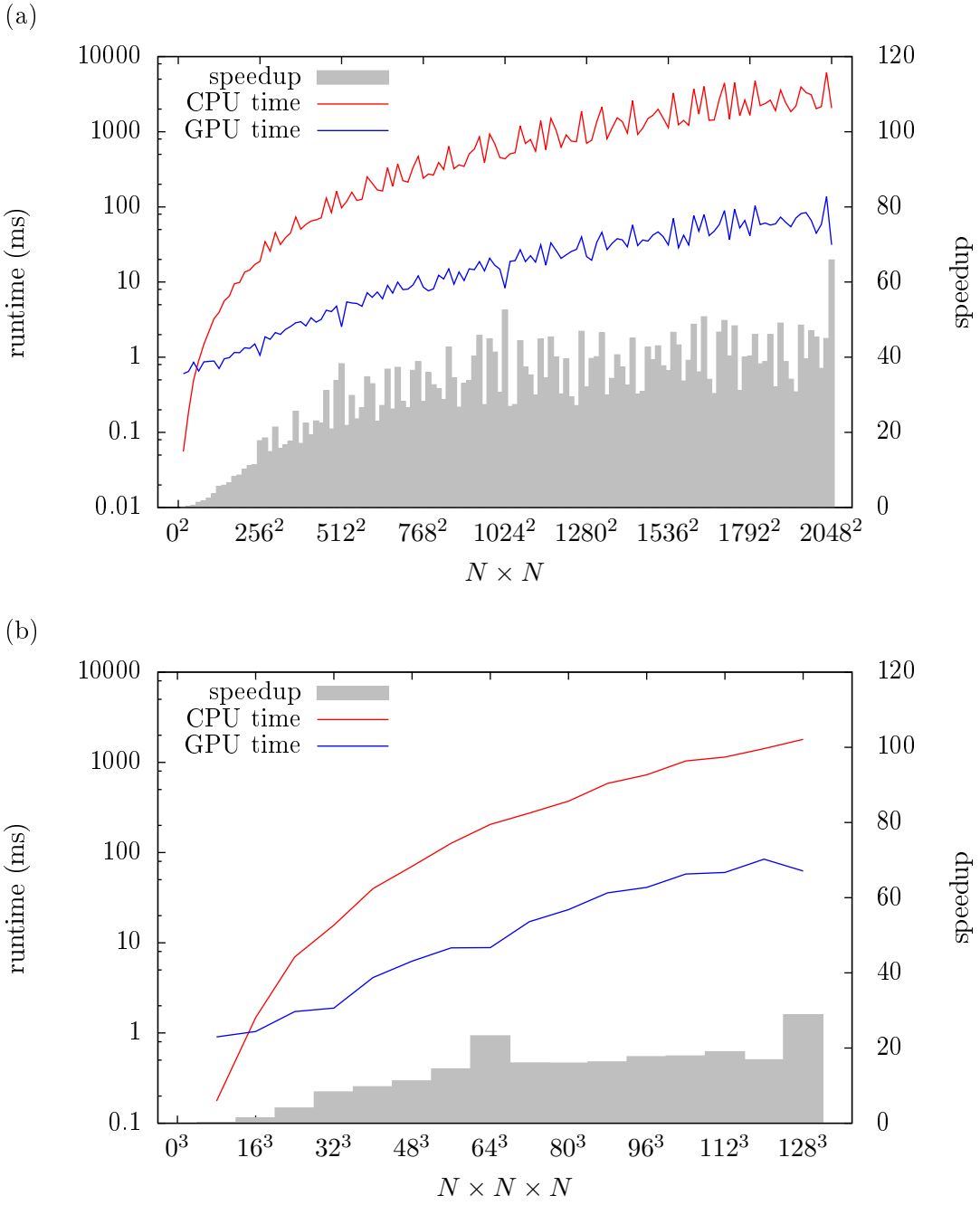


Figure 7.6: Computation time of one demagnetization field computation for (a) different 2-D grids of $N \times N$ cells and (b) different grids of $N \times N \times N$ cells. The red (blue) line shows the run time on the CPU (GPU) configuration. The grey bars show the speedup, e.g. the ratio of the CPU time to the GPU time. The zero-padding mode *round_4* (see Tab. 3.1) is used.

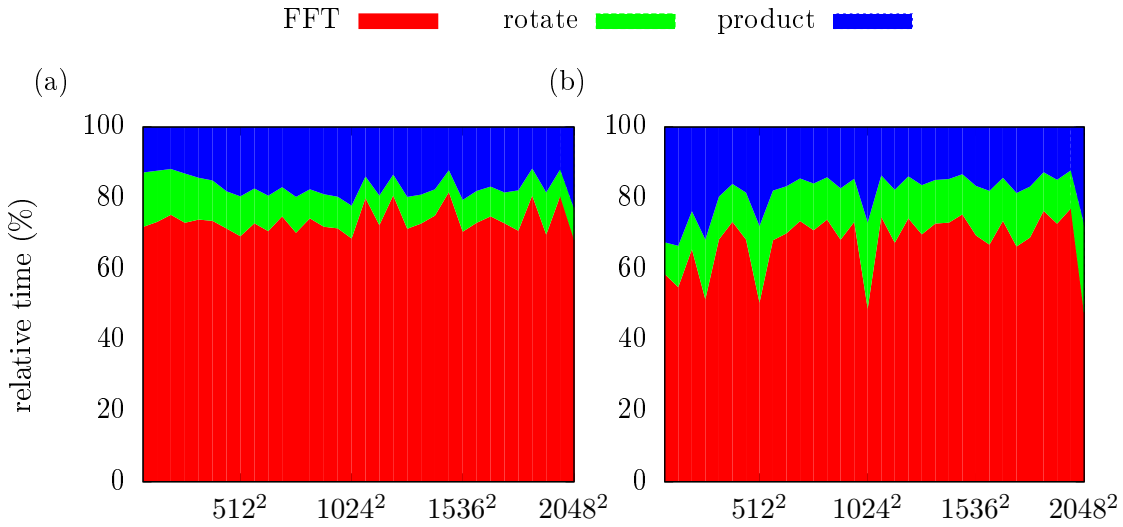


Figure 7.7: Proportional run times of the demagnetization field computation on (a) the CPU and (b) on the GPU of the iterated one-dimensional FFTs and inverse FFTs (red), array rotation, zero-padding and unpadding (green), and multiplication in the frequency domain (blue) on different two-dimensional grids of $N \times N$ cells.

width of a CUDA device is the amount of data per time[1] that can be transfered between the GPU and the on-board memory of the graphics card. It can be calculated using the hardware specifications of the graphics card, namely the bus type, the memory clock rate and the data bus width. The GPU of configuration 2 has a M2050 graphics processor that has a memory clock rate of 1546 MHz and a data bus width of 384 bits. Its peak rate is calculated as $2 \cdot 384/8$ bytes $\cdot$ 1546 MHz = 138.2 GB/s. The factor 2 is due to the fact that the GDDR5 memory of the M2050 is capable of transferring two memory blocks per clock cycle. The effective bandwidth that is reached by a given CUDA function is the amount of data per time that is transfered between the GPU and its memory. It is calculated as $(N_r + N_w)/t$, the number of bytes read and the number of bytes written per time. It is by definition lower than the device's peak bandwidth, which defines an upper limit for functions that work on data stored in memory. Table 7.2 shows the bandwidth of the array rotation algorithm, given in listing 22 in the appendix, on the CPU and on the GPU. It can be seen that the GPU implementation has a much higher bandwidth than the CPU implementation. On the CPU, the bandwidth decreases with larger arrays which is likely due to cache effects. On the GPU, the bandwidth increases with larger arrays due to the constant overhead of calling CUDA functions from the host system. Also three-dimensional array rotations on the GPU are less efficient than two-dimensional rotations. This is likely due to the fact that the three-dimensional CUDA rotation function contains more arithmetic operations to calculate the pointers to the location of the source cell and the destination cell. Also, in the current implementation, the memory reads are not completely coalesced. In total, a data bandwidth of up to 67 GB/s is reached for

[1] given in Gigabytes per second (GB/s = 2^{30} bytes/s)

	bandwidth (GB/s)			bandwidth (GB/s)	
array size	CPU	GPU	array size	CPU	GPU
$128 \times 128 \times 1$	3.09139	6.61886	$16 \times 16 \times 16$	2.74521	1.32414
$256 \times 256 \times 1$	2.82188	20.7008	$32 \times 32 \times 32$	2.43229	11.2789
$512 \times 512 \times 1$	1.7044	44.4882	$64 \times 64 \times 64$	1.6626	32.7198
$1024 \times 1024 \times 1$	1.02023	61.581	$128 \times 128 \times 128$	0.946076	40.2831
$2048 \times 2048 \times 1$	0.876946	67.1275			

Table 7.2: Bandwidth, measured in bytes per second, of the *rotate-zeropad* algorithm (specified in listing 22) on configuration 1 (CPU) and on configuration 2 (GPU). The bandwidth includes read and write accesses to memory.

	CPU			GPU		
cells	t_N (ms)	t_S (ms)	speedup	t_N (m)	t_S (ms)	speedup
$16 \times 16 \times 16$	1.60	1.10	1.45	0.36	0.26	1.38
$32 \times 16 \times 16$	3.20	2.18	1.47	0.48	0.34	1.41
$32 \times 32 \times 16$	7.08	4.68	1.51	0.73	0.52	1.40
$32 \times 32 \times 32$	16.94	11.09	1.52	1.19	0.83	1.43
$64 \times 32 \times 32$	37.01	24.32	1.52	1.99	1.37	1.45
$64 \times 64 \times 32$	89.11	57.43	1.55	3.63	2.47	1.47
$64 \times 64 \times 64$	213.02	135.95	1.57	6.88	4.65	1.48
$128 \times 64 \times 64$	461.12	300.86	1.53	12.27	8.30	1.48
$128 \times 128 \times 64$	933.03	614.62	1.52	21.45	14.71	1.46
$128 \times 128 \times 128$	1917.29	1258.24	1.52	37.27	25.69	1.45

Table 7.3: Computation time of the demagnetization field calculation with different spatial discretizations on the CPU and the GPU using the demagnetization tensor method (t_N) and the scalar potential method (t_S). The speedup is the ratio between t_N and t_S. (Table and caption adapted from Ref. [58], Copyright 2012 IEEE.)

large 2-D arrays, which is about half of the theoretical peak performance of the GPU.

7.3.3 Demagnetization field (scalar potential method)

The computation speed for the demagnetization field using the scalar potential method as described in section 2.1.4 is compared to the speed of the demagnetization tensor method. The results, which are taken from the publication in Ref. [58], are reproduced in table 7.3. Different 3-D cell grids of increasing size are examined. The benchmark reveals that there is a consistent speedup due to the usage of the scalar potential method. The consistent speedup is due to the fact that the method always requires four instead of six inverse fast Fourier transforms, independent of the grid size. On the CPU there is an average speedup of $1.52 \approx 6/4$. On the GPU a speedup of about 1.44 is reached. The slightly lower speedup on the GPU is due to the fact that the multiplication in the

frequency domain for the convolution takes a larger fraction of the time on the GPU than on the CPU.

7.3.4 Exchange field

After the demagnetization field, the exchange field and the anisotropy field are the second-most computation time intensive fields in the basic micromagnetic model. Figure 7.8 shows the computation time of the exchange field on configurations 1 (CPU) and 2 (GPU) and the respective speedup for different numbers of simulation cells in 2-D (with a grid size of $N \times N$) and 3-D grids (with a grid size of $N \times N \times N$).

On 2-D grids (Fig. 7.8 (a)), both the CPU and GPU run times show a linear dependence on the number of cells. An average GPU speedup of about 20 to 28 is reached for cell grids larger than 256×256. For smaller grids the speedup decreases due to the constant overhead of calling CUDA routines on the GPU. At 128×128, the speedup has halved to about 10. In this benchmark the break-even point for using the GPU is approximately at 32×32 simulation cells.

On 3-D grids (Fig. 7.8 (b)), a speedup of up to 15 is reached. In comparison to 2-D grids, this speedup is significantly lower. The reason is that move from 2-D to 3-D on the GPU introduces more overhead than on the CPU. On the CUDA implementation, the limited amount of shared memory of a thread block forces the thread block size to become $8 \times 8 \times 8$, which increases the number of ghost cells (see section 5.2.6). In 2-D the ghost cells make up roughly 20 % of the cells stored in shared memory. In 3-D, the ratio increases to 48 %. Ghost cells result in additional memory accesses that are partly uncoalesced, thus causing the lower speedup.

7.3.5 Comparison to OOMMF

The performance of the demagnetization field computation is compared with the *Object Oriented MicroMagnetic Framework* (OOMMF)[19]. OOMMF is the most widely used finite difference micromagnetic simulator in the micromagnetic community with over a thousand citations in publications. OOMMF is CPU-only and does not support parallel processing in the current stable release. The most recent alpha release adds support for parallel computations using threads, where a speedup of up to 3 using four processors cores was achieved[117]. Kanai et al. report a speedup of up to 5.6 on eight SMP cores with a custom code base[118]. In the following the performance of the demagnetization field computation of MicroMagnum is compared to the same computation of the latest OOMMF version including thread support. In the benchmark an Intel Core i5-3570K CPU running at 3.40GHz with four cores is used.

The demagnetization field computation time for different numbers of cells are shown in Table 7.4. The CPU computations are done using 64-bit precision and the GPU computations using 32-bit precision arithmetics. If one CPU core is used, the speed of

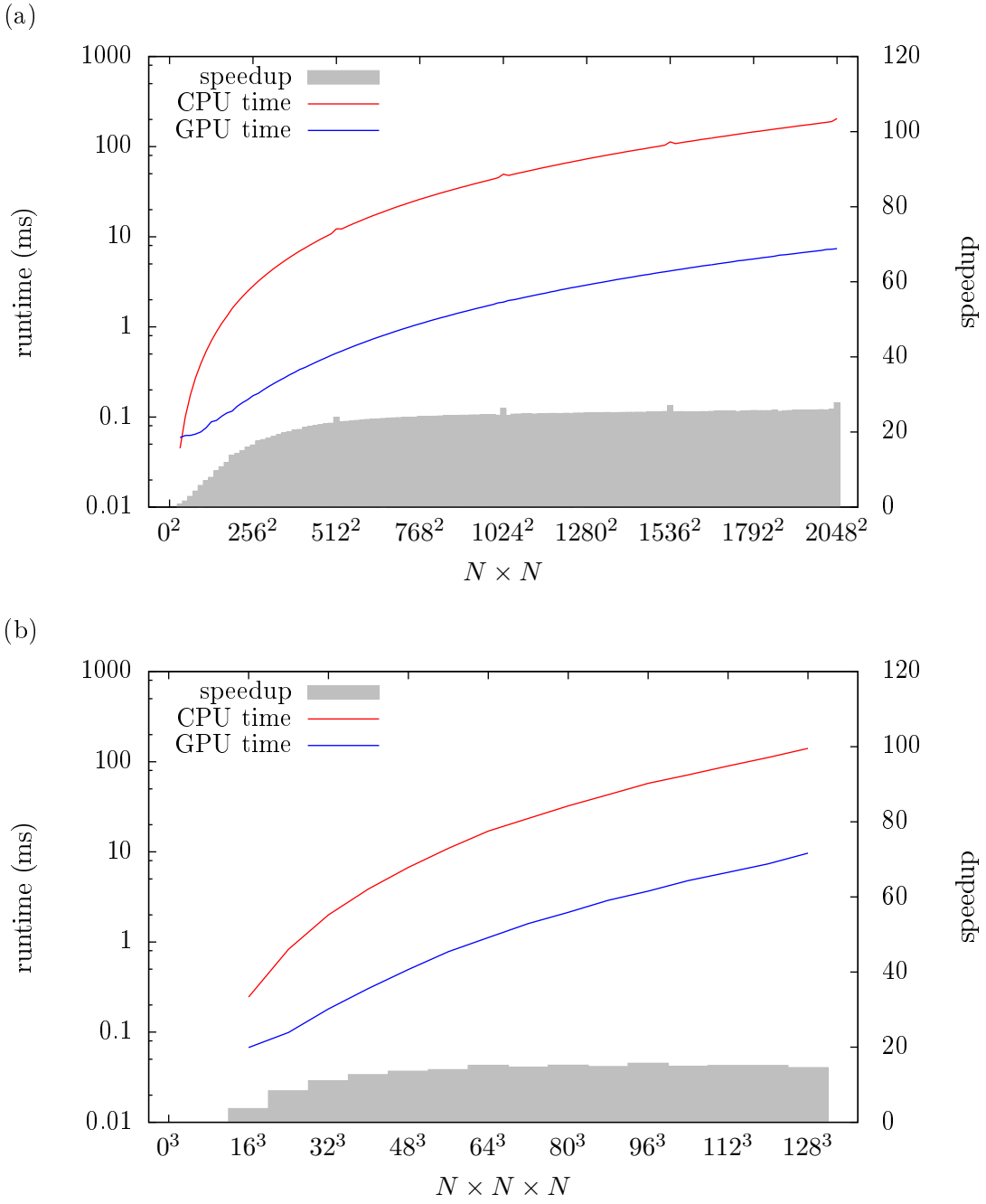


Figure 7.8: Computation time of one exchange field computation for (a) different 2-D grids of $N \times N$ cells and (b) different grids of $N \times N \times N$ cells. The red (blue) line shows the run time on the CPU (GPU) configuration. The grey bars show the speedup, e.g. the ratio of the CPU time to the GPU time.

	MicroMagnum		OOMMF	
cells	CPU	GPU	CPU × 1	CPU × 4
128 × 128	2.00	0.38	2.25	0.71
256 × 256	12.18	0.77	11.11	3.55
512 × 512	59.90	2.16	52.81	15.77
1024 × 1024	271.95	7.74	247.11	72.99
2048 × 2048	1386.90	32.31	1106.40	379.73
4096 × 2048	2840.86	66.16	2361.46	907.61
200 × 200	6.61	–	10.06	3.78
400 × 400	36.16	–	47.97	14.11
800 × 200	184.67	–	223.34	80.34
1600 × 1600	742.63	–	1015.35	347.09

Table 7.4: Run time of the demagnetization field computation on MicroMagnum on CPU and GPU, and on OOMMF on the CPU for different grids of simulation cells. The GPU is on a Nvidia M2050 graphics card, and the CPU is an Intel Core i5-3570K CPU running at 3.40GHz with four cores.

MicroMagnum is comparable to OOMMF for grid dimensions that are a power of two (i.e., $N = 2^n$ for an integer n). Here OOMMF is roughly 10 to 20 % faster. For non-power-of-two sizes, MicroMagnum is faster because internally OOMMF only supports power-of-two fast Fourier transforms, requiring addition zero-padding of the magnetization array. If OOMMF is run with four parallel cores, it gets a speedup of 3 to 4. In contrast, if MicroMagnum is run on the GPU, a speedup of roughly 30 to 40 is reached[1]. MicroMagnum supports NPOT sizes due to the use of the FFTW and CUFFT libraries that provide optimized algorithms for both power-of-two and non-power-of-two fast Fourier transforms. In conclusion MicroMagnum has a comparable performance to OOMMF on single-threaded CPUs. If both simulators run parallel, i.e. on the GPU and on a quad core CPU, MicroMagnum is about 10 times faster due to the use of the GPU.

7.3.6 Memory usage

Table 7.5 shows the amount of temporary working memory that is needed per simulation cell for the computation of the effective fields. There are different values for 2-D and 3-D meshes for the demagnetization field and the Oersted field computation because of the required extra zero-padding in the z-dimension in the 3-D case. For the exchange field and the anisotropy field computation, which are both local fields, the field is computed on the fly from the magnetization array and thus does not require extra temporary working memory. In contrast, the demagnetization field and Oersted field computation

[1]This speedup differs from the previously reported maximum speedup of 66, because in this benchmark a faster CPU was used.

effective field	floats (per cell)	32-bit floats (bytes/cell)	64-bit floats (bytes/cell)
demagnetization field (2-D)	36	144	288
demagnetization field (3-D)	72	288	576
Oersted field (2-D)	24	96	192
Oersted field (3-D)	48	192	384
exchange field	0	0	0
anisotropy field	0	0	0
external field	0	0	0

Table 7.5: Amount of temporary memory needed per simulation cell for the computation of the effective fields.

requires the largest amount of memory. For the demagnetization field, the total amount of additional memory is the sum of the size of the precomputed demagnetization tensor and the size of the buffers to hold intermediary fast Fourier transform results of the zero-padded magnetization array. The number of floating point numbers to store the six unique components of the demagnetization tensor in memory is roughly $6N2^d$ where N is the number of cells, and $d \in \{2, 3\}$ the dimension of the grid. Similarly, the amount of working space for the magnetization array is $3N2^d$ for each of the components M_x, M_y, and M_z. A technique that reduces the amount of memory for this buffer is described in [117]. The reduction is achieved by interleaving a subset of the lower-dimensional sub-transforms of the fast Fourier transform with the multiplication in the frequency domain. This technique is used in the OOMMF simulator, but is not included in MicroMagnum.

In conclusion, the size of the discretizing mesh is limited by the amount of RAM that is installed on the host computer or, in the case of GPU computation, the amount of global memory available to the GPU. For example, computing the demagnetization field using a 3-D discretizing mesh with 5.000.000 cells requires 2747 MB of GPU RAM for the demagnetization field computation, which just fits into the 3072 MB that is installed on the Nvidia M2050 graphics processors. On the CPU, the amount of needed memory doubles due to the employed 64 bit precision arithmetics.

7.4 Conclusion

Several test are applied to test the correctness and the efficiency of the MicroMagnum simulator. The correctness is verified by unit tests and the μMAG standard problems. The efficiency is examined by several benchmarks of MicroMagnum's software components. Depending on the component, speedups of one to two orders of magnitude are reached on the GPU in comparison to the CPU implementations. It is shown that while the overhead introduced by the Python virtual machine is significant, its running time is clearly overlayed by parallel computations of the GPU. The speed of MicroMagnum on

the CPU is comparable to the speed of OOMMF running on one thread, the most widely used simulator in the micromagnetic community. On the GPU, MicroMagnum is about ten times faster than OOMMF running on 4 threads. Finally, it was shown that the used GPUs have enough on-board memory to host simulations with millions of cells.

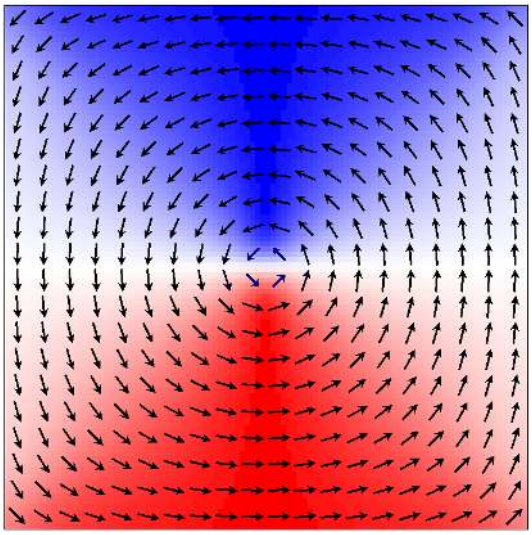


Figure 8.1: Computed vortex ground state in a Permalloy square ($200 \times 200 \times 20$ nm^3) with positive polarization $p = 1$ and chirality $c = 1$. The angle between the M_x and the M_y component is color-coded.

8 Use Case: Non-linear Magnetic Vortex Core Dynamics

In recent years magnetic vortices have become one of the most studied subjects in the micromagnetic community. Many publications deal with the dynamics of vortices. The focus of experiments and micromagnetic simulations about vortices has lead to the linear, nonlinear and highly nonlinear dynamics of vortex core switching driven by external fields or spin-polarized currents. Insights from these investigations have been used to propose a storage concept, the so-called vortex random access memory[1] (VRAM). This section introduces the magnetic vortex and deals with the simulation of magnetization dynamics of vortices using MicroMagnum. The simulated results are summarized from the article "Non-linear magnetic vortex gyration"[37] by André Drews, Benjamin Krüger, Gunnar Selke, Thomas Kamionka, Andreas Vogel, Michael Martens, Ulrich Merkt, Dietmar Möller and Guido Meier, Copyright 2012 by the American Physical Society. For this publication MicroMagnum was used to run a large number of simulations.

8.1 Magnetic vortex configuration

A magnetic vortex forms in ferromagnetic thin-film elements[41] where the magnetization curls in-plane around an out-of-plane region, the so-called vortex core, see Fig. 8.1. The orientation of the curling in-plane magnetization is called chirality c. A mathematical positive (negative) orientation of the magnetization is defined as $c = -1$ ($c = 1$). The orientation of the vortex core is called polarization p which can point either in the plane ($p = -1$) or out of the plane ($p = 1$). The core typically has a diameter of a

few nanometers[42]. The vortex is an energetic ground state which forms in softmagnetic thin-film elements like Permalloy with a lateral size from $\approx$ 100 nm up to some microns. The formation of the vortex can be explained by considering the exchange and the demagnetization energy of Permalloy. The demagnetization energy is minimized by aligning the magnetization vectors in parallel to the sample's surface to avoid surface charges resulting in the curling of the magnetization. In the center of the curling magnetization pattern the magnetization vectors would align antiparallely. This would drastically increase the exchange energy which wants adjacent magnetization vectors to align in parallel. Thus the magnetization at the center points out-of-plane resulting in the formation of the vortex core.

In the following the usage of MicroMagnum for the investigation of magnetic vortices is presented. Since the magnetic vortex is a widely used examination object MicroMagnum includes predefined routines to form the ground state and to analyze the magnetization dynamics. The approximate ground state of a magnetic vortex in Fig. 8.1 is generated by employing the `vortex.magnetizationFunction` function of the `vortex` toolbox. The function applies the parametrization

$$
\begin{aligned}
m_{\mathrm{x}} &= \frac{c \cdot (y - y_0)}{\sqrt{(x - x_0)^2 + (y - y_0)^2 + R^2)}} \cdot M_s \\
m_{\mathrm{y}} &= \frac{x - x_0}{\sqrt{(x - x_0)^2 + (y - y_0)^2 + R^2)}} \cdot M_s \\
m_{\mathrm{z}} &= \frac{R}{\sqrt{(x - x_0)^2 + (y - y_0)^2 + R^2)}} \cdot M_s
\end{aligned}
\tag{8.1}
$$

to the magnetization at each position (x, y, z) in the sample with the position of the center x_0, y_0, the saturation magnetization M_s, the chirality c, and the diameter of the core R. The parametrization is an approximation of the desired ground state magnetization pattern and thus has an energy in the vicinity to the energy minimum to reduce computation time at the relaxation of the magnetization pattern. Currently MicroMagnum does not support direct energy minimization, e.g. by using gradient methods. The energy minimum is computed by disabling the precession term in the Landau-Lifshitz-Gilbert equation. The magnetization dynamics starts at the given vortex parametrization and relaxes to the energetic minimum. The stopping criterion is a small remaining change of the maximum angle between each magnetization vector and its neighbors of the whole sample, given in degrees per nanosecond. The MicroMagnum script to create a vortex configuration is shown in listing 12. In Lines 9–12 the vortex is parametrized and in line 13 the vortex is relaxed. In Line 15 the resulting magnetization pattern is stored on disk for later use.

```
1 from magnum import *
2
3 world = World(
```

```
  RectangularMesh((100, 100, 1), (2e-9, 2e-9, 2e-8)),
  Body("sample", Material.Py(), Everywhere())
)

solver = create_solver(world, [StrayField, ExchangeField], do_precess=
    False, log=True)
solver.state.M = vortex.magnetizationFunction(
  core_x=1e-7, core_y=1e-7,
  polarization=1, core_radius=1e-8
)
solver.relax(max_deg_per_ns = 5)

writeOMF("groundstate.omf", solver.state.M)
```

Listing 12: Computation of the vortex ground state in Fig. 8.1.

8.2 Vortex dynamics

Experiments and micromagnetic simulations have shown that the vortex forms without external excitations only due to the internal exchange and demagnetization energy. If the core is deflected by a magnetic field or spin-polarized current and the excitation is turned off it performs a spiral trajectory around its equilibrium position in the center. For small amplitudes of excitation close to the resonance frequency the gyration has been described within a two-dimensional harmonic oscillator model. In this picture the core gyrates like a quasi particle in a potential

$$V(x, y) = -\frac{\kappa}{2}(x^2 + y^2) \tag{8.2}$$

that is given by the intended symmetry of the in-plane magnetization pattern in order to minimize the demagnetization and exchange energy. The potential includes the components x and y of the core position relative to the core position at the ground state x_0, y_0 and the strength κ of the curvature of the potential. The parameter κ depends on the thickness t and the lateral size l of the sample[119].

For an alternating excitation and a small amplitude of excitation the core gyrates on an elliptical trajectory in the steady state. Elliptical trajectories are a typical characteristic of two-dimensional harmonic oscillators. To obtain the dynamical response of a vortex to a wide range of excitation frequencies a resonance curve can be generated. In the linear regime the resonance curve has a Lorentz profile with a peak at the resonance frequency where the vortex translates the maximum energy from excitation into gyration. Generating a resonance curve requires an accurate determination of the vortex core position. From the raw simulated data a rough approximation of the core position is given by the position of the simulation cell with the largest out-of-plane component $|M_z|$. Usually, the simulation cell size is close to the exchange length of about 4 nm for Permalloy. The vortex core has a diameter of about 20 nm. For the determination of

the vortex core position the accuracy of one simulation cell length is not sufficient. A deviation of the core position of about 1 nm could strongly distort the determination of the resonance curve, since a resonance curve of vortices usually has a narrow shape of the Lorentz profile. A spatial resolution far below 1nm is required to determine the resonance frequency. In order to get a higher accuracy the function of the vortex toolbox of MicroMagnum applies polynomial fits of second order by employing Lagrange interpolation in x- and $y-$ direction by considering the cell with the maximum M_z value and its next two neighbors[120]. The function reaches an accuracy of about one thousands of the vortex core diameter which is sufficient for core dynamics.

The harmonic oscillator model adequately describes core gyration in the linear regime. In many applications, like the operation of a vortex in a VRAM, large amplitudes of excitation would be required covering the nonlinear regime. To find an analytical description of core gyration in the nonlinear regime the harmonic potential in Eq. 8.2 is expanded by higher order terms leading to the potential

$$V(x,y) = \frac{\kappa}{2}(x^2 + y^2 + a(x^2+y^2)^2 - bx^2y^2), \tag{8.3}$$

which includes the nonlinear parameters a and b that depend on the lateral size l and the thickness t of the sample in a different way than the harmonic parameter κ. Using this potential and the Thiele equation[121] with the collective coordinate approach the nonlinear equation of vortex motion is derived,

$$\begin{aligned}\begin{pmatrix} v_x \\ v_y \end{pmatrix} = &-p\omega_f \begin{pmatrix} y + 2ay^3 + 2ax^2y - bx^2y \\ -x - 2ax^3 - 2axy^2 + bxy^2 \end{pmatrix} \\ &-\Gamma \begin{pmatrix} x + 2ax^3 + 2axy^2 - bxy^2 \\ y + 2ay^3 + 2ax^2y - bx^2y \end{pmatrix} + \begin{pmatrix} u \cdot \cos(\Omega t) \\ 0 \end{pmatrix}\end{aligned} \tag{8.4}$$

with the free frequency ω_f, the damping Γ, the velocity due to current excitation u, and the exciting frequency Ω, see description of Eq. (3) in Ref. [37]. In comparison to the micromagnetic model this oscillator model only includes a few degrees of freedom. To test if the reduction of degrees of freedom is a good approximation this model is compared to micromagnetic simulation using MicroMagnum. Vortex gyration is simulated and compared to a numerical integration of Eq. 8.4. For the simulations of resonance curves a large number of simulations are performed. Vortices in Permalloy squares with varying thicknesses $t = 10, 20, 30$ nm and lateral sizes $l = 200, 500$ nm are excited by alternating currents of amplitudes $2{\cdot}10^{10}, 5{\cdot}10^{10}, 1{\cdot}10^{11}, 1.5{\cdot}10^{11}, 2{\cdot}10^{11}$ A/m^2 and varying excitation frequencies Ω. The large number of simulations is performed concurrently on a computer cluster by employing the `Controller` class of MicroMagnum. The Python simulation script in listing 13 includes arrays that cover the varying parameters, i.e. lateral size, thickness, amplitude, and exciting frequency. The input of the `Controller` object is the

name of the Python function that represents the simulation which is defined the script and parameter array.

```
#!/usr/bin/python
from magnum import *
import math

# Simulation function
def nonlinsim(l, t, Omega, J):
    mesh = RectangularMesh((l/4e-9, l/4e-9, 1), (4e-9, 4e-9, t))
    world = World(mesh, Body("all", Material.Py(), Everywhere()))
    core_x, core_y = l/2.0, l/2.0

    # Load vortex groundstate from file
    M0 = readOMF("square_%s_%s_%.0f.omf" %
                     (int(l*1e9), int(l*1e9), (thickness*1e9)))

    # Excite vortex by an AC current and save vortex core trajectory
    solver = create_solver(world,
        [StrayField, ExchangeField, AlternatingCurrent, SpinTorque],
        log=True)
    solver.state.M = M0
    solver.state.j_amp = (j, 0, 0)
    solver.state.j_freq = (Omega, 0, 0)

    log = DataTableLog(
        "square_%s_%s_%.0f_j%.1e_%.3e.odt" %
            (int(l*1e9), int(l*1e9), (thickness*1e9), j_amp, j_freq))
    log.addColumn(
        ("Vortexcore X-pos.", "core_x"), ("Vortexcore Y-pos.", "core_y"
            ),
        lambda state: vortex.findCore(solver, l/2, l/2))
    solver.addStepHandler(log, condition.EveryNthStep(10))

    solver.solve(condition.Time(50*2*math.pi/j_freq))

# Parameter space
l     = [2e-7, 5e-7]                        # lateral size
t     = [1e-8, 2e-8, 3e-8]                  # thickness
Omega = [1e9, (~100 entries omitted), 5e9]  # frequency
J     = [2e10, 5e10, 1e11, 1.5e11, 2e11]    # current

c = Controller(nonlinsim, l, t, Omega, J)
c.start()
```

Listing 13: Excitation of a vortex by an AC current with varying simulation parameters.

In these lines of the simulation script thousands of simulations are performed concurrently on different nodes of the computer cluster. The resulting resonance curves are illustrated in Fig. 8.2. The lines are numerical solutions of Eq. 8.4, which are fitted to the simulated results and show good accordance. From these fits the parameters a and b in Eq. 8.3 are

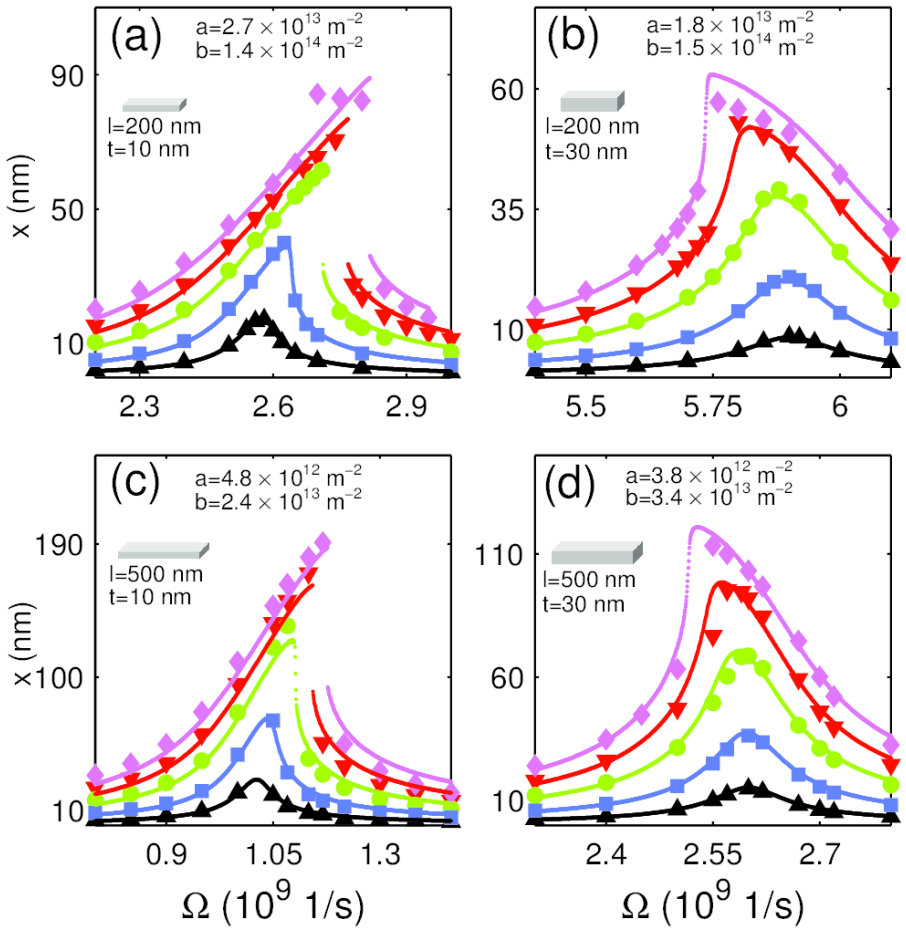


Figure 8.2: Resonance curves of gyrating vortices driven by spin-polarized currents. The maximum core displacement in x direction versus excitation frequency Ω is shown. The symbols show simulated results for current densities from 2×10^{10} Am^{-2} to 2×10^{11} Am^{-2}. The lines are fits according to Eq. 8.4. (Figure and caption adapted from Ref. [37], Copyright 2012 by the American Physical Society.)

obtained.

The simulations show that in the nonlinear regime, i.e. for an increase of the current amplitude, the resonance curves deform. The deformation depends on the thickness of the magnetic vortices. Thin (thick) vortices show a blue (red) shift of the resonance curve, see Fig. 8.2. The red shift of thick samples could also be verified by micromagnetic simulations and XMCD experiments of magnetic field induced vortex gyration of a Permalloy square $1050 \times 1050 \times 50$ nm^{3}as depicted in Fig. 8.3. A deformation is a typical characteristic of nonlinear dynamics, but the thickness dependence is a special feature of vortices. A shift of the resonance frequency in the nonlinear regime could for example be important to achieve low energy consumption by exciting vortices at resonance in VRAMs by adapting the excitation frequency on the amplitude of excitation to match resonance.

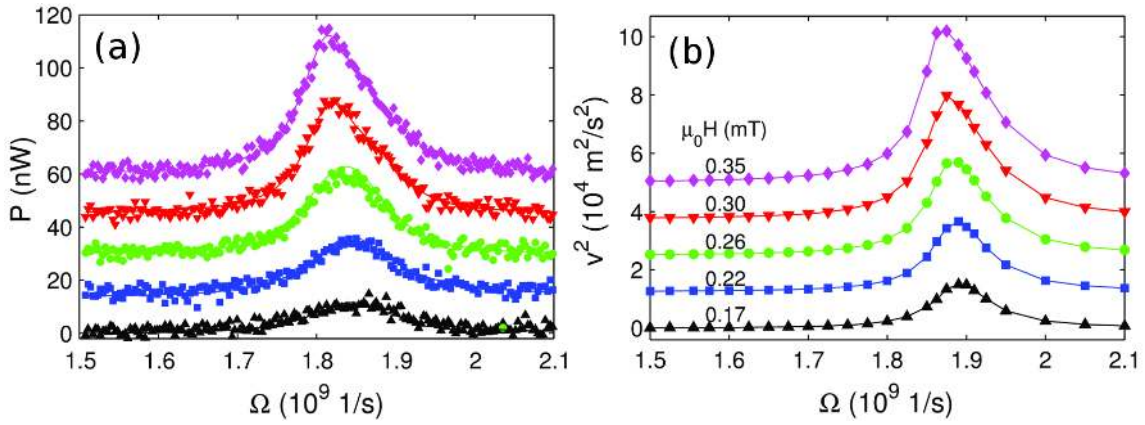


Figure 8.3: (a) Power absorbed by the squares as obtained from field-induced ferromagnetic resonance measurements at different strengths of the exciting magnetic field $\mu_0 H$. (b) Square of the velocity of field-induced gyration from micromagnetic simulations. The lines are guides to the eyes. For clarity the resonance curves are offset successively by $\Delta P = 15$ nW and $\Delta v^2 = 1.25 \times 10^4$ m^{2}s^{-2}. (Figure and caption adapted from Ref. [37], Copyright 2012 by the American Physical Society.)

8.3 Conclusion

This use case shows the impact of using MicroMagnum and its small simulation scripts to obtain a large amount of scientific results. Hundreds of simulations have been executed with different values of the parameters lateral size l, thickness t excitation frequency Ω, energy curvature parameters a and b of magnetic vortices. For each parameter set of a simulation the Python scripting language allows an intuitive scripting of relaxing a magnetic vortex and subsequently excite it by a spin-polarized current j until it reaches its stationary motion. From the scan of the parameters space a dependence of the sample thickness on the frequency of nonlinear magnetic vortex gyration could be observed. This is a fundamental scientific result, since usually the gyration frequency of a nonlinear oscillator mainly depends on the amplitude of gyration.

Conclusion and Outlook

In this work the performance of micromagnetic simulations is increased on the algorithmic as well as on the hardware level. Algorithms based on Fourier transforms and the finite difference method are adapted to the CPU as well as to the graphics processing units programming model to compute the micromagnetic model efficiently. On graphics processing units, the algorithms run massively parallel and yield a speed up of a factor of up to 40 compared to single-threaded CPU computations. Each iteration in time is performed by solving the Landau-Lifshitz-Gilbert equation by the Euler and the Runge-Kutta method. For each iteration step the total effective field is computed. The most time consuming part of the computation of the total effective field is the convolution at the demagnetization and Oersted field computation, which requires about 70% of the total computation time. The convolution theorem which employs the fast Fourier transform reduces its computation time of the convolution. Additionally, implementation details are presented which drastically increase the speedup of the total computation time. In case of CPU and GPU implementation, zero padding and symmetry considerations and sparse Fourier transforms lower the computation time. The algorithm including Fourier transforms is formulated via transpositions to increase the spatial locality of memory accesses. For GPU implementation these transpositions are performed in the shared memory to exploit the larger data bandwidth between streaming processors and shared memory in comparison to the lower data bandwidth between the streaming processors and the global memory. Operations in the shared memory require an elaborate reordering of the threads in the memory area. Only the usage of the shared memory results in strong speed ups. Further software criteria beside the performance are considered, namely maintainability, extendability, usability, correctness, by developing a multi-layer modular software architecture of the micromagnetic simulation tool MicroMagnum. On the bottom layer performance intense algorithms are implemented in C++. On higher layers physical modules are implemented in Python, which is an easy-to-use interpreter language. Rapid development times allow an easy extension of the simulation tool by further physical modules. The modules can be easily connected to achieve a multi-physical approach. The usability of the simulation tool is further increased by developing a domain specific language. Every user can easily build up a simulation example by writing Python scripts. The correctness is achieved by performing unit test as well as system tests. The well-established system tests, called the standard problems 1 to 5 of the μMAG group are performed to ensure the correctness of MicroMagnum. MicroMagnum is available at the website

`http://micromagnum.informatik.uni-hamburg.de`,

and the source code is available under the GNU General Public License[29] at

`http://github.com/MicroMagnum/MicroMagnum`.

Users from Chile, France and Germany reported that MicroMagnum is subjectively easier to use than the established micromagnetic simulator OOMMF due to the imperative rather than declarative nature of the simulation definition language. MicroMagnum has become a successful micromagnetic simulator that has attracted many users[122, 123, 124, 125, 37, 126].

Future Development of MicroMagnum .

This work describes the state of MicroMagnum at version 0.2. The development of MicroMagnum is an ongoing project with the following roadmap. The next version 0.3, which is currently in development, will include

- Oersted field and current path modules:

  Developed by Benjamin Krüeger[60]. Additional extensions to the LLG equation will include the computation of current paths due to the AMR effect and the Oersted field due to electrical and spin currents.

- Application programming interface (API):

  The user API will be stabilized for version 0.3.

- Documentation:

  The documentation will be improved to that it covers the whole public user API.

Future plans beyond version 0.3 include the development of

- Improved solvers:

  Implicit solvers can potentially improve the overall speed of the simulation due to higher time steps due to their better ability to solve stiff differential equations like the LLGE compared to the currently used explicit schemes.

  Currently during a simulation only one instance of the Landau-Lifshitz-Gilbert equation is computed at any time. Parallel solvers evaluate many instances of the underlying ordinary differential equation at the same time in order to increase the parallelism. For example, the currently implemented classical Runge-Kutta-Fehlberg method could be replaced with a parallel iterated Runge-Kutta (PIRK) methods[127]. Here several Runge-Kutta steps can be computed in parallel, thus potentially increasing the overall performance of the simulator.

- More physical modules:

  The micromagnetic model can be further extended. Thermal effects can be simulated by the inclusion of a random fluctuating magnetic field that depends on the temperature. If currents are simulated, the temperature is influenced by Joule heating.

The magnetostriction effect[6] causes ferromagnetic materials to change their shape during their magnetization process, which results in energy loss in form of heat.

- Multi-GPU support:

  One of the most important feature for the acceptance of a micromagnetic simulator is its performance. In this work the performance was mainly achieved by parallel computing on a single graphics processing unit.

  One obvious future extension would be to allow the computation of one simulation on two or more GPUs in parallel. Currently the number of simulation cells and thus the size of the simulation volume are limited by the size of the GPU memory. By coupling multiple GPUs their combined memory could increase that limit. Additionally parallel GPUs could potentially lead to a higher overall performance. In the micromagnetic model, the most difficult part of multi-GPU parallelization is the computation of the convolution-based fields, i.e. the demagnetization field and the Oersted field. Their non-local nature would lead to large amounts of data that have to be communicated between the GPUs. In contrast, the local fields can be computed with a small constant amount of communication.

  The same considerations for multiple GPUs also apply for simulations on multiple CPUs, which might be distributed across a cluster of computers. Currently, the parallelism of a computer cluster can only be exploited by executing different independent simulations at the same time. Successfully parallelizing a single simulation across a computer cluster would be a challenge because of the increased bottleneck of the message passing system between the computer nodes. Also, achieving a latency and a data rate comparable to, for example, the PCI express bus between two graphics processors in one computer, might be prohibitively expensive.

- OpenCL support:

  Due to the use of the CUDA computing environment MicroMagnum only supports GPU hardware by Nvidia. The competing OpenCL[80] programming model is similar to that of CUDA, but is supported by more hardware architectures, including graphics processors manufactured by Nvidia and AMD. Thus adding OpenCL support would enable MicroMagnum to run on a wide range of GPU architectures. In addition, recent OpenCL implementations also support multicore CPUs. This might remove the need to implement each numerical algorithms for the CPU and the GPU altogether.

Acknowledgement

During this work I received support and encouragement from many people. In particular, I thank

- Professor Dr. Dietmar Möller for giving me the possibility to work on this project, his encouragement and enthusiasm, financial support, and for reviewing the thesis and defense.
- Professor Dr. Stephan Olbrich and
  Dr. habil. Guido Meier for kindly agreeing to review this thesis and the defense.
- Professor Dr. Ulrich Merkt, the speaker of the Graduiertenkolleg 1286, for giving me the possibility to work on this project.
- Dr. André Drews for being a great colleage and very good friend, his supervision of this work, and support.
- Dr. Claas Abert for the good collaboration on the potential field method and the many nice discussions in the office.
- Dr. Benjamin Krüger for the good collaboration on the Oersted field computation and the inclusion of periodic boundary conditions, and many inspiring discussions well past midnight at the office.
- Theo Gerhard for the good collaboration and his invaluable feedback from the user perspective.
- Bodo Krause-Kyora for keeping the whole computer cluster up and running, and many fun discussions about the latest on data center technology.
- Dr. Markus Bolte for his support, especially at the beginning of this project.
- Dr. Katrin Buth for her work for the Graduiertenkolleg.
- all members of the Graduiertenkolleg and the group Technische Informatiksysteme.
- the Deutsche Forschungsgemeinschaft for the financial support via the Graduiertenkolleg 1268.

Finally, I would like to thank my family and friends for all their invaluable support.

Appendix

A Listings

A.1 Functionality tests

This section contains the listings that implement the functionality tests described in chapter 7.

A.1.1 μMAG standard problem 1

```
from magnum import *
from math import sin, cos, pi

Py = Material.Py(Ms=8e5, A=1.3e-11,k_uniaxial=5e2, axis1=(0,1,0))

world = World(
  RectangularMesh((50,100,1), (20e-9, 20e-9, 20e-9)),
  Body("square", Py, Everywhere())
)

def hysteresis(name, axis):
  f = open("hysteresis-%s-axis.txt" % name, "w+")
  f.write("# H (mT)\t<mx> <my> <mz>\n")

  solver = create_solver(
    world, [StrayField, ExchangeField, AnisotropyField, ExternalField]
  )
  solver.state.M = (axis[0]*Py.Ms, axis[1]*Py.Ms, axis[2]*Py.Ms)

  for H in [x*1e-3/MU0 for x in range(50,-51,-1) + range(-50,51,1)]:
    Hx, Hy, Hz = H*axis[0], H*axis[1], H*axis[2]
    solver.state.H_ext_offs = (Hx, Hy, Hz)
    solver.relax(1.0)
    Mx, My, Mz = solver.state.M.average()
    f.write("%s\t%s %s %s\n" % (H*MU0, Mx/Py.Ms, My/Py.Ms, Mz/Py.Ms))

    if H == 0:
      writeOMF("M_remanence-%s-%s.omf" % (name, "up" if H > H_last else
          "down"), solver.state.M)
    H_last = H

  f.close()

hysteresis( "long", (sin(pi/180), cos(pi/180), 0.0))
hysteresis("short", (cos(pi/180), sin(pi/180), 0.0))
```

Listing 14: Simulation script for the μMAG standard problem 1 test.

A.1.2 μMAG standard problem 2

```
from magnum import *
import math

s = math.sqrt(3.0)

# Material parameters
Py = Material.Py()
A = Py.A
Ms = Py.Ms
K_m = 0.5*MU0*(Ms*Ms)
l_ex = math.sqrt(A/K_m)

# Geometry: ratio = d/l_ex
def geometry(ratio):
  d = ratio * l_ex
  t = 0.1 * d
  L = 5.0 * d
  return L, d, t

def discretize(L, d, t):
  nn = (10,10,10)
  dd = (1e-9,1e-9,1e-9)
  return RectangularMesh(nn, dd)

for ratio in (0.1,0.2):
  L, d, t = geometry(ratio)
  print "d/l_ex=%s, L=%s, d=%s, t=%s" % (ratio, L, d, t)

  mesh = discretize(L, d, t)
  world = World(mesh, Body("thinfilm", Material.Py(), Everywhere()))
  solver = create_solver(world, [StrayField, ExchangeField,
      ExternalField], log=True, do_precess=False)
  solver.state.M = (Ms,0.0,0.0)

  # do hysteresis
  H = 0.0/MU0

  while True:
    solver.state.H_ext_offs = (H/s,H/s,H/s)
    solver.relax()
    M = solver.state.M.average()

    print H*MU0, M[0]+M[1]+M[2]
    if M[0]+M[1]+M[2] == 0.0:
      H_coerc = H
      break
    H += 0.1/MU0

  while True:
```

```
49      solver.state.H_ext = (H/s,H/s,H/s)
50      solver.relax()
51
52      if H == 0.0:
53        M_rem = solver.state.M.average()
54        break
55      H -= 0.1/MU0
```

Listing 15: Simulation script for the μMAG standard problem 2 test.

A.1.3 μMAG standard problem 3

```
1  from magnum import *
2  import math
3
4  K_m = 0.5*MU0*Material.Py().Ms*Material.Py().Ms
5  Py = Material.Py(k_uniaxial=0.1*K_m,axis1=(0,0,1),alpha=0.5)
6  l_ex = math.sqrt(Py.A/K_m)
7
8  def sim(N, ratio):
9    def get_info(state):
10     E_str = state.E_str / K_m / L**3
11     E_ex  = state.E_ex  / K_m / L**3
12     E_ani = state.E_ani / K_m / L**3
13     E_tot = E_str + E_ex + E_ani
14     return E_tot, E_st, E_ex, E_ani, [x/Py.Ms for x in state.M.average
           ()]
15
16   def my_vortex(field, pos):
17     x, y, z = pos
18     Mx = 8e-9
19     My = -(z-0.5*L)
20     Mz = +(y-0.5*L)
21     scale = Py.Ms / math.sqrt(Mx**2 + My**2 + Mz**2)
22     return Mx*scale, My*scale, Mz*scale
23
24   L = ratio*l_ex # cube side length
25   world = World(RectangularMesh((N,N,N),(L/N,L/N,L/N)), Body("cube", Py
       ))
26   solver = create_solver(world,
27     [StrayField, ExchangeField, AnisotropyField],
28     log=True, do_precess=False)
29
30   solver.state.M = my_vortex
31   solver.relax(2)
32   writeOMF("omf/state-vortex-%s-%s.omf" % (N, ratio), solver.state.M)
33   vo = get_info(solver.state)
34
35   solver.state.M = (0, 0, Py.Ms)
36   solver.relax(2)
37   writeOMF("omf/state-flower-%s-%s.omf" % (N, ratio), solver.state.M)
38   fl = get_info(solver.state)
```

```
39
40    return vo, fl
41
42  def full_sp3():
43    for N in [12,16,20,24,28,32,36,40,44,48]:
44      for ratio in [r/100.0 for r in range(840, 860)]:
45        fl, vo = sim(N, ratio)
46        fl_E_tot, fl_E_str, fl_E_ex, fl_E_ani, (fl_mx, fl_my, fl_mz) = fl
47        vo_E_tot, vo_E_str, vo_E_ex, vo_E_ani, (vo_mx, vo_my, vo_mz) = vo
48
49        f = open("log-%s.txt" % N, "a")
50        f.write("ratio=%s, E_tot=(%s vs %s, diff=%s)\n" %
51          (ratio, fl_E_tot, vo_E_tot, abs(fl_E_tot - vo_E_tot)))
52        f.write(" * flower: %s\n" %
53          ((fl_E_tot, fl_E_str, fl_E_ex, fl_E_ani, (fl_mx,fl_my,fl_mz)),)
              )
54        f.write(" * vortex: %s\n" %
55          ((vo_E_tot, vo_E_str, vo_E_ex, vo_E_ani, (vo_mx,vo_my,vo_mz)),)
              )
56        f.close()
57
58  full_sp3()
```

Listing 16: Simulation script for the μMAG standard problem 3 test.

A.1.4 μMAG standard problem 4

```
1  from magnum import *
2  from math import sin, cos, pi
3
4  world = World(
5    RectangularMesh((100, 25, 1), (5e-9, 5e-9, 3.0e-9)),
6    Body("all", Material.Py(alpha=0.02))
7  )
8
9  # Create an s-state as the initial magnetization
10 def make_initial_sp4_state():
11   # Specify an s-state-like starting state
12   def state0(field, pos):
13     u = abs(pi*(pos[0]/field.mesh.size[0]-0.5)) / 2.0
14     return 8e5 * cos(u), 8e5 * sin(u), 0
15   # Relax to get initial state for SP4
16   solver = create_solver(world,
17     [StrayField, ExchangeField], log=True, do_precess=False,
18     evolver="rkf45", eps_abs=1e-4, eps_rel=1e-5
19   )
20   solver.state.M = state0
21   solver.state.alpha = 0.5
22   solver.relax(1.0)
23   return solver.state.M # return the final magnetization
24
25 # Apply an external field H on initial magnetization M0
```

```
def apply_field(M0, H, file_prefix):
  class ZeroCrossChecker(StepHandler):
    def __init__(self):
      self.crossed = False
    def handle(self, state):
      if not self.crossed and state.M.average()[0] < 0.0:
        print "First zero-crossing of <Mx> at", state.t*1e9, "ns!"
        writeOMF(file_prefix + "-Mx-zero.omf", state.M)
        self.crossed = True

  solver = create_solver(world,
    [StrayField, ExchangeField, ExternalField], log=True,
    evolver="rkf45", eps_abs=1e-4, eps_rel=1e-4
  )
  solver.state.M = M0
  solver.state.H_ext_offs = H
  solver.addStepHandler(ZeroCrossChecker(), condition.Always())
  solver.addStepHandler(DataTableLog(file_prefix + ".odt"), condition.
      EveryNthStep(10))
  solver.solve(condition.Time(1.0e-9))

# MAIN PROGRAM
M0 = make_initial_sp4_state()
apply_field(M0, (-24.6e-3/MU0, +4.3e-3/MU0, 0.0), "sp4-1")
apply_field(M0, (-35.5e-3/MU0, -6.3e-3/MU0, 0.0), "sp4-2")
```

Listing 17: Simulation script for the μMAG standard problem 4 test.

A.1.5 Spin-Torque standard problem

```
from magnum import *

world = World(
  RectangularMesh((100, 100, 1), (1e-9, 1e-9, 10e-9)),
  Body("all", Material.Py(xi=0.05, P=1.0, alpha=0.1))
)

# First step: Relax to get a vortex state
solver = create_solver(world, [StrayField, ExchangeField], log=True)
solver.state.M = vortex.magnetizationFunction(
  core_x = 50e-9, core_y = 50e-9,
  polarization = 1, core_radius = 10e-9
)
solver.state.alpha = 0.3
solver.relax()
vortex_M = solver.state.M

writeOMF("sp5-M0.omf", vortex_M) # save start configuration

# Second step: Apply DC to vortex
solver = create_solver(
  world,
```

```
  [StrayField, ExchangeField, SpinTorque, AlternatingCurrent], log=True
)

solver.state.M = vortex_M
solver.state.j_offs = (1e12, 0, 0)

# record vortex core position in .odt file "sp5.odt"
odt = DataTableLog("sp5.odt")
odt.addColumn(
  ("Vx", "Vortexcore X-pos.", "m"), ("Vy", "Vortexcore Y-pos.", "m"),
  lambda s: vortex.findCore(solver, 50e-9, 50e-9, "all")
)
solver.addStepHandler(odt, condition.EveryNthStep(100))

solver.solve(condition.Time(30.0e-9)) # start simulation

writeOMF("sp5-M_end.omf", solver.state.M) # save end configuration
```

Listing 18: Simulation script for the proposed μMAG standard problem 5 test.

A.1.6 Larmor precession test

```
from magnum import *

# Simple Larmor precession frequency test.
# Analytical frequency: MU0*H = 2.2102e11 rad/s
mesh = RectangularMesh((1,1,1), (1e-9, 1e-9, 1e-9))
mat  = Material.Py(alpha=0, Ms=1.0/MU0)
world = World(mesh, Body("all", mat, Everywhere()))

solver = create_solver(world, [ExternalField], log=True)
solver.state.M = (1,1,1)
solver.state.H_ext_offs = (0,0,1e6)
solver.addStepHandler(DataTableLog("larmor.odt"), condition.Always())
solver.solve(condition.Time(0.3e-9))
```

Listing 19: Simulation script for the Larmor precession test.

A.2 Sparse convolution subroutines

In this section the subroutines that are used to implement the sparse multidimensional transforms and convolutions (see Figs. 3.5 and 3.6). Here Python is used to specify the behavior of the subroutines. In MicroMagnum, they are implemented in C++ for CPUs and CUDA C for GPUs.

```
def zeropad(A, B, n_x, n_y, n_z, e_x):
  """
  Zeropad array A and store result in B.

  Input  'A': 3D array of size (n_x, n_y, n_z)
  Output 'B': 3D array of size (e_x, n_y, n_z)
  """
```

```
  for x in range(0, e_x):
    for y in range(0, n_y):
      for z in range(0, n_z):
        if x < n_x:
          B[x][y][z] = A[x][y][z]
        else:
          B[x][y][z] = 0.0
```

Listing 20: Algorithm: Array 3D zeropad.

```
def unpad(A, B, n_x, n_y, n_z, e_x):
  """
  Zeropad array A and store result in B.

  Input  'A': 3D array of size (e_x, n_y, n_z)
  Output 'B': 3D array of size (n_x, n_y, n_z)
  """
  for x in range(0, e_x):
    for y in range(0, n_y):
      for z in range(0, n_z):
        if x < n_x:
          B[x][y][z] = A[x][y][z]
        else:
          B[x][y][z] = 0.0
```

Listing 21: Algorithm: Array 3D unpad.

```
def rotate_and_zeropad(A, B, n_x, n_y, n_z, e_y):
  """
  Rotate 3D array from 'xyz' order to 'yzx' order while extending its
  size along the y-direction by zero-padding.

  Input  'A': 3D array of size (n_x, n_y, n_z)
  Output 'B': 3D array of size (e_y, n_x, n_z)
  """
  for y in range(0, e_y):
    for x in range(0, n_x):
      for z in range(0, n_z):
        if y < n_y:
          B[y][z][x] = A[x][y][z]
        else:
          B[y][z][x] = 0.0
```

Listing 22: Algorithm: Array 3D rotate/zeropad.

```
def rotate_and_cut(A, B, n_x, n_y, n_z, e_y):
  """
  Rotate 3D array from 'yzx' order to 'xyz' order while reducing its
  size along the y-direction.

  Input  'A': 3D array of size (n_x, e_y, n_z)
  Output 'B': 3D array of size (n_x, n_y, n_z)
  """
```

```
  for x in range(0, n_x):
    for y in range(0, e_y):
      for z in range(0, n_z):
        B[x][y][z] = A[y][z][x]
```

Listing 23: Algorithm: Array 3D rotate/cut.

```
def iterated_fft(A, B, n_x, n_y, n_z):
  """
  Perform (n_y*n_z) iterated fast Fourier transforms of size n_x along
  the x-direction.

  Input  'A': 3D array of size (n_x, n_y, n_z)
  Output 'B': 3D array of size (n_x, n_y, n_z)

  A and B may point to the same array.
  """
  tmp = [0.0] * n_x # temporary array of size n_x
  for y in range(0, n_y):
    for z in range(0, n_z):
      for x in range(0, n_x): tmp[x] = A[x][y][z]
      tmp = fft(tmp)
      for x in range(0, n_x): B[x][y][z] = tmp[x]
```

Listing 24: Algorithm: Iterated discrete Fourier transforms along the first dimension

```
def iterated_fft_r2c(A, B, n_x, n_y, n_z):
  """
  Perform (n_y*n_z) iterated real-to-complex Fourier transforms of size
      n_x along
  the x-direction.

  Input  'A': 3D real array of size (n_x, n_y, n_z)
  Output 'B': 3D complex array of size (round(n_x/2+1), n_y, n_z)
  """
  tmp = [0.0] * n_x # temporary array of size n_x
  for y in range(0, n_y):
    for z in range(0, n_z):
      for x in range(0, n_x): tmp[x] = A[x][y][z]
      tmp = fft(tmp)
      for x in range(0, round(n_x/2+1)): B[x][y][z] = tmp[x]
```

Listing 25: Algorithm: Iterated discrete real-to-complex Fourier transforms along the first dimension

```
def iterated_fft_c2r(A, B, n_x, n_y, n_z):
  """
  Perform (n_y*n_z) iterated complex-to-real Fourier transforms of size
      n_x along
  the x-direction.

  Input  'A': 3D complex array of size (round(n_x/2+1), n_y, n_z)
  Output 'B': 3D real array of size (n_x, n_y, n_z)
```

```
  """
  tmp = [0.0] * n_x # temporary array of size n_x
  l = round(n_x/2+1)

  for y in range(0, n_y):
    for z in range(0, n_z):
      tmp[0] = A[0][y][z]
      for x in range(0, l):
        tmp[ x+1] = A[x][y][z]
        tmp[-x-1] = A[x][y][z].conjugate()
      tmp = fft(tmp)
      for x in range(0, l): B[x][y][z] = tmp[x]
```

Listing 26: Algorithm: Iterated discrete real-to-complex Fourier transforms along the first dimension

```
def product(Axx, Axy, Axz, Ayy, Ayz, Azz, Bx, By, Bz, Cx, Cy, Cz, n_x,
    n_y, n_z):
  """
  Compute elementwise product of an array containing complex
  symmetric (3x3) matrices and and array containg complex (3)-vectors,
     e.g.

  / Cx \   / Axx Axy Axz \   / Bx \
  | Cy | = | Axy Ayy Ayz | * | By |   for all x,y,z.
  \ Cz /   \ Axz Ayz Azz /   \ Bz /

   Input 'Aij': 3d complex array of size (n_x, n_y, n_z)
   Input  'Bi': 3d complex array of size (n_x, n_y, n_z)
   Output 'Ci': 3d complex array of size (n_x, n_y, n_z)
  """
  for x in range(0, n_x):
    for y in range(0, n_y):
      for z in range(0, n_z):
        Cx[x][y][z] = Axx[x][y][z]*Bx[x][y][z] +
                      Axy[x][y][z]*By[x][y][z] +
                      Axz[x][y][z]*Bz[x][y][z]
        Cy[x][y][z] = Axy[x][y][z]*Bx[x][y][z] +
                      Ayy[x][y][z]*By[x][y][z] +
                      Ayz[x][y][z]*Bz[x][y][z]
        Cz[x][y][z] = Axz[x][y][z]*Bx[x][y][z] +
                      Ayz[x][y][z]*By[x][y][z] +
                      Azz[x][y][z]*Bz[x][y][z]
```

Listing 27: Algorithm: Elementwise product of an array containing complex symmetric (3x3) matrices and and array containg complex (3)-vectors.

References

[1] S. Bohlens *et al.*, Current controlled random-access memory based on magnetic vortex handedness, Appl. Phys. Lett. **93**, 142508 (2008).

[2] S. Tehrani *et al.*, Progress and Outlook for MRAM technology, IEEE Trans. Magn. **35**, 2814 (1999).

[3] S. S. P. Parkin, M. Hayashi, and L. Thomas, Magnetic Domain-Wall Racetrack Memory, Science **320**, 190 (2008).

[4] S. S. P. Parkin, U. S. Patent 7315470.

[5] A. Drews *et al.*, Current- and field-driven magnetic antivortices for nonvolatile data storage, Appl. Phys. Lett. **94**, 062504 (2009).

[6] A. Hubert and R. Schäfer, *Magnetic Domains - The Analysis of Magnetic Microstructures* (Springer, 1998).

[7] S. W. Yuan and H. N. Bertram, Fast adaptive algorithms for micromagnetics, IEEE Trans. Magn. **28**, 2031 (1992).

[8] N. Hayashi, K. Saito, and Y. Nakatani, Calculation of Demagnetizing Field Distribution Based on Fast Fourier Transform of Convolution, Jap. J. Appl. Phys. **35**, 6065 (1996).

[9] J. E. Miltat and M. J. Donahue, in *Handbook of Magnetism and Advanced Magnetic Materials*, edited by H. Kronmüller and S. S. P. Parkin (Wiley, 2007), Vol. 2, pp. 716–793.

[10] D. R. Fredkin and T. R. Koehler, Hybrid method for computing demagnetizing fields, IEEE Trans. Magn. **26**, 415 (1990).

[11] T. R. Koehler and D. R. Fredkin, Finite element methods for micromagnetics, IEEE Trans. Magn. **28**, 1239 (1992).

[12] J. Fidler and T. Schrefl, Micromagnetic modelling - the current state of the art, J. Phys. D: Appl. Phys. **33**, R135 (2000).

[13] T. Schrefl *et al.*, in *Handbook of Magnetism and Advanced Magnetic Materials*, edited by H. Kronmüller and S. S. P. Parkin (Wiley, 2007), Vol. 2, pp. 765–794.

[14] C. Seberino and H. N. Betram, Concise, efficient three-dimensional fast multipole method for micromagnetics, IEEE Trans. Magn. **37**, 1078 (2001).

[15] P. B. Visscher and D. M. Apalkov, Charge-based recursive fast-multipole micromagnetics, Physica B **343**, 184 (2004).

[16] A. A. Khan, P. Lugli, W. Porod, and G. Csaba, in *2010 14th International Workshop on Computational Electronics, IWCE* (IEEE, 2010), pp. 1–4.

[17] B. Van de Wiele, F. Olyslager, and L. Dupre, Application of the fast multipole method for the evaluation of magnetostatic fields in micromagnetic computations, J. Comput. Phys. **227**, 9913 (2008).

[18] C. Abert *et al.*, Numerical methods for the stray-field calculation: A comparison of recently developed algorithms, J. Magn. Magn. Mater. **326**, 176 (2013).

[19] M. J. Donahue and D. G. Porter, Object Oriented Micromagnetic Framework, 1999, Interagency Report NISTIR 6376.

[20] K. M. Tako, T. Schrefl, M. A. Wongsam, and R. W. Chantrell, Finite element micromagnetic simulations with adaptive mesh refinement, J. Appl. Phys. **81**, 4082 (1997).

[21] T. Fischbacher, M. Franchin, G. Bordignon, and H. Fangohr, A Systematic Approach to Multiphysics Extensions of Finite-Element-Based Micromagnetic Simulations: Nmag, IEEE Trans. Magn. **43**, 2896 (2007).

[22] T. Fischbacher *et al.*, Parallel execution and scriptability in micromagnetic simulations, J. Appl. Phys. **105**, 07D527 (2009).

[23] W. Scholz *et al.*, Scalable parallel micromagnetic solvers for magnetic nanostructures, Comp. Mater. Sci. **28**, 366 (2003).

[24] L. Lopez-Diaz *et al.*, Micromagnetic simulations using Graphics Processing Units, J. Phys. D: Appl. Phys. **45**, 323001 (2012).

[25] M. Najafi, Micromagnetic Modeling by Computational Science Integrated Development Environments (CSIDE), Dissertation, Hamburg University, 2011, `http://ediss.sub.uni-hamburg.de/volltexte/2011/5388/`.

[26] A. Vansteenkiste and B. V. de Wiele, MuMax: A new high-performance micromagnetic simulation tool, J. Magn. Magn. Mater. **323**, 2585 (2011).

[27] P. Visscher and D. Apalkov, Simple recursive implementation of fast multipole method, J. Magn. Magn. Mater. **322**, 275 (2010).

[28] R. Chang *et al.*, FastMag: Fast micromagnetic simulator for complex magnetic structures, J. Appl. Phys. **109**, 07D358 (2011).

[29] GNU General Public License, 2013, `http://www.gnu.org/licenses/gpl.html`.

[30] A. Kakay, E. Westphal, and R. Hertel, Speedup of FEM Micromagnetic Simulations With Graphical Processing Units, IEEE Trans. Magn. **46**, 2303 (2010).

[31] S. Li, B. Livshitz, and V. Lomakin, Graphics Processing Unit Accelerated O(N) Micromagnetic Solver, IEEE Trans. Magn. **46**, 2373 (2010).

[32] MicroMagnum website, `http://micromagnum.informatik.uni-hamburg.de`.

[33] MicroMagnum code repository, `http://github.com/MicroMagnum/MicroMagnum`.

[34] B. Stroustrup, *The C++ Programming Language: 4th ed.* (Addison-Wesley, 2013).

[35] CUDA Programming Guide, 2013, `http://docs.nvidia.com/cuda/`.

[36] G. van Rossum, *The Python Language Reference Manual* (Network Theory, 2003).

[37] A. Drews *et al.*, Nonlinear magnetic vortex gyration, Phys. Rev. B **85**, 144417 (2012).

[38] W. Heisenberg, Zur Theorie des Ferromagnetismus, Z. Phys. **49**, 619 (1928).

[39] L. Landau and E. Lifshitz, On the theory of the dispersion of magnetic permeability in ferromagnetic bodies, Phys. Z. Sow. **8**, 153 (1935).

[40] F. Bloch, Zur Theorie des Austauschproblems und der Remanenzerscheinung der Ferromagnetika, Z. Phys. **74**, 295 (1932).

[41] T. Shinjo *et al.*, Magnetic Vortex Core Observation in Circular Dots of Permalloy, Science **289**, 930 (2000).

[42] A. Wachowiak *et al.*, Direct Observation of Internal Spin Structure of Magnetic Vortex Cores, Science **298**, 577 (2002).

[43] W. F. Brown, *Micromagnetics* (Wiley, 1965).

[44] T. L. Gilbert, A Lagrangian formulation of gyromagnetic equation of the magnetization field, Phys. Rev. **100**, 1243 (1955).

[45] T. L. Gilbert, A phenomenological theory of damping in ferromagnetic materials, IEEE Trans. Magn. **40**, 3443 (2004).

[46] J. D. Jackson and R. F. Fox, *Classical Electrodynamics, 3rd ed.* (Wiley, 1998).

[47] L. Berger, Emission of spin waves by a magnetic multilayer traversed by a current, Phys. Rev. B **54**, 9353 (1996).

[48] S. F. Zhang and S. S. L. Zhang, Generalization of the Landau-Lifshitz-Gilbert Equation for Conducting Ferromagnets, Phys. Rev. Lett. **102**, 086601 (2009).

[49] Z. Li and S. Zhang, Domain-wall dynamics driven by adiabatic spin-transfer torques, Phys. Rev. B **70**, 024417 (2004).

[50] J. C. Slonczewski, Current-driven excitation of magnetic multilayers, J. Magn. Magn. Mater. **159**, L1 (1996).

[51] S. Bohlens, Interplay of Inhomogeneous Currents and Magnetization Textures, Dissertation, Hamburg University, 2011,
`http://ediss.sub.uni-hamburg.de/volltexte/2011/5054/`.

[52] Y. Nakatani and T. Ono, Effect of the Oersted field on a vortex core switching by pulse spin current, Appl. Phys. Lett. **99**, 122509 (2011).

[53] M. J. Donahue and R. D. McMichael, Exchange energy representations in computational micromagnetics, Physica B: Condensed Matter **233**, 272 (1997).

[54] A. J. Newell, W. Williams, and D. J. Dunlop, A Generalization of the Demagnetizing Tensor for Nonuniform Magnetization, J. Geophys. Res. **98**, 9551 (1993).

[55] M. Schabes and A. Aharoni, Magnetostatic interaction fields for a three-dimensional array of ferromagnetic cubes, IEEE Trans. Magn. **23**, 3882 (1987).

[56] M. Maicus *et al.*, Magnetostatic energy calculations in two- and three-dimensional arrays of ferromagnetic prisms, IEEE Trans. Magn. **34**, 601 (1998).

[57] B. Krüger, G. Selke, A. Drews, and D. Pfannkuche, Fast and Accurate Calculation of the Demagnetization Tensor for Systems with Periodic Boundary Conditions, IEEE Trans. Magn. **49**, 4749 (2013).

[58] C. Abert, G. Selke, B. Krüger, and A. Drews, A Fast Finite-Difference Method for Micromagnetics Using the Magnetic Scalar Potential, IEEE Trans. Magn. **48**, 1105 (2012).

[59] D. V. Berkov, K. Ramstöcck, and A. Hubert, Solving Micromagnetic Problems. Towards an Optimal Numerical Method, Phys. Status Solidi A **137**, 207 (1993).

[60] B. Krüger *et al.*, Fast and accurate calculation of demagnetization and Oersted fields (working title) , in preparation.

[61] B. Krüger *et al.*, Fast dynamic current-path computation for micromagnetic simulation (working title) , in preparation.

[62] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations* (Wiley, 1963).

[63] E. Fehlberg, Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems, 1969, NASA technical report 315.

[64] J. R. Dormand and P. J. Prince, A family of embedded Runge-Kutta formulae, J. Comput. Appl. Math. **6**, 19 (1980).

[65] J. R. Cash and A. H. Karp, A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides, ACM Trans. Math. Softw. **16**, 201 (1990).

[66] J. W. Cooley and J. W. Tukey, An algorithm for the machine computation of the complex Fourier series, Math. Comp. **19**, 297 (1965).

[67] M. Najafi *et al.*, in *Proc. of the 2008 Grand Challenges in Modeling and Simulation Conference* (Society for Modeling and Simulation, 2008), pp. 427–434.

[68] C. M. Rader, Discrete Fourier transforms when the number of data samples is prime, Proc. IEEE **56**, 1107 (1968).

[69] L. Bluestein, A linear filtering approach to the computation of discrete Fourier transform, IEEE T. Acoust. Speech **18**, 451 (1970).

[70] M. Frigo and S. G. Johnson, The Design and Implementation of FFTW3, Proc. IEEE **93**, 216 (2005).

[71] G. Selke, Optimierung und Parallelisierung der Berechnung des Demagnetisierungsfelds im mikromagnetischen Simulator M3S, Diplomarbeit, Hamburg University, 2008.

[72] M. Frigo and S. G. Johnson, FFTW: an adaptive software architecture for the FFT, Int. Conf. Acoust. Spee. **3**, 1381 (1998).

[73] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, in *Ann. IEEE Symp. Found. Comp. Sci.* (IEEE Computer Society, 1999), p. 285.

[74] M. Frigo, A fast Fourier transform compiler, ACM SIGPLAN Not. **34**, 169 (1999).

[75] M. Frigo and S. G. Johnson, A Modified Split-Radix FFT With Fewer Arithmetic Operations, IEEE T. Signal Proces. **55**, 111 (2007).

[76] T. Matsuo and Y. Yamazaki, Demagnetizing Field in Micromagnetic Simulation Under Periodic Boundary Conditions, IEEE Trans. Magn. **47**, 902 (2011).

[77] K. M. Lebecki, M. D. Donahue, and M. W. Gutowski, Periodic boundary conditions for demagnetization interactions in micromagnetic simulations, J. Phys. D: Appl. Phys. **41**, 175005 (2008).

[78] L. Dagum and R. Menon, OpenMP: An industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. **5**, 46 (1998).

[79] Portable Operating System Interface (POSIX), 2008, IEEE Std 1004.1-2008.

[80] A. Munshi, The OpenCL Specification, 2009, technical report.

[81] MPI: A Message-Passing Interface Standard Version 3.0, 2012, `http://www.mpi-forum.org/docs/docs.html`.

[82] A. Grama, *Introduction to Parallel Computing, 2nd Ed.* (Pearson Education, 2003).

[83] K. Asanovic *et al.*, The Landscape of Parallel Computing Research: A View from Berkeley, 2006, technical report.

[84] R. J. Rost, *OpenGL Shading Language* (Addison-Wesley, 2004).

[85] F. Luna, *Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach* (Wordware Publishing, Inc., 2006).

[86] CUBLAS library reference manual, 2013, `http://docs.nvidia.com/cuda/`.

[87] J. Dongarra, Basic Linear Algebra Subprograms Technical Forum Standard, Int. J. High Perform. Comput. Appl. **16**, 1 (2002).

[88] CUFFT library reference manual, 2013, `http://docs.nvidia.com/cuda/`.

[89] M. Garland *et al.*, Parallel Computing Experiences with CUDA, IEEE Micro **28**, 13 (2008).

[90] M. J. Flynn, Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput. **C-21**, 948 (1972).

[91] S. Che *et al.*, in *Symposium on Application Specific Processors, 2008.* (IEEE, 2008), pp. 101–107.

[92] A. Nukada and S. Matsuoka, in *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09* (ACM, 2009), pp. 30:1–30:10.

[93] N. K. Govindaraju and D. Manocha, Cache-efficient numerical algorithms using graphics hardware, Parallel Comput. **33**, 663 (2007).

[94] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, in *Proc. of the 4th international workshop on End-user software engineering, WEUSE '08* (ACM, 2008), pp. 1–5.

[95] B. Kitchenham and S. L. Pfleeger, Software quality: the elusive target, IEEE Software **13**, 12 (1996).

[96] B. W. Boehm, J. R. Brown, and M. Lipow, in *Proc. of the 2nd International Conference on Software Engineering, ICSE '76* (IEEE Computer Society Press, 1976), pp. 592–605.

[97] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language* (Addison-Wesley, 2006).

[98] MATLAB Version 7.10.0 (R2010a), 2010, `http://www.mathworks.com/`.

[99] T. E. Oliphant, *A Guide to NumPy* (Trelgol Publishing, 2006).

[100] E. Jones *et al.*, SciPy: Open source scientific tools for Python, 2013, `http://www.scipy.org/`.

[101] J. K. Ousterhout, Scripting: higher level programming for the 21st Century, Computer **31**, 23 (1998).

[102] P. H. Langtangen, in *Python Scripting for Computational Science*, Vol. 3 of *Texts in Computational Science and Engineering*, edited by T. J. Barth *et al.* (Springer, 2008), pp. 189–226.

[103] F. Perez and B. E. Granger, IPython: A System for Interactive Scientific Computing, Comput. Sci. Eng. **9**, 21 (2007).

[104] S. Meyers, *Effective C++ : 55 Specific Ways to Improve your Programs and Designs* (Addison-Wesley, 2005).

[105] D. M. Beazley, Automated scientific software scripting with SWIG, Future Gener. Comput. Syst. **19**, 599 (2003).

[106] D. Abrahams, in *Generic Programming*, Vol. 1766 of *Lecture Notes in Computer Science*, edited by M. Jazayeri, R. Loos, and D. Musser (Springer, 2000), pp. 69–79.

[107] S. Ghali, *Introduction to Geometric Computing* (Springer, 2008).

[108] Visualization Toolkit (VTK), 2013, `http://www.vtk.org`.

[109] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing* (Wiley, 2011).

[110] Micromagnetic Modeling Activity Group website, `http://www.ctcms.nist.gov/rdm/mumag.org.html`.

[111] M. Najafi *et al.*, Proposal for a standard problem for micromagnetic simulations including spin-transfer torque, J. Appl. Phys. **105**, 113914 (2009).

[112] G. Venkat *et al.*, Proposal for a standard micromagnetic problem: Spin wave dispersion in a magnonic waveguide, IEEE Trans. Magn. **49**, 524 (2013).

[113] R. Hertel and H. Kronmüller, Finite element calculations on the single-domain limit of a ferromagnetic cube—a solution to muMAG Standard Problem No. 3, J. Magn. Magn. Mater. **238**, 185 (2002).

[114] X. Cai, H. P. Langtangen, and H. Moe, On the performance of the Python programming language for serial and parallel scientific computations, Scientific Programming **13**, 31 (2005).

[115] S. Masini and P. Bientinesi, in *Euro-Par 2010 Parallel Processing Workshops* (Springer, 2011), pp. 541–548.

[116] CUDA C Best Practices Guide, 2013, `http://docs.nvidia.com/cuda/`.

[117] M. J. Donahue, Parallelizing a Micromagnetic Program for Use on Multiprocessor Shared Memory Computers, IEEE Trans. Magn. **45**, 3923 (2009).

[118] Y. Kanai *et al.*, Micromagnetic Analysis of Shielded Write Heads Using Symmetric Multiprocessing Systems, IEEE Trans. Magn. **46**, 3337 (2010).

[119] K. Y. Guslienko, R. H. Heredero, and O. Chubykalo-Fesenko, Nonlinear gyrotropic vortex dynamics in ferromagnetic dots, Phys. Rev. B **82**, 014402 (2010).

[120] B. Krüger, Current-Driven Magnetization Dynamics: Analytical Modeling and Numerical Simulation, Dissertation, Hamburg University, 2011, `http://ediss.sub.uni-hamburg.de/volltexte/2012/5887/`.

[121] A. A. Thiele, Steady-State Motion of Magnetic Domains, Phys. Rev. Lett. **30**, 230 (1973).

[122] A. Vogel *et al.*, Vortex dynamics in triangular-shaped confining potentials, J. Appl. Phys. **112**, 063916 (2012).

[123] A. Vogel, A. Drews, M. Weigand, and G. Meier, Direct imaging of phase relation in a pair of coupled vortex oscillators, AIP Advances **2**, 042180 (2012).

[124] J. Kimling *et al.*, Tuning of the nucleation field in nanowires with perpendicular magnetic anisotropy, J. Appl. Phys. **113**, 163902 (2013).

[125] A. Drews, G. Selke, and D. P. F. Möller, in *Proc. of the 2010 Conference on Grand Challenges in Modeling and Simulation* (Society for Modeling and Simulation, 2010), pp. 152–157.

[126] D. P. F. Möller, A. Drews, and G. Selke, in *Proc. of the AlaSim 2012 Conference, Huntsville, AL.* (Society for Modeling and Simulation, 2012).

[127] P. J. van der Houwen and B. P. Sommeijer, Parallel iteration of high-order Runge-Kutta methods with stepsize control, J. Comput. Appl. Math. **29**, 111 (1990).