Georg Brantegger, BSc

Python-based Analytical Computation of the Magnetic Field of Hard Magnetic Facet Bodies

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Technical Physics

submitted to

Graz University of Technology

Supervisor

Markus Aichhorn, Assoc.Prof. Dr.techn.

Institute of Theoretical and Computational Physics

In collaboration with Silicon Austria Labs GmbH

Supervisor: Michael Ortner, Dr.rer.nat

Graz, December 2020

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Thanks and Acknowledgements

I want to thank my friends and family, my colleagues and companions and all the people who shared part of my journey with me. Everyone of these people shaped me and I am grateful for that.

I want to especially thank my parents, for providing the foundation, upon which I am building my life. I want to thank Anna, for keeping me open-minded, for reminding me every day, what difference even a single smile makes and for fighting fiercely to keep boredom away from us.

I also want to thank my friends Alex, Jakob, Matthias, Simon and Stefan for their company in early-morning classes, for their last minute help with exercises and for our *Semester-Kick-Off* dinners.

Furthermore I want to thank TU Graz for being a great place to study. It was a great pleasure for me to learn from my supervisor Markus Aichhorn in lectures and exercise courses, and I am grateful for his assistance and input during authoring this thesis.

At last I want to thank Silicon Austria Labs for giving me the opportunity to learn from experienced researchers. Thank you, Michael Ortner, for guiding me on the journey towards this thesis! Also great thanks to Christina, Lukas and Wolfgang for sharing their knowledge with me during five exciting internships.

This thesis has been supported by the COMET K1 centre ASSIC Austrian Smart Systems Integration Research Center. The COMET (Competence Centers for Excellent Technologies) Program is supported by BMK, BMDW, and the federal provinces of Carinthia and Styria.

Abstract

In this work Python code will be introduced, that allows the calculation of the magnetic field of facet bodies, based on analytical formulas. Chapter 1 gives a short introduction into magnetic system design and computational magnetism. These two topics form the context of this thesis and provide insights into the motivation behind this work. The theoretical part of this thesis in chapter 2 treats relevant aspects of computer science and efficient programming and discusses the theory of magnetism. Section 2.1 delves into the inner workings of a computer and, in addition to that, introduces the fundamentals of high-performance programming in the context of Python. The part on the theory of magnetism, discussed in section 2.2, showcases the derivation of the scalar magnetic potential of a triangular facet from the underlying Maxwell equations. With that knowledge the theoretical path from the magnetic scalar potential of a facet towards the magnetic field of a facet body is outlined in section 2.3. Chapter 4 presents the validation of the proposed code. This validation includes the comparison of the results of the field calculation to another already published analytical solution and to a state of the art finite element method simulation. These results are analysed in terms of accuracy and beyond that the performance of the different methods s discussed. Following the validation, in chapter 5 a review of the findings and a conclusion is provided.

Kurzfassung

Diese Arbeit stellt Python Code vor, der in der Lage ist das Magnetfeld eines Facettenkörpers anhand analytischer Formeln zu berechnen. In Kapitel 1 werden die Begriffe „magnetic system design“ und „computational magnetism“ eingeführt. In diesem Kontext wird die Motivation hinter dieser Arbeit vorgestellt. Der theoretische Teil der Arbiet beleuchtet Grundsätze der Computerwissenschaft und des Magnetismus. In Abschnitt 2.1 wird die Funktionsweise moderner Computer besprochen und es wird auf die Grundlagen der Programmierung in Python eingegangen. Im Abschnitt über die Theorie des Magnetismus 2.2 wird die Herleitung des skalaren magnetischen Potentials einer dreieckigen Facette von den zugrundeliegenden Maxwell Gleichungen besprochen. Ausgehend von diesem skalaren Potential wird im Abschnitt 2.3 gezeigt, wie man das Magnetfeld eines Facettenkörpers berechnet. Das Kapitel 4 behandelt die Validierung des vorgestellen Codes. Diese beinhalted den Vergleich der Ergebnisse der Magnetfeldberechnung mit mehreren Methoden. Zum einen werden die Ergebnisse mit einer anderen, bereits publizierten analytischen Lösung verglichen. Zum anderen findet ein Vergleich mit einer state-of-the-art Finite-Elemente-Methode Simulation statt. Die Ergebnisse werden unter den Aspekten der Genauigkeit und Geschwindigkeit diskutiert. Auf die Validierung folgend werden im Kapitel 5 die Erkenntnisse dieser Arbeit besprochen und eine Conclusio gefasst.

Contents

List of Figures

1. Motivation

This first chapter gives an introduction into *magnetic system design* and computational magnetism. It will provide context and motivation for this work by discussing the applications of magnetic systems and how research in this area benefits from the use of fast analytical formulas. Furthermore, the appeal of facet bodies in the context of magnetic systems will be discussed, and the motivation behind the Magpylib package, to which this work presents an extension, will be introduced.

1.1. Magnetic System Design

The design of magnetic systems is an important topic in many industries and applications [1, p. 201ff]. A magnetic system consists of magnetic sources and sensors, positioned in a way that enables the measurement of diverse system properties [2, 3, 4]. The determination of position, orientation and rotation are prime examples of industry applications of magnetic sensor systems [5, p. 27ff]. The appeal of using magnetic materials in different sensor applications originates from distinct properties of the magnetic field itself. The first is that magnetic field sensing is not reliant on physical contact. This is an advantage in many mechanical setups, since contactless measurement means that there is no mechanical wear, neither of the magnet source nor of the sensor, which enables long lifetimes [6]. The second defining characteristic of the magnetic field, which gives magnetic sensors an edge over optical measurement approaches for example, is that the magnetic field permeates many materials. For that reason, contaminants like

dust or oil, have a negligible impact on the measurement, which lessens the need for environmental contamination control [7]. This robustness against environmental factors make magnetic sensor systems cheaper and easier to realize, and therefor more attractive for industrial applications, all while still enabling high resolution sensing [8].

1.2. Computational Magnetism

Having listed the benefits and some application of magnetic systems, this short section is dedicated to the introduction of computational magnetism. Computational magnetism investigates ways to simulate magnetic fields and materials, using the power of modern computers and algorithms to deepen the understanding of magnetic systems and aid magnetic system design [9]. There are two main approaches to the computer based calculation of the magnetic field. The first one is the numerical simulation of magnetic systems via the *finite element method*, or FEM. While being the most versatile option, the nature of the FEM, which will be discussed in more detail in section 4.3.1, leads to long computation times. In cases where one wants to understand a given system and the complex interactions like demagnetization or electromagnetic interference in it, the FEM approach is the method of choice. In such cases, where the design of the magnetic system is already given, the longer computation time is not of big concern. The second approach, which is pursued in this thesis, is to create a simplified model of the magnetic system by disregarding demagnetization phenomenons and by only treating homogeneous magnetizations. By doing so, an approximation of the real setup is derived, which is describable by analytical formulas. The use of analytical formulas in contrast to the numerical approaches of the FEM enables rapid computation of the magnetic field. With very small computation times for the fields of these idealized systems, many new application are possible. One of these applications is the optimization of such magnetic systems. Optimization in this context targets the measurability of

the magnetic field and the detectability of changes to the magnetic field, which can be used to extract geometric parameters of the system. One example where such optimizations were employed is this paper on the improvement of 1D linear position measurements by *field shaping* [10]. Another real world magnetic sensing applications is tracking the motion of a 3-axis joystick [11]. In this case the manufacturing tolerances of the components involved in the analyzed 3-axis joystick give rise to a multi-variable optimization problem. Such a problem is only treatable in a reasonable time frame, if the field evaluating functions can be computed very quickly. With high-performance functions to do so, complex optimization problems can be solved and new applications of magnetic systems can be developed.

1.3. Facet Bodies

Before explaining the motivation behind the treatment of facet bodies, the term facet body has to be defined. A facet body, in the context of this thesis, is a three dimensional body, whose surface is describable by triangular facets. This work introduces Python code, that enables the calculation of the magnetic field of such bodies, and treats the assumptions these calculations are based on. The extension of analytical methods towards more complex shapes will come at a time when magnet forms beyond simple geometries become more common. The technique of additive manufacturing for example, gives the possibility to create magnetic materials, customized to a specific problem [12, 13]. Combined with the potential to rapidly calculate the magnetic field of such a magnet, even before production, optimization of the magnet geometry is enabled.

1.4. Magpylib

Magpylib [14] is a

> Free Python package for calculating magnetic fields of magnets, currents and moments (sources), which provides convenient methods to create, geometrically manipulate, group and visualize assemblies of [magnetic] sources. [15]

It was developed by Michael Ortner, one of the supervisors of this thesis, with the goal to make computational magnetism more accessible by providing an easy-to-use interface to build and compute magnetic systems. Part of the considerations on accessibility are the platforms, on which Magpylib is available and the amount of prior knowledge the end user needs. For that reason Python, which is wide spread across the scientific community, was chosen as a programming language and by only using NumPy [16] beyond standard Python, compatibility with most operating systems is ensured. Another founding principle of Magpylib is the focus on the performance of the implemented methods. Therefore, the calculation of the magnetic field for every implemented magnetic source is based on analytical formulas.
The targeted applications of Magpylib are twofold. On the one hand, the easy-to-use interface enables utilization in an educational setup. The possibility that "with Magpylib, the field is only three lines of code away" [14] provides a quick and easy way to visualize magnetic fields and therefor help the understanding of this topic by students. On the other hand, the foundation on high-performance analytical formulas enables research applications, that are not possible with standard FEM approaches.

2. Theory

After learning about the motivation behind the Magpylib project and what benefits the proposed facet solution promises, this chapter treats the theoretical basis for the practical implementation. Therefore, a brief detour into computer science is taken, during which the basics for high-performance computation in Python are explored. In this context the benefits of the NumPy library will be discussed too. Furthermore, the fundamentals of the theory of magnetism are revisited, before the specific theory used to calculate the magnetic field of a right triangle with special positioning is introduced. The last part of this chapter deals with the formalism that enables the calculation of the magnetic field of a general triangle.

2.1. Computer Science Basics

In this section the subset of computer science basics which is relevant for this thesis will be discussed. The scope of this discussion is limited to the introduction of concepts necessary for understanding performance oriented computing in general and in Python especially. Therefore, in a first step the performance critical hardware components of a computer are highlighted. In addition to that, a closer look at how a computer handles instructions and data is taken. The final section will treat good software development practices and following general valid advise Python specific aspects will be discussed.

2.1.1. How a Computer Works

In order to understand what it takes to achieve high-performance computation, it is necessary to understand how a computer works on a basic level. The following paragraphs will introduce the abstract structure of a computer and thereafter add details on the individual components. For further information on different aspects of the hardware structure of a computer the reader is referred to a selection of excellent textbooks on that matter [17, 18, 19].

Prior to any discussion of a modern computer, its basic structure has to be established. In the most general sense a computer consists of a processor capable of performing mathematical operations, a memory system responsible for storing data, and an interface connecting the processor and the memory [20]. To explore the interplay between these components it is advisable to follow the path of data. When a computer is instructed to execute an operation, it has to perform multiple steps. First it has to decode the request, which involves translating the desired operation into machine code[1], and potentially deconstructing complex operations into chains of simpler ones. Then the processor needs data to work on. Therefore, it sends a request for data to the memory system, which subsequently routes this data through possibly multiple layers of memory and interfaces to the processor. With both the instructions and the data loaded, the processor can operate on the data and write the final result back through the interfaces to memory. The performance of such a system depends on multiple factors. Foremost, in an ideal system with an infinitely fast memory system the maximum achievable performance is still limited by the capabilities of the processor. This, while sounding obvious, has to be kept in mind, especially when comparing the performance between different systems. In a real world scenario the memory system will not work perfectly, i.e. infinitely fast, which gives rise to a bottleneck: keeping the processor fed with data to work on. This complication is known as the von-Neumann bottleneck or *memory*

[1] the language the processor understands

wall [21]. Mitigating the effects of this bottleneck is a multifaceted problem, where beyond technical hardware aspects and programming practices also economic constraints in the semiconductor industry are to be respected [17, ch. 2.1]. The important hardware aspects are memory speed, memory size, interface width, latency, and in some sense space constraints. On the software side efficient programming practices are expected - which will be treated in section 2.1.2. The compromises on the memory side can be illustrated by the *memory hierarchy* [18, p. 4], [17, App. B]. It is depicted in figure 2.1 and lists various types of memory, annotated with their relative cost and performance. In general, the higher up the hierarchy a storage type is, the faster but also the more expensive it is.

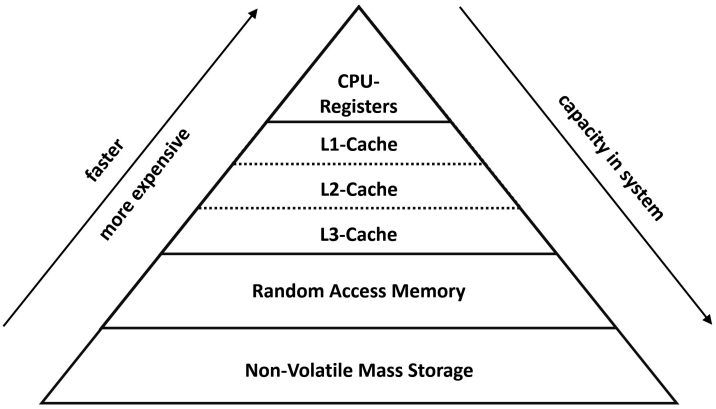


Figure 2.1.: Schematic of the storage hierarchy, annotated with relative performance, price and capacity built into a system.

On the top of the memory hierarchy is a memory type called *processor registers*. This is the fastest type of memory built into a computer and it is directly connected to the processor. The small physical distance between the memory and the processor enables a broad interface and minimizes latency, i.e. the delay between the processor requesting data and receiving it. Furthermore data stored in registers can be directly accessed by the processor, without going through another layer of the memory system [19, ch. 2].

Below the registers in the hierarchy, there is the *processor cache*. The cache, like the registers, is built onto the CPU die but further away from the processing units. Although it is slower than the registers, it is still a very fast type of memory. Processor cache memory in itself is tiered, meaning there are different *cache levels* with increasing size but decreasing speed. In modern CPUs there are usually two to three levels of cache, with level 1 being the highest up in the hierarchy, i.e. closest to the CPU.
The next step down from the processor cache is the *dynamic Random Access Memory*, often abbreviated by only RAM. RAM is not integrated into the CPU, which entails an increase in latency and additionally it is much slower than either of the on-die memory types. Its benefit is the relatively low cost and the capacity in which it is available. Often RAM is referred to as the systems *main memory*, since every program that is running, is stored in RAM.
On the levels below RAM, there are non-volatile mass-storage types, like hard disks or SSDs[2], which are of no interest for this thesis. For a more extensive treatment of computer memory Jacob, Ng and Wang's *Memory Systems* [18] is recommended.
An overarching theme when going up the memory hierarchy is the increase in performance but decrease in memory capacity that is actually built into a system. This is the result of a trade-off between performance and economic cost. For maximum performance the whole storage infrastructure of a computer would consist of register- or cache-like memory. The higher production expense of high-performance memory and the increase in CPU-chip size would lead to an enormous cost increase. Furthermore, as stated above, the computational performance of a system is not only dependent on the memory system but also on the capability of the processor. A storage system that provides data to the processor at a higher rate than the processor can handle is economically inefficient. Therefore, only the amount of any memory type is built into a system that is necessary to achieve the required performance. This especially holds true for registers and cache, since the amount of these memory types is not controllable by the end-user. In ad-

[2]Solid State Drive

dition to the structure of the storage hierarchy, it is important to know that data which is called by the processor, is copied from main memory through the different levels of cache to the registers [18, p. 63]. This is relevant for one principle of high performance computing that is going to be introduced in the following.

The *principle of locality* is an important topic in high performance computing, and is extensively exploited in this thesis. It states that a computer performs better, when data is called that has been called recently or that lies next to some data that has been processed lately [18, p. 63 f.]. The first part is referred to as *temporal locality* and is intuitive: data that is still located in a lower level memory[3] is accessed faster than data that has to be fetched from main memory, due to latency and bandwidth limitations. The second one, called *spatial locality*, is a side effect of the way memory is organized and how systems load data. Data is stored in fixed size pieces of multiple bits called *words*, and the cache is organized into also fixed sized patches of storage cells called *cache lines*. Usually cache lines are larger than a single word and every time new data is copied from RAM into cache, a full cache line is copied. This means that besides the data specifically requested, also neighboring data is copied into the faster cache. Therefore, when this neighbouring data is requested later, it already lies in cache and is quickly available to the processor. Especially the spatial locality is important for this work and will be addressed again, when talking about the physical location of data stored in the basic NumPy object, the *nd-array*.

Having discussed the way data gets to the processor, the next step is to shed light on the way the CPU treats that data. Since this is a vast topic, only the concepts directly connected to this thesis will be mentioned. One of the basic terms needed for this discussion is an *instruction*. It instructs the processor to perform particular operation(s) on specific data. In the context of this work, such operations can be memory related, like setting a register to a specific value, or an arithmetic or logic operation. Beyond that

[3] closer to the processor

there are various other instructions, enabling diverse functionalities of a processor, but too extensive to be detailed here. An in depth discussion of instructions on Intel processors, accompanied by introductory details, can be found in [22]. All instructions a processor can execute are combined in its *instruction set* [22, ch. 3]. There are two philosophies regarding processor design and size of an instruction set. On the one hand there are *CISC* - or complex instruction set computer - architectures. These processors include many highly specialized instructions, accepting that these are used less often. On the other hand there are *RISC*, or reduced instruction set computer, architectures. These only support a limited set of more simple instructions, and achieve the functionality of the specialized instructions of CISC systems by smartly combining multiple simpler but faster operations. Modern processors are not purely CISC or RISC based, but implement ideas from both philosophies. An example for such processors is the *Woodcrest* series by Intel [23]. Furthermore, the supported instruction set of a new generation of processors differs from its predecessor. This generational evolution and sometimes expansion is driven by new demands in applications and improvements in processor design.

One notable extension to the instruction set of modern processors was the introduction of SIMD-operations. SIMD as defined in *Flynn's taxonomy* [17, ch. 1.1] stands for *single instruction multiple data*. This describes a processor's ability to apply one operation not only to one data word but to several data words at once. Processors supporting SIMD instruction have specialized registers, capable of storing arrays of data. SIMD instructions enable improved performance for many operations. Beside multimedia processing, linear algebra calculations benefit especially from the advantages SIMD provides.
Since linear algebra is deeply fundamental and performance critical to many scientific calculations *Basic Linear Algebra Subprograms*, BLAS in short, were introduced. BLAS are highly optimized subroutines, aimed at minimizing the computational cost of linear algebra operations by efficient implementation of the best available algorithms. An additional benefit of the existence of these

standardized libraries is, that processor manufacturers are able to optimize their design for these algorithms and thereby further increase performance. For a further reaching introduction into BLAS and the basics of computational linear algebra, the textbook by Nassif, Erhel and Phillipe on that topic is recommended [24].
Up to this point the fundamentals of high-performance programming, that are not controllable by the developer, were introduced. The next section will treat the steps one can take during program development to enable fast execution speeds.

2.1.2. Efficient Programming

After the short introduction of the hardware aspects of high performance computing, the next step is to discuss a subset of principles on efficient programming. In this section the path to computationally efficient code will be laid out and key methods available in the Python programming language will be discussed.

When solving a programming problem, the first working version will rarely be fast. This is the starting point on the endless quest for optimal performance. There are some general rules to consider before and during code optimization. When aiming for fast code it is recommended to keep this quote, attributed to Donald E. Knuth, in mind:

Premature optimization is the root of all evil.

It captures the essence of the following guideline. When writing high performance code:

- first make it work.
- then make it right.
- finally make it fast (enough).

In code development, the first step should always be to *make it work*. This means to write code that consistently and in all known edge cases gives the expected result. Test driven development, like detailed in Govindaraj's textbook [25], is a good practice to ensure

that this goal is achieved. Usually when reaching a working code version, it will be unstructured and hard to read. This is when one should *make it right*. Making it right involves refactoring the code[4] and ensuring that a consistent programming style is used. The availability of adequate documentation will enable future work on the code, either by the original developer or by someone else. After making it right performance testing should take place and the developer has to decide if the code is fast enough. *Timing* and *profiling* are the main tools to assess this. If the current code version does not fulfill the performance requirements, optimization should take place.

How to optimize

There are many routes for optimization, all of which should be accompanied by profiling and unit testing[5]. In the following, general approaches to increase performance are listed, and thereafter Python specific means will be discussed. The first impulse when confronted with code that runs too slowly should always be to look for algorithmic improvements. Implementing a more efficient algorithm has a high potential for performance enhancement, although it can also be a very time consuming effort. An upside of implementing a better algorithm by oneself is the possibility to adept to the specific problem one faces. Another approach is to build on the work of others and import code provided by third parties. This can be a quick remedy, especially for more complex algorithms that are commonly used. Such algorithms are usually well tested and often are already highly optimized. Nevertheless, some care has to be taken, like with every third party software, since there might by problems that the original developer has not accounted for. Finally, when all the resources the current programming language offers are exhausted, there is often the possibility

[4] rearranging code segments without changing the functionality
[5] testing if the code produces the expected result

to tie in code from lower level languages for performance critical sections.

How to optimize in Python

In this section the routes for optimization will be discussed in the context of Python. The main focus here will lie on the last option of tying-in code from lower level languages and the possibility of compiling Python code. Nevertheless, the general options of algorithmic improvement and import of third-party code are also available in Python. In fact the open-source nature of Python and its broad use in many scientific areas has led to many projects that provide highly optimized algorithms. Additional details on the development of high-performing Python code can be found in Gorelick and Ozsvald's book on high performance Python [26].

One main disadvantage of pure Python is its relative slowness when performing numerical operations. This has two major reasons, explained in detail in chapter one of Gorelick and Ozsvald's *High Performance Python* [26, ch. 1] or in this blog post by Jake van der Plas [27].
The first is the absence of *static typing*, i.e. the initialization of a variable with a fixed data type. Instead, variables in Python can change data type and *lists* or *tuples* can contain various data types. While undoubtedly useful in most circumstances, when dealing with large arrays with entries of identical data type, having to check every entry's type degrades performance.
The second factor contributing to Pythons inferior performance is the way Python executes code. Python is an interpreted language. Interpreted code in general has performance disadvantage compared to compiled code. This is based on the fundamental difference between these two ways of translating human written code into machine language. Interpreters translate programs line-by-line, while compilers perform the translation before the program is executed. From a performance standpoint interpretation is worse,

since the processor has to wait for the interpreter to finish its translation. Furthermore, since the interpreter only sees one line at a time, it lacks the possibility for automatic optimization.
An additional peculiarity of Python is the *Global Interpreter Lock*, or GIL. The GIL complicates the application of multi-threading, and can be a performance hindrance for heavily parallel operations. A more detailed description of the inner workings of Python is given in Jake van der Plas' handbook on data science [28] or the official Python documentation [29]. There are a few popular ways to keep the general high development speed of Python programs, but increase its execution speed. Five of them are introduced in the following.

PyPy is the first in the list. PyPy [30] replaces the standard Python interpreter, which is written in C, with a *JIT*[6] compiler written in Python itself. Just-in-time compilers perform the compilation while the program is executing. In most cases this compilation results in faster execution, compared to standard Python. There are also a few downsides to PyPy. It is build on a reduced set of the Python functionality, called the RPython language, and is therefor not fully compatible with all Python programs and libraries.

Cython is a project dedicated to connecting C and Python [31]. It enables easy integration of existing C functions into Python programs, but also the generation of fast C code from Python code by including type declaration. One main reason to use Cython is the possibility to write functions in Python, enjoy the easy development environment, and then "translate" this function into a C-extension, which is significantly faster.

F2Py is the Fortran equivalent of Cython [32]. F2Py itself is part of NumPy, which will be discussed later in more detail. The F2Py

[6] just-in-time

project enables the integration of Fortran routines in Python, by means of an extension module.

Numba is a project aimed at increasing the speed of numerical calculations [33]. It works by JIT-compiling Python functions via the LLVM compiler [34] directly to machine code. The charm of Numba is the ease of use. In the ideal case, one just adds a *decorator* to a Python function operating on NumPy arrays, and achieve a significant performance increase.

NumExpr is a very specialized compiler, targeting large array operations [35]. It works by parsing single expressions and smartly breaking down big arrays into smaller blocks. This results in more efficient cache usage and NumExpr also enables the distribution of the calculation across multiple CPU cores.

Since one of the main goals of the Magpylib project is to create an easy to use tool, running on a wide range of computers, and even under different operating systems, the decision was made to stick to the standard Python library only with the addition of **NumPy**. Given a specific use case, employing one of the above mentioned projects, may yield significant performance improvements, at the cost of flexibility.

NumPy

NumPy [36] is one of the non-standard modules that enable vast performance increases in Python. Since it is heavily used in this thesis, a more detailed introduction is provided here. Additional insight into the inner workings of NumPy can be found at the official project page [37]. More information on the history and motivation behind NumPy is provided in the first chapter of Travis E. Oliphant *Guide to NumPy* [16].
NumPy has a few distinct features, that make it ideal for high-performance numerical computing. The first is centered around the

basic NumPy object, the *nd-array*. In NumPy data is organized into *arrays*. Such an array, like a Python *list* can be multidimensional, with the significant difference that a NumPy array stores data of only one *data-type*. This way the *dynamic type checking* needed in a Python list, as well as the overhead associated with it, can be avoided. Another property differentiating nd-arrays from Python lists is the way NumPy physically stores data. Whereas in standard Python lists, not the data itself but pointers to the data's memory address are stored, data in NumPy arrays is stored in a contiguous block in memory. When operating on elements of the array in a serial fashion, the spatial locality of data leads to an increase in performance. The full potential of NumPy is unfolded when one performs the same operation on the whole array simultaneously. This leverages the SIMD capabilities of modern CPUs and in combination with the removal of the dynamic type checking leads to a significant performance increase. Another way the use of NumPy enables faster execution speeds is the utilization of the integrated NumPy *universal functions*, ufunc in short. These are functions written to perform operations, like many linear algebra operations, on nd-arrays. Since most of scientific and high-performance computing in Python relies on NumPy, these ufuncs are highly optimized and can be used to increase performance over a standard Python implementation. These functions are often written in C or Fortran and make use of underlying BLAS libraries. For further information on NumPy, with the focus on data science, chapter 2 of Jake van der Plas' *Python Data Science Handbook* [28] is recommended.

This concludes the discussion of computer science related topics in this thesis. In the next section the fundamentals of magnetism will be introduced.

2.2. Magnetism Basics

In this section the classical theory of magnetism will be revisited and the aspects on which this thesis builds will be highlighted. The

discussion will start with the microscopic Maxwell equations but omit the introduction to these formulas. For a extensive introduction to these equations and a broader treatment of the fundamental theory of magnetism, Griffiths' *Introduction to Electrodynamics* [38], Jackson's *Classical Electrodynamics* [39] and Purcell's *Electricity and Magnetism* [40] are recommended.

The microscopic Maxwell equations, connecting the microscopic electric and magnetic fields $\boldsymbol{\mathcal{E}}$ and $\boldsymbol{\mathcal{B}}$ with the total charge and current densities ρ and $\mathbf{j}$ are listed below.

$$\nabla \cdot \boldsymbol{\mathcal{E}} = \frac{\rho}{\epsilon_0} \tag{2.1}$$

$$\nabla \cdot \boldsymbol{\mathcal{B}} = 0 \tag{2.2}$$

$$\nabla \times \boldsymbol{\mathcal{E}} = -\frac{\partial \boldsymbol{\mathcal{B}}}{\partial t} \tag{2.3}$$

$$\nabla \times \boldsymbol{\mathcal{B}} = \mu_0 \mathbf{j} + \mu_0 \epsilon_0 \frac{\partial \boldsymbol{\mathcal{E}}}{\partial t} \tag{2.4}$$

Albeit these microscopic equations describe the whole of electromagnetic phenomena perfectly, most applications can be described by a macroscopic formalism, that is easier to handle. In the following the introduction of macroscopic quantities will be motivated. An impediment to working with the microscopic equations are the high spatial and temporal variation rates of the $\boldsymbol{\mathcal{E}}-$ and $\boldsymbol{\mathcal{B}}-$ fields. One can circumvent the cumbersome treatment of these fluctuations by assuming that the spatial and temporal extent of the region of interest is comparatively large. In that case one can simplify the calculation by averaging the $\boldsymbol{\mathcal{E}}-$ and $\boldsymbol{\mathcal{B}}-$ field over a suitably large volume and time frame. Denoting this average determination by $\langle \ldots \rangle$, leads to the definition of the macroscopic fields $\mathbf{E}$ and $\mathbf{B}$.

$$\langle \boldsymbol{\mathcal{E}} \rangle = \mathbf{E} \tag{2.5}$$

$$\langle \boldsymbol{\mathcal{B}} \rangle = \mathbf{B} \tag{2.6}$$

Assuming the averaging is commutable with spatial and temporal derivations, the macroscopic Maxwell equations follow from equations (2.1) to (2.4).

$$\nabla \cdot \mathbf{E} = \frac{\langle \rho \rangle}{\epsilon_0} \tag{2.7}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{2.8}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{2.9}$$

$$\nabla \times \mathbf{B} = \mu_0 \langle \mathbf{j} \rangle + \mu_0 \epsilon_0 \frac{\partial \mathbf{E}}{\partial t} \tag{2.10}$$

In the macroscopic equations the averaged total charge and current densities $\langle \rho \rangle$ and $\langle \mathbf{j} \rangle$ appear. Close inspection of the physical origins of these densities warrants the subdivision into *free* and *bound*, denoted by the indices f and b [41].

$$\langle \rho \rangle = \rho_f + \rho_b \tag{2.11}$$

$$\langle \mathbf{j} \rangle = \mathbf{j}_f + \mathbf{j}_b \tag{2.12}$$

The meaning of those terms and the implications on the formalism will be discussed in the following. First the difference between free and bound charges will be described.
An example for free charges are the conduction electrons in a metal. In the presence of an external electric field, they accelerate in one direction and form a macroscopic current. Bound charges on the other hand are restricted to a specific area in the material. This does not mean, that they can not move, but rather that they are subject to a strong rebounding force. A familiar example of such a property are dielectric materials in a capacitor. The electric field dislocates positive and negative charges, without resulting in a sustained macroscopic current. This behaviour gives rise to (or orients already existing) electrical dipoles, which can be described

by the total electrical dipole moment per unit volume or *polarization* **P**. Mathematically the relation between the polarization **P** and the bound charge density ρ_b reads

$$-\nabla \cdot \mathbf{P} = \rho_b \tag{2.13}$$

Combining equations (2.11), (2.13) and (2.22) leads to the introduction of the electric displacement field **D**

$$\begin{aligned} \epsilon_0 \nabla \cdot \mathbf{E} &= \rho_f + \rho_b \\ \epsilon_0 \nabla \cdot \mathbf{E} &= \rho_f - \nabla \cdot \mathbf{P} \\ \nabla \cdot (\epsilon_0 \mathbf{E} + \mathbf{P}) &= \rho_f \Rightarrow \\ \mathbf{D} &= \epsilon_0 \mathbf{E} + \mathbf{P} \,. \end{aligned} \tag{2.14}$$

Equation (2.14) is called the constitutive equation of electrodynamics. Similar to the charge density, also the current density can be subdivided into a free current density $\mathbf{j}_f$ and a bound current density $\mathbf{j}_b$ [40, ch. 11.10]. The free current is the macroscopic flow of charges, like the current sent around a wire by a battery. For the bound current, there are two contributions. The first one follows from the time derivative of equation (2.13). This reads

$$\frac{\partial \rho_b}{\partial t} - \nabla \cdot \frac{\partial \mathbf{P}}{\partial t} = 0 \,. \tag{2.15}$$

This equation is reminiscent of the continuity equation and therefore it stands to reason to define a the polarization current density $\mathbf{j}_P$ as

$$\mathbf{j}_P = \frac{\partial \mathbf{P}}{\partial t} \,. \tag{2.16}$$

Careful inspection of equation (2.15) reveals an interesting possibility. One can add a divergence free term and still fulfill the equation.

This in fact necessary to achieve a general solution to equation (2.15). The term that is missing is called $\mathbf{j}_M$, where the subscript M stands for *magnetization*. As stated, the divergence of $\mathbf{j}_M$ vanishes

$$\nabla \cdot \mathbf{j}_M = 0 \,. \tag{2.17}$$

This current density describes the orbital motion of electrons around the nucleus. In the classical picture, such a motion constitutes a tiny current loop, which according to Biot-Savart law gives rise to a magnetic field [38, ch. 5.2]. The classical picture also explains the attribution of $\mathbf{j}_M$ to the bound current density. The orbital electrons do not contribute to a macroscopic current flow. Therefore, the bound current density $\mathbf{j}_b$ summarizes the polarization and magnetization current densities.

$$\mathbf{j}_b = \mathbf{j}_P + \mathbf{j}_M \tag{2.18}$$

With $\mathbf{j}_M$ an important concept was introduced: the magnetization $\mathbf{M}$, further treatment of which can be found in [42]. Analogous to the polarization $\mathbf{P}$, the magnetization $\mathbf{M}$ describes the total magnetic dipole moment per unit volume. The relationship between the magnetization current density $\mathbf{j}_M$ and the magnetization $\mathbf{M}$ is given by

$$\nabla \times \mathbf{M} = \mathbf{j}_M \,. \tag{2.19}$$

With this knowledge one can define another field describing magnetic behaviour, the $\mathbf{H}$-field, via an equation that is called the constitutive equation of magnetism.

$$\mathbf{H} = \frac{\mathbf{B}}{\mu_0} - \mathbf{M} \tag{2.20}$$

At this point it is necessary to clarify which quantity is meant when talking about *the* magnetic field and how to call $\mathbf{B}$ and $\mathbf{H}$.

A historical recap of the study of magnetism, that also touches on this topic can be found in Daniel C. Mattis' chapter *History of Magnetism* [43].
Commonly the **B**-field is called *magnetic flux density* or *magnetic induction* and is measured in units of Tesla (T). For the **H**- field the names *magnetic field strength/intensity* or *magnetizing field* are used and it is measured in A/m. Mathematically, there is no ambiguity (see eq. (2.20)) and the derivation outlined in this chapter acknowledges the **B**-field as the fundamental microscopic quantity, with **H** being a useful auxiliary field. There are several contributing factors to why there is no clear convention which quantity should be called *the magnetic field*. The first follows from equation (2.20). In free space, where the magnetization **M** is zero, the **B**- and **H**-field only differ by a constant factor μ_0, the permeability of free space. Therefore, examination of the magnetic field outside magnets gives no indication whether the **B**- or the the **H**-field is to be preferred. The second is encoded in equation (2.25), which is yet to be derived. Nevertheless one can verify by inspection of the formula that in the absence of a time varying electric displacement field the relation

$$\nabla \times \mathbf{H} = \mathbf{j}_f \tag{2.21}$$

holds: a free current density creates an **H**-field. Since free current densities are often what is experimentally controlled, it becomes reasonable to treat the **H**-field as *the* magnetic field.
Summarizing it can be stated, that according to the theoretical derivation, the **B**-field is the fundamental quantity, but in an experimental setup, where a magnetic field is produced from a free current density, the **H**-field is the intuitive description [40, ch. 11.10]. In the remainder of this thesis when talking about the magnetic field in free space the distinction between **B** and **H** will be omitted. In that scenario the special case of equation (2.20)

$$\mathbf{B} = \mu_0 \mathbf{H}$$

holds and there is no benefit to explicitly differentiate. When treating the magnetic field inside a material, the **B**- and **H**-field will be named explicitly, since the magnetization **M** has to be taken into account.

With every quantity needed to describe the macroscopic Maxwell equations being introduced, they are collected in the following. In analogy to the microscopic equation eq. (2.1) to (2.4) they read:

$$\nabla \cdot \mathbf{D} = \rho_f \tag{2.22}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{2.23}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{2.24}$$

$$\nabla \times \mathbf{H} = \mathbf{j}_f + \frac{\partial \mathbf{D}}{\partial t} \tag{2.25}$$

With **D** and **H** being defined by the constitutive equations

$$\mathbf{D} = \epsilon_0 \mathbf{E} + \mathbf{P} \tag{2.26}$$

$$\mathbf{B} = \mu_0 \mathbf{H} + \mu_0 \mathbf{M} \,. \tag{2.27}$$

The difficulty when solving equations (2.22) to (2.25) is to find expressions for **P** and **M** since they are not externally imposed. Furthermore, they are affected by the external fields **E** and **H**. In the description of this behaviour, one has to distinguish between cases, where an unambiguous functional relationship exists, and more complex cases. The first kind are *para*- and *dia*-magnetic, as well as dielectric materials. For such materials the relations between **E** and **P**, and **H** and **M** are described by the susceptibility χ

$$\mathbf{P} = \epsilon_0 \chi_e \mathbf{E} \tag{2.28}$$

$$\mathbf{M} = \chi_m \mathbf{H} \tag{2.29}$$

with the subscripts e and m denoting the dielectric and magnetic susceptibility respectively. The distinction between para- and diamagnetic is based in the sign of χ. Positive susceptibilities describe paramagnets, while negative susceptibilities are associated with diamagnets. For most isotropic materials and small external fields, the susceptibilities can assumed to be constant. This implies that relations (2.28) and (2.29) in such cases simplify to linear relations. Permanent magnets constitute the complex case mentioned above, in which the relation between $\mathbf{M}$ and $\mathbf{H}$ is ambiguous and equation (2.29) is an inadequate description. This means, that knowledge of the externally imposed $\mathbf{H}$-field and is not sufficient to deduce the resulting magnetization. In addition to the external field, the magnetization history of the material has to be known. The behaviour of the magnetization in permanent magnets is described by a *hysteresis*. A broader discussion of susceptibilities can be found in chapter 6.4 of Griffiths' *Introduction of Electrodynamics*, a more quantum mechanical treatment is laid out in chapter 8 of Ibach and Lüth's book on solid state physics [44] and a comprehensive introduction is given in Fazekas' *Lecture Notes on Electron Correlation and Magnetism* [45]. An example of a book which discusses hysteresis phenomena is *The Science of Hysteresis* by Giorgio Bertotti and Isaak D. Mayergoyz [46].

Having discussed the general macroscopic Maxwell equations, the next step is to introduce the restrictions under which the theoretical basis of this thesis is developed. When solving problems in the realm of magnetism, a *vector potential* $\mathbf{A}$ is often introduced, which is implicitly defined by

$$\mathbf{B} = \nabla \times \mathbf{A} \,. \tag{2.30}$$

This definition is warranted, since

$$\nabla \cdot (\nabla \times \mathbf{V}) = 0 \tag{2.31}$$

holds for any vector field $\mathbf{V}$. From that follows directly, that every $\mathbf{B}$ defined by equation (2.30) fulfills the Maxwell equation (2.23),

which legitimizes the introduction of the magnetic vector potential **A**.
In the absence of free currents and time varying electric fields, a special case of equation (2.25) emerges. With these assumptions equation (2.25) simplifies to

$$\nabla \times \mathbf{H} = 0 \, . \tag{2.32}$$

In that case a magnetic scalar potential Φ_m can be defined by

$$\mathbf{H} = \nabla \Phi_m \, . \tag{2.33}$$

Analogous to the introduction of the vector potential **A**, the scalar potential Φ_m is legitimized by the vector identity

$$\nabla \times \nabla V = 0 \, . \tag{2.34}$$

which holds for every scalar field V. In the following section the explicit calculation of Φ_m will be provided. It is important to be stated, that this magnetic scalar potential is only applicable in the magnetostatic case, where there are no free currents. Otherwise its introduction is not warranted.

2.3. The Magnetic Field of a special Triangle

In this section the procedure to calculate the magnetic field of a *homogeneously magnetically charged* triangle will be derived. This derivation is based on the work of Rubeck et al. [47], who found an analytical expression for a specific setup. In their work they also laid out how to build on their solution to treat a more general case. This extension will also be discussed.

Potential Theory

The start for the derivation of the formula for the $\mathbf{H}-$field is Appendix B in Griffith's *Introduction to Electrodynamics* on potential theory and the Helmholtz theorem [38, App. B]. It states that any well-behaved vector field can be described by a scalar and a vector potential. Griffith writes [38, p. 582ff.]:

> If the divergence and the curl of a vector function $\mathbf{F}(\mathbf{r})$ are specified, and if they both go to zero faster than $1/r^2$ as $r \to \infty$, and if $\mathbf{F}(\mathbf{r})$ goes to zero as $r \to \infty$, then $\mathbf{F}(\mathbf{r})$ is given uniquely by:
>
> $$\mathbf{F} = -\text{grad}\, U + \text{curl}\, \mathbf{W} \tag{2.35}$$
>
> with U and $\mathbf{W}$, called the *scalar* and *vector* potential respectively, given by:
>
> $$U(\mathbf{r}) \equiv \frac{1}{4\pi} \int \frac{\nabla' \cdot \mathbf{F}(\mathbf{r}')}{\| \mathbf{r} - \mathbf{r}' \|} \mathrm{d}^3 r' \,, \tag{2.36}$$
>
> $$\mathbf{W}(\mathbf{r}) \equiv \frac{1}{4\pi} \int \frac{\nabla' \times \mathbf{F}(\mathbf{r}')}{\| \mathbf{r} - \mathbf{r}' \|} \mathrm{d}^3 r' \,. \tag{2.37}$$

The corollary to this theorem also stated in the Appendix B [38, p. 582ff.], is of importance for this thesis:

> Any (differentiable) vector function $\mathbf{F}(\mathbf{r})$ that goes to zero faster than $1/r$ as $r \to \infty$ can be expressed as the gradient of a scalar plus the curl of a vector:
>
> $$\begin{aligned} \mathbf{F} = &- \nabla \left(\frac{1}{4\pi} \int \frac{\nabla' \cdot \mathbf{F}(\mathbf{r}')}{\| \mathbf{r} - \mathbf{r}' \|} \mathrm{d}^3 r' \right) \\ &+ \nabla \times \left(\frac{1}{4\pi} \int \frac{\nabla' \times \mathbf{F}(\mathbf{r}')}{\| \mathbf{r} - \mathbf{r}' \|} \mathrm{d}^3 r' \right) \,. \end{aligned} \tag{2.38}$$

Note that the volume integrals in this formulation reach from $-\infty$ to $+\infty$ and the differentiation with respect to the primed variable within the integral.

To obtain a formula for the magnetic scalar potential Φ_m equation (2.38) is adapted to the **H**-field. Utilizing that in the context of this thesis[7] the curl of **H** vanishes (see eq. (2.32)) and applying the constitutive relation (2.20) it follows that

$$\begin{aligned} \mathbf{H} &= -\nabla \left(\frac{1}{4\pi} \int \frac{-\nabla' \cdot \mathbf{M}(\mathbf{r}')}{\| \mathbf{r} - \mathbf{r}' \|} \mathrm{d}^3 r' \right) = -\nabla \Phi_m \\ &\Rightarrow \Phi_m = \frac{-1}{4\pi} \int \frac{\nabla' \cdot \mathbf{M}(\mathbf{r}')}{\| \mathbf{r} - \mathbf{r}' \|} \mathrm{d}^3 r' \, . \end{aligned} \tag{2.39}$$

Since the volume integrals in equation (2.39) have to be evaluated over the whole $\mathbb{R}^3$, they are not optimal for practical usage. In the following, formulas that only need to be evaluated over the volume of a given magnet V, will be derived. In the derivation an **idealization** is assumed: for ease of calculation it is presumed that the magnetization **falls off discontinuously** at the boundary of the magnet, like proposed by Jackson in his *Classical Electrodynamics* [39, p. 196f.].
There he starts with equation (2.39)

$$\Phi_m = \frac{-1}{4\pi} \int \frac{\nabla' \cdot \mathbf{M}(\mathbf{r}')}{\| \mathbf{r} - \mathbf{r}' \|} \mathrm{d}^3 r', \tag{2.39}$$

which has to be evaluated over the whole space. Using the idealization of a discontinuous magnetization an advantageous result can be achieved. The integration region in (2.39) can be divided into three regions. The first is the inner region of the magnet, denoted by V. The second is an infinitely thin region around the surface of the magnetic region ∂V. The last one is the outer region, which, since it is empty, does not contribute to the potential. By applying

[7] no electric fields and no free current density

Gauß's theorem to the surface region, and defining an *effective magnetic surface charge density* σ_m as

$$\sigma_m = \mathbf{n} \cdot \mathbf{M} \,, \tag{2.40}$$

Jackson's calculation results in

$$\Phi_m = \frac{-1}{4\pi} \int_V \frac{\nabla' \cdot \mathbf{M}(\mathbf{r}')}{\| \mathbf{r} - \mathbf{r}' \|} \mathrm{d}^3 r' + \frac{1}{4\pi} \oint_{\partial V} \frac{\sigma_m}{\| \mathbf{r} - \mathbf{r}' \|} \mathrm{d}a' \tag{2.41}$$

with $\mathbf{n}$ being the outward-directed surface normal vector.
For a homogeneous magnetization the volume integral gives no contribution and the problem simplifies to a two-dimensional surface integral. Jackson explicitly states that the result in (2.41) constitutes a special case and that the introduction of a surface charge should be used carefully.

Magnetic Potential of a Triangle

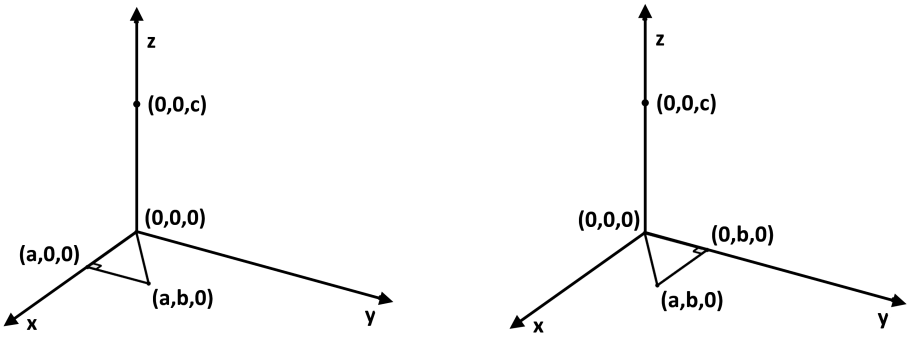
With equation (2.41) a formula that is applicable to the special case of a homogeneously magnetically charged triangle was found. The constraint that σ_m is constant is fulfilled when the triangle is part of the surface of a homogeneously magnetized body. The uniform magnetization also causes the first term in equation (2.41) to vanish. Therefore the equation simplifies to the formula found in the paper by Rubeck et al. [47]:

$$\Phi_m = \frac{\sigma_m}{4\pi} \iint_S \frac{1}{\| \mathbf{r} - \mathbf{r}' \|} \mathrm{d}S \,. \tag{2.42}$$

It has to be noted, that equation (2.42) is not truly a physical solution for the magnetic scalar potential, since the integral in the derivation is defined over a closed surface (see equation (2.41)). Nevertheless, the integral over the closed surface can be split up into the sum of multiple integrals over parts of the surface, and

in that context equation (2.42) has to be interpreted: as the contribution of a surface facet to the total scalar magnetic potential of a magnetic body.

The configuration of interest is now a right triangle with side lengths a and b, lying in the x-y-plane, with the catheti of the triangle defining the x- and y-axes. There are two cases for such a configuration. One with the right angle on the x- and the other with the right angle located on the y-axis. This distinction is necessary, to enable the calculation of arbitrary triangles, as will be described in the next subsection. The observation point, where the magnetic field is calculated is located on the z-axis.



(a) Triangle of Rubeck Type A

(b) Triangle of Rubeck Type B

Figure 2.2.: Triangles of Rubeck Type A and B. Annotated by the definition of the geometry parameters a, b and c as well as the local coordinate system.

From equation (2.42) the field components for a triangle of type A can be derived. The detailed derivation can be found in the Appendix A.

$$H_{x,\mathrm{A}} = \frac{\sigma_m}{4\pi\mu_0}\left\{\frac{-b}{2D_{ab}}\ln\left(\frac{D_{abc}+D_{ab}}{D_{abc}-D_{ab}}\right)+\frac{1}{2}\ln\left(\frac{D_{abc}+b}{D_{abc}-b}\right)\right\} \quad (2.43)$$

$$H_{y,\mathrm{A}} = \frac{\sigma_m}{4\pi\mu_0}\left\{\frac{a}{2D_{ab}}\ln\left(\frac{D_{abc}+D_{ab}}{D_{abc}-D_{ab}}\right)-\frac{1}{2}\ln\left(\frac{D_{ac}+a}{D_{ac}-a}\right)\right\} \quad (2.44)$$

$$H_{z,\mathrm{A}} = \frac{\sigma_m}{4\pi\mu_0}\left\{\arctan\left(\frac{a\,D_{abc}}{bc}\right)-\frac{|c|}{c}\arctan\left(\frac{a}{b}\right)\right\} \quad (2.45)$$

Here the abbreviations

$$\begin{aligned} D_{abc} &= \sqrt{a^2+b^2+c^2} \\ D_{ab} &= \sqrt{a^2+b^2} \\ D_{ac} &= \sqrt{a^2+c^2} \end{aligned}$$

were used.

In the case of a type B triangle, the field equations have to be slightly modified:

$$H_{x,\mathrm{B}} = \frac{\sigma_m}{4\pi\mu_0}\left\{\frac{b}{2D_{ab}}\ln\left(\frac{D_{abc}+D_{ab}}{D_{abc}-D_{ab}}\right)-\frac{1}{2}\ln\left(\frac{D_{bc}+b}{D_{bc}-b}\right)\right\} \quad (2.46)$$

$$H_{y,\mathrm{B}} = \frac{\sigma_m}{4\pi\mu_0}\left\{\frac{-a}{2D_{ab}}\ln\left(\frac{D_{abc}+D_{ab}}{D_{abc}-D_{ab}}\right)+\frac{1}{2}\ln\left(\frac{D_{abc}+a}{D_{abc}-a}\right)\right\} \quad (2.47)$$

$$H_{z,\mathrm{B}} = \frac{\sigma_m}{4\pi\mu_0}\left\{-\arctan\left(\frac{b\,D_{abc}}{-ac}\right)-\frac{|c|}{c}\arctan\left(\frac{b}{a}\right)\right\} \quad (2.48)$$

Calculating the Magnetic Field of a general Triangle

With the introduction of formulas (2.43) to (2.45) and (2.46) to (2.48) the theoretical foundation for the implementation presented in this thesis is laid. From these equations, describing the special case of right triangles with a specific location of the observation point, a generalization can be derived. The idea was also introduced in the

paper by Rubeck et al. [47] and is based on a geometric decomposition of an arbitrary triangle. The start of this decomposition is a setup of a homogeneously magnetically charged triangle and an observation point, that does not lie in the plane of the triangle. Such a configuration is shown in figure 2.3.

The first step in the decomposition is to orthogonally project the observation point **P** onto the plane and the edges of the triangle. This defines four *right angle points* $\mathbf{RAP}_i$. The subscript denotes the target of the projection, i.e. either the plane or the edge between the specified corners. These projections are displayed in figure 2.4, where both the 2D and 3D view are shown.
Given these right angle points, one can define 6 right triangles, each of which has one corner in the projection point on the plane $\mathbf{RAP}_P$. Another corner always lies in one of the projection points onto the edges. The edges of these newly specified sub-triangles define the x- and y- axis in the local coordinate system of the sub-triangle. Hence, depending on the type of sub-triangle, the formulas (2.43) to (2.45) or (2.46) to (2.48) can be applied. The sub-triangle defined by the points $\mathbf{RAP}_P$-$\mathbf{RAP}_{AC}$-**C** is shown in figure 2.5.
As mentioned above, the magnetic field of each of these sub-triangles can be calculated by the equations corresponding to the Rubeck type of the triangle. The Rubeck types are defined by figure 2.2. For the calculation it is assumed, that all sub-triangles have the same magnetic charge density σ_m as the original triangle. Using the superposition principle, one then can express the field of the original triangle as the sum of the fields of the sub-triangles, with appropriate sign. Rubeck proposed in section IV of his paper [47] that the sign of each sub-triangle can be deduced by inspecting its relative location with respect to the original triangle:

> a right [sub-]triangle is counted positively if the right [sub-] triangle covers (or covers partially) the [original] triangle . Otherwise it is counted negatively.

For the example triangle from figure 2.3 all six right sub-triangles are collected in figure 2.6. In this figure the top view of this setup

is displayed for better clarity. Furthermore, the sub-triangles are colored according to their sign.

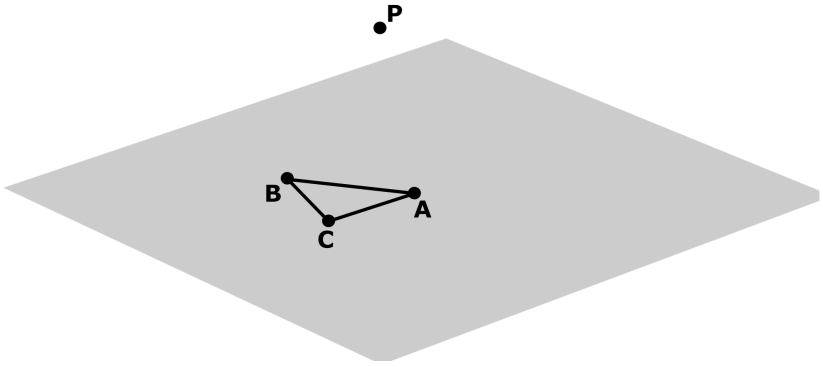
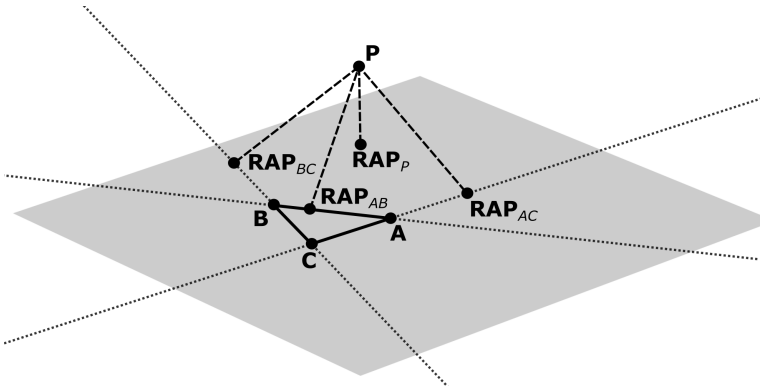
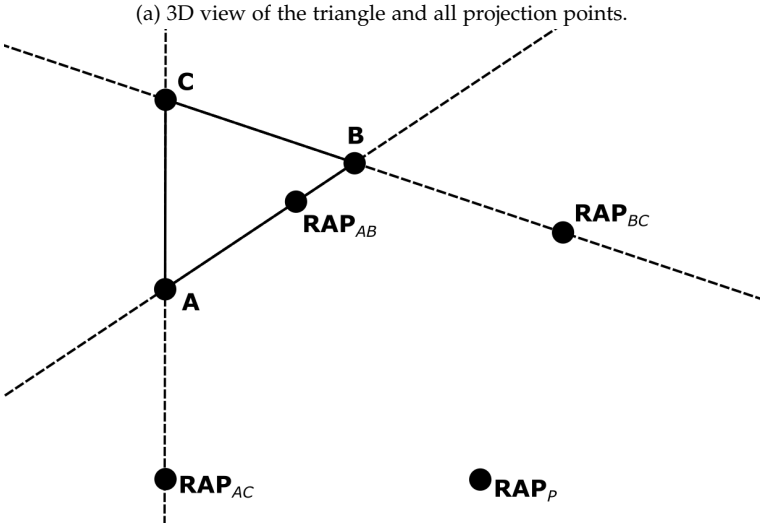


Figure 2.3.: Setup of a general triangle A-B-C and observation point P.



(a) 3D view of the triangle and all projection points.



(b) 2D view of the triangle and all projection points.

Figure 2.4.: View of the triangle and all projection points in 2D and 3D. **RAP** in this context stands for *Right Angle Point* and emphasizes, that the point is the orthogonal projection of the observation point **P** onto the respective plane or triangle edge.

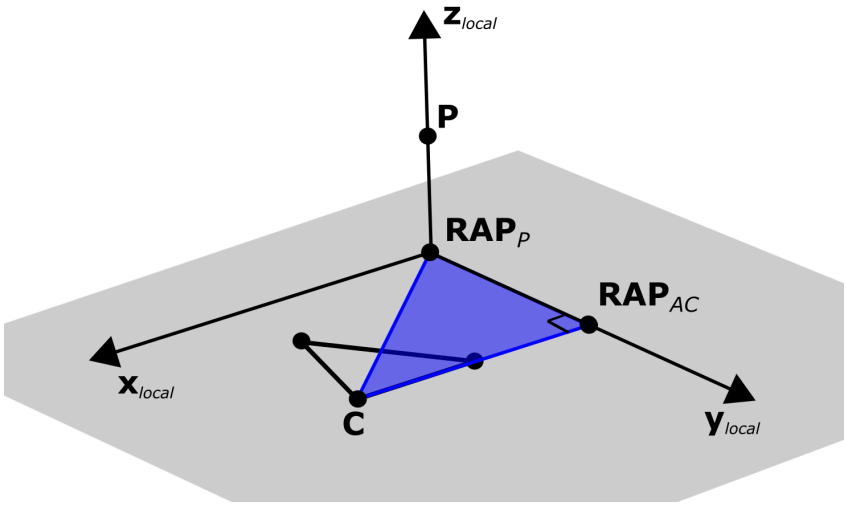
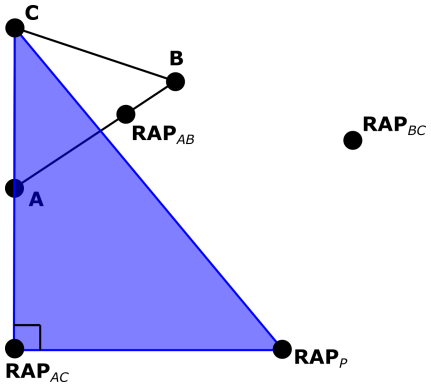
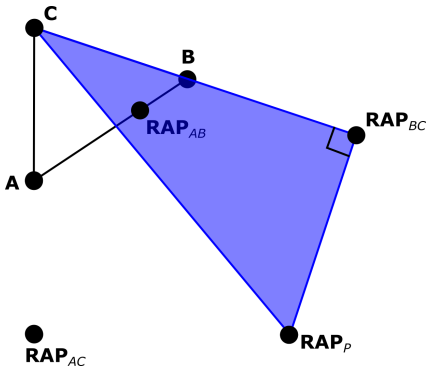


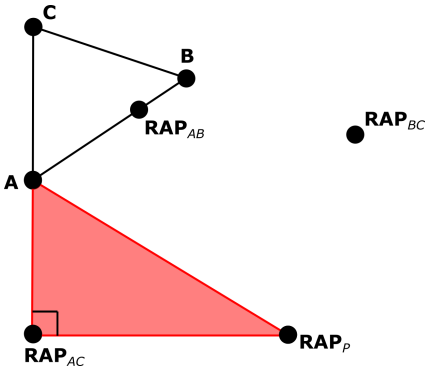
Figure 2.5.: The sub-triangle defined by the points $\mathbf{RAP}_P$-$\mathbf{RAP}_{AC}$-**C** is highlighted in blue. Additionally, the local coordinate system of this sub-triangle is drawn.



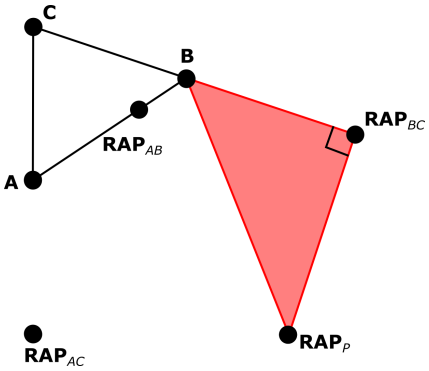
(a) Sub-triangle $\mathbf{RAP}_P - \mathbf{RAP}_A C - \mathbf{C}$. The blue color indicates, that the sign of this triangle is positive.



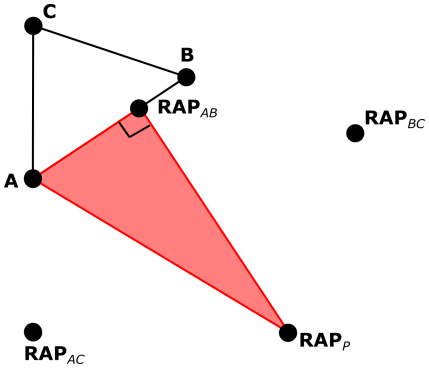
(b) Sub-triangle $\mathbf{RAP}_P - \mathbf{RAP}_{BC} - \mathbf{C}$. The blue color indicates, that the sign of this triangle is positive.



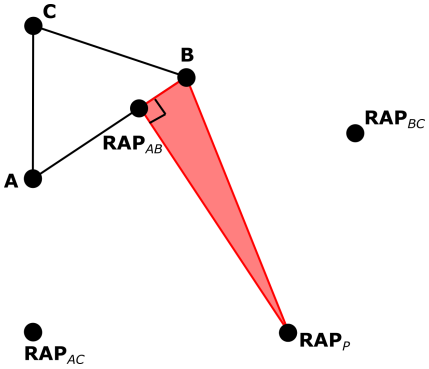
(c) Sub-triangle $\mathbf{RAP}_P - \mathbf{RAP}_{AC} - \mathbf{A}$. The red color indicates, that the sign of this triangle is negative.



(d) Sub-triangle $\mathbf{RAP}_P - \mathbf{RAP}_{BC} - \mathbf{B}$. The red color indicates, that the sign of this triangle is negative.



(e) Sub-triangle $\mathbf{RAP}_P - \mathbf{RAP}_{AB} - \mathbf{A}$. The red color indicates, that the sign of this triangle is negative.



(f) Sub-triangle $\mathbf{RAP}_P - \mathbf{RAP}_{AB} - \mathbf{B}$. The red color indicates, that the sign of this triangle is negative.

Figure 2.6.: Top view of all 6 right sub-triangles obtained by the geometric decomposition of the setup shown in Figure 2.3. Each sub-triangle is colored according to its sign, i.e. if it counts positively (blue) or negatively (red) to the magnetic field of the original triangle.

This concludes the chapter on the theoretical introduction. In this chapter a detour into computer science and programming basics, as well as an detailed derivation of formulas describing the magnetic field of a *homogeneously magnetically charged* triangle was given. The next step is an in-depth discussion of the algorithm used to implement this calculation in Python.

3. Introduction to the Algorithm

The aim of this chapter is to explain the implementation of the magnetic field calculation in Python. The original Python code can be found in Appendix B. While the code presented in the appendix is capable of accepting multiple facets and observation points, this discussion will start with the case of one facet and one observation point. After the detailed description of the algorithm employed to compute this non-vectorized case, a separate section will treat the peculiarities of the vectorization.

3.1. Algorithmic Outline

Before discussing the algorithm in detail, a summary of the structure of the presented code is given. In the following, each step will be discussed separately. The first point, the extraction of the inputs, will be pushed to section 3.3, since it is only a matter of interest if more than one facet or observation point is involved.

- Extract the inputs
- Calculate the orthogonal projection of the observation point onto the triangle plane.
- Sort and rename the triangle points with respect to the observation point and oriented plane normal.
- Calculate the orthogonal projection of the observation point onto the triangle edges, which defines six right sub-triangles.
- For each of these sub-triangles:

  - Calculate the Rubeck type and the sign in the geometric decomposition
  - Calculate the geometry parameters and the local axes
  - Calculate the field in the observation point in the local coordinate system and transform it into the global coordinate frame
- Add the fields of all sub-triangles with their respective sign
- return the total field

For the next sections, it is assumed that the inputs include the three triangle vertices, called $\mathbf{T}_1, \mathbf{T}_2$ and $\mathbf{T}_3$, the observation point $\mathbf{P}$, a normalized and *outward pointing* plane normal vector $\mathbf{n}$ and a *magnetic surface charge density* σ_m. How to derive the last two parameters will be discussed in section 3.3.

3.2. Steps of the Algorithm

3.2.1. Projecting the Observation Point onto the Triangle Plane

The first step in the case of one facet and one observation point is to find the projection of the observation point $\mathbf{P}$ onto the plane of the triangle. To achieve this, the orthogonal distance of the observation point from the plane is calculated, by taking the inner product of the vector from one of the triangle vertices $\mathbf{T}_i$ to $\mathbf{P}$ with the plane normal $\mathbf{n}$. Then a vector parallel to the plane normal $\mathbf{n}$ and with this length is subtracted from the observation point. The result of this subtraction is the projection point in the plane of the triangle $\mathbf{RAP}_P$. Written mathematically this reads

$$\mathbf{RAP}_P = \mathbf{P} - \left((\mathbf{P} - \mathbf{T}_i) \cdot \mathbf{n}\right) \mathbf{n} \tag{3.1}$$

A three dimensional graphical visualization of the projection is given in Figure 3.1

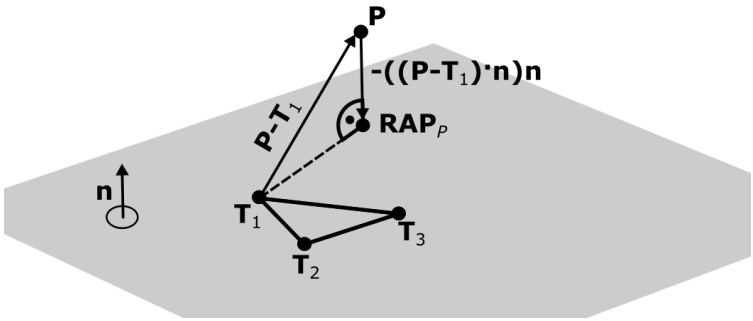


Figure 3.1.: Three dimensional representation of the orthogonal projection of the observation point **P** onto the plane of the triangle defined by the points $\mathbf{T}_1, \mathbf{T}_2$ and $\mathbf{T}_3$. **n** denotes the normalized *outward pointing* plane normal vector.

3.2.2. Sorting the Triangle

Following the determination of the projection onto the triangle plane, the corners of each triangle can to be sorted. The decision to sort the triangle corners was made to ensure consistency and the ability to rely on the knowledge of the order of elements in an array. The triangle is sorted according to the following pattern. First, a coordinate frame is defined, in which the projection of the observation point $\mathbf{RAP}_P$ lies in the origin, the triangle lies in the x-y-plane, and the oriented plane normal **n** defines the positive z-axis. In this system three vectors are defined. Each vector starts at the projection of the observation point and ends at one of the triangle corners. One of these vectors is selected as reference and then the *oriented* angle in the range of $-\pi$ to π to the other two vectors is calculated. A detailed discussion of the meaning of the *oriented* angle follows in the next subsection. With the angles between these vectors known, new names are assigned to the triangle vertices. The corner with the largest oriented angle is assigned to the variable **A**, the one with the smallest oriented angle to the variable **B** and the remaining point to the variable **C**. Note that the standard convention of counting angles counter-clockwise is used. Put in non mathematical terms, this means that when standing in $\mathbf{RAP}_P$

and with **n** defining *up*: **A** is the far *left* triangle vertex, while **B** is the far *right* one. The remaining *middle* vertex is renamed **C**. A graphical representation of this sorting is given in Figure 3.2.

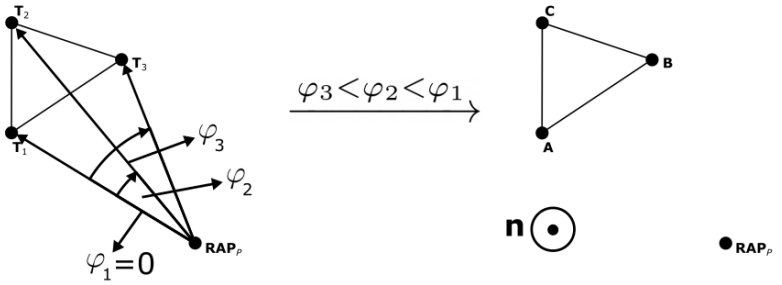


Figure 3.2.: Two dimensional representation of the sorting of the triangle $\mathbf{T}_1$-$\mathbf{T}_2$-$\mathbf{T}_3$ with respect to the projection of the observation point onto the triangle plane $\mathbf{RAP}_P$ and the plane normal vector **n**. φ denotes the oriented angle. In this case **n** points out of the paper plane towards the reader. Note: **n** defines the *up*-direction, and **P** does not necessarily lie *above* the triangle. A detailed discussion of **n** is given in section 3.3.

Calculating the Oriented Angle between two Vectors

An important step in the sorting of the input triangle is the calculation of the oriented angle between two vectors. This is not a trivial task, since this is not a well defined problem with only two input vectors[1]. To fully define the system, additionally the direction of the right handed plane normal vector has to be given, or in simpler terms, the *up*-direction has to be specified. In the problem at hand, an oriented plane normal vector is already defined by the outward pointing surface normal vector **n**. This **n** is repurposed here to play the part of the right handed plane normal to the plane specified by the two input vectors. Given two vectors and *up* being defined by the normalized plane normal, the oriented angle can be calculated

[1] because the cross product is not commutative

by the following method. The presented procedure was proposed by the user Adrian Leonhard on the StackOverflow forum [48]. Using the definition of the dot product and the cross product, one can compute the *unoriented* angle α between the two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ in two different ways:

$$\cos(\alpha) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\| \, \|\mathbf{v}_2\|} , \tag{3.2}$$

$$\sin(\alpha) \, \mathbf{n}_1 = \frac{\mathbf{v}_1 \times \mathbf{v}_2}{\|\mathbf{v}_1\| \, \|\mathbf{v}_2\|} , \tag{3.3}$$

with $\mathbf{n}_1$ being the normalized right handed cross product between $\mathbf{v}_1$ and $\mathbf{v}_2$

$$\mathbf{n}_1 = \frac{\mathbf{v}_1 \times \mathbf{v}_2}{\|\mathbf{v}_1 \times \mathbf{v}_2\|} , \tag{3.4}$$

and equation (3.3) following from the connection between the norm of the cross product and the sine of the angle between two vectors

$$\|\mathbf{v}_1 \times \mathbf{v}_2\| = \sin(\alpha) \|\mathbf{v}_1\| \, \|\mathbf{v}_2\| . \tag{3.5}$$

In a two dimensional and orthonormal coordinate system, where $\mathbf{v}_1$ is the positive x-axis and both $\mathbf{v}_1$ and $\mathbf{v}_2$ are anchored in the origin, the **oriented** angle β from $\mathbf{v}_1$ to $\mathbf{v}_2$ is related to α by

$$\beta = \begin{cases} \alpha, & \text{for } \mathbf{v}_2 \text{ ending in quadrant I or II} \\ 2\,\pi - \alpha, & \text{for } \mathbf{v}_2 \text{ ending in quadrant III or IV} \end{cases} . \tag{3.6}$$

Knowing the symmetry and periodicity of the trigonometric functions

$$\cos(\phi) \equiv \cos(-\phi) \equiv \cos(2\pi - \phi) , \tag{3.7}$$

$$\sin(\phi) \equiv -\sin(-\phi) \equiv -\sin(2\pi - \phi) \,, \tag{3.8}$$

the unoriented angle α in equations (3.2) and (3.3) can be replaced by the oriented angle β:

$$\cos(\beta) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\| \, \|\mathbf{v}_2\|} \,, \tag{3.9}$$

$$\pm \sin(\beta) \, \mathbf{n}_1 = \frac{\mathbf{v}_1 \times \mathbf{v}_2}{\|\mathbf{v}_1\| \, \|\mathbf{v}_2\|} \,, \tag{3.10}$$

with the plus-minus sign indicating the cases from equation (3.6). To get rid of the plus-minus sign equation (3.10) can be multiplied by the normalized oriented plane normal vector $\mathbf{n}$, known from the input. The inner product of $\mathbf{n}_1$ and $\mathbf{n}$ yields another plus-minus sign under the same conditions for the orientation of $\mathbf{v_2}$ relative to $\mathbf{v_1}$. This cancels the plus-minus sign introduced in equation (3.10). Dividing the modified (3.10) by (3.9) and using the definition of the tangent leads to

$$\tan(\beta) \equiv \frac{\sin(\beta)}{\cos(\beta)} = \frac{(\mathbf{v}_1 \times \mathbf{v}_2) \cdot \mathbf{n}}{\mathbf{v}_1 \cdot \mathbf{v}_2} \,. \tag{3.11}$$

Inverting the tangent finally yields the formula for the oriented angle β between the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$

$$\beta = \arctan\left(\frac{(\mathbf{v}_1 \times \mathbf{v}_2) \cdot \mathbf{n}}{\mathbf{v}_1 \cdot \mathbf{v}_2}\right) \,. \tag{3.12}$$

In the implementation care has to be taken, that the arctan function returns the angle in the correct quadrant, which is ensured by using *numpy.arctan2()*.

3.2.3. Projecting the Observation Point onto the Triangle Edges

After the sorting of the triangle corners, the next step is to find the projection of the observation point onto the triangle edges. By performing this projection after sorting, the order of the projection points is known. A similar procedure to finding the projection onto the triangle plane is implemented. For each edge of the triangle a corner $\mathbf{X}$ on this edge is chosen, and the vector towards the observation point $\mathbf{P}$ is calculated. Furthermore, the vector towards the other corner on that edge $\mathbf{Y}$ is defined. Then the inner product of the vector from the triangle vertex to the observation point and the normalized edge vector $\mathbf{v}_{XY}$, which is given by

$$\mathbf{v}_{XY} = \frac{\mathbf{Y} - \mathbf{X}}{\|\mathbf{Y} - \mathbf{X}\|} , \tag{3.13}$$

is computed. This gives the length of the projection of ($\mathbf{P}$-$\mathbf{X}$) onto the edge. Therefore, the distance of the projection point from the corner is known and one can go this distance in the direction of the edge unit vector $\mathbf{v}_{XY}$. Mathematically put this reads

$$\mathbf{RAP}_{XY} = \mathbf{X} + ((\mathbf{P} - \mathbf{X}) \cdot \mathbf{v}_{XY})\, \mathbf{v}_{XY} . \tag{3.14}$$

Figure 3.3 shows a graphical interpretation of formula (3.14).

3.2.4. Defining the Sub-Triangles

Having sorted the triangle facet and knowing all projection points, one can now define the six right sub-triangles of the geometric decomposition described section 2.3. For the implementation of the code it was advantageous to have the corners and also the sub-triangles put in order. The order and the constituting points of the sub-triangles are listed in Table 3.1.

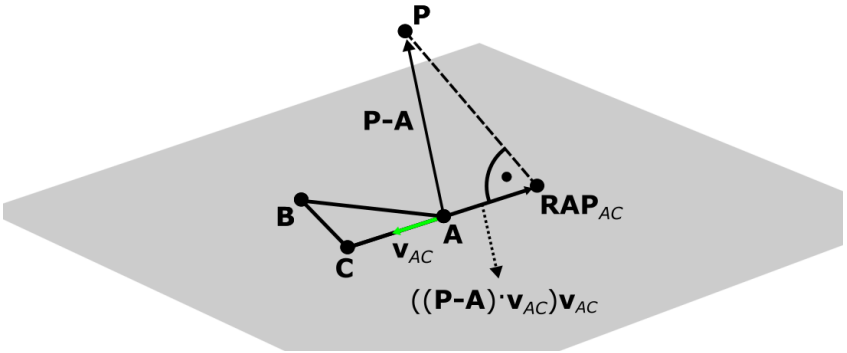


Figure 3.3.: Three dimensional representation of the projection of the observation point **P** onto the triangle edge. $\mathbf{v}_{AC}$ is the normalized edge vector, colored in green.

Table 3.1.: Ordered list of all six sub-triangles.

n ... numbering of the sub-triangle
$\mathbf{P}_i$ ... i^{th} corner of the sub-triangle
$\mathbf{RAP}_P$ ... projection of observation point onto triangle plane
$\mathbf{RAP}_{XY}$ ... projection of observation point onto triangle edge defined by the corners **X** and **Y**.

n	$\mathbf{P}_1$	$\mathbf{P}_2$	$\mathbf{P}_3$
1	$\mathbf{RAP}_P$	$\mathbf{RAP}_{AC}$	**C**
2	$\mathbf{RAP}_P$	$\mathbf{RAP}_{BC}$	**C**
3	$\mathbf{RAP}_P$	$\mathbf{RAP}_{AC}$	**A**
4	$\mathbf{RAP}_P$	$\mathbf{RAP}_{BC}$	**B**
5	$\mathbf{RAP}_P$	$\mathbf{RAP}_{AB}$	**A**
6	$\mathbf{RAP}_P$	$\mathbf{RAP}_{AB}$	**B**

3.2.5. Calculating the Field of each Sub-Triangle

After sorting the original facet and finding all projection points, the next step is to calculate the contribution to the field of each sub-triangle. Before performing the field computation, the Rubeck type[2] and the geometry parameters for equations (2.43) to (2.48) have to be extracted. Following the field calculation in the local coordinate

[2] see Figure 2.2

system, the result has to be transformed into the global coordinate frame and added up correctly. Therefore, the local coordinate basis and the sign from the decomposition[3] are needed. The following subsections treat the derivation of these parameters.

Find the Rubeck Type of the Sub-Triangle

Before assigning the geometry parameters to the sub-triangles, the Rubeck type of each sub-triangle has to be known. The type can be obtained by inspecting the orientation of the sub-triangles plane normal vector. The sub-triangles plane normal is consistently defined by

$$\mathbf{n}_{st} = (\mathbf{RAP}_{XY} - \mathbf{RAP}_{P}) \times (\mathbf{X} - \mathbf{RAP}_{XY}) \,, \tag{3.15}$$

with the index *st* standing for *sub-triangle* and **X** being the triangle vertex that is part of the respective sub-triangle. This sub-triangle's surface normal is either parallel of anti-parallel to the outward pointing surface normal **n**, defined with the input. In the parallel case the sub-triangle is of type A. In the anti-parallel case it is of type B.

$$\text{Rubeck Type} = \begin{cases} \text{A}, & \text{if } \mathbf{n}_{st} \text{ is parallel to } \mathbf{n} \\ \text{B}, & \text{if } \mathbf{n}_{st} \text{ is anti-parallel to } \mathbf{n} \end{cases} .$$

Find the Sign of the Sub-Triangle

Obtaining the sign used in the geometric decomposition is more involved. The intuitive way described in the paper by Rubeck et al. to assign every sub-triangle that overlaps with the original triangle a positive sign, is not easily implementable. A computationally

[3] see section 2.3 and Figure 2.6

light-weight solution for right and acute triangles was found, by inspecting all possible locations of $\mathbf{RAP}_P$ relative to the facet. One of these configurations is shown in Figure 2.6. The calculation of the sign is based on three aspects:

- Does $\mathbf{RAP}_P$ lie in a *big* or a *small* sector of the original triangle?
- Does $\mathbf{RAP}_P$ lie inside the triangle?
- Where on the edge, relative to the corner, does $\mathbf{RAP}_{XY}$ lie?

Before engaging the procedure used to determine these factors, it has to be defined what a *big* and what a *small* sector is. In Figure 3.4 a configuration of a triangle, labeled according to the systematic laid out in the paragraph on sorting, is shown. The two edges crossing in the point C define a cone, in which $\mathbf{RAP}_P$ is lying. Note that this assumption is only always true, if the triangle is acute or right and the corners are sorted. Such a cone has one distinguished sector: the one containing the triangle. This sector is called the *big* sector, whereas the other one is called the *small* sector. The terminology *big* and *small* was created during discussion about the code and then adopted without any deeper meaning.

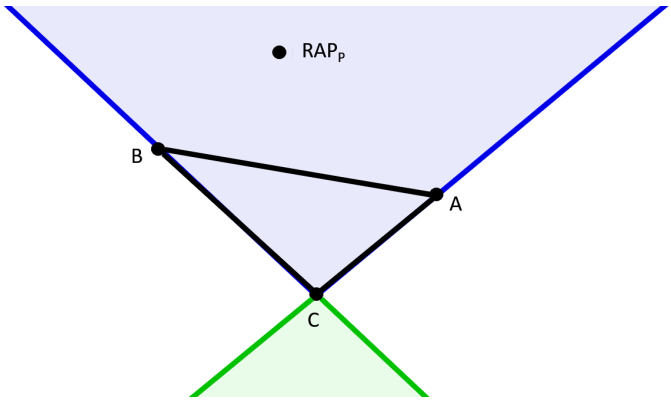


Figure 3.4.: Subdivision of the plane into a *big* (blue) and a *small* (green) sector, relative to the sorted triangle **A**-**B**-**C**. Sorting ensures that $\mathbf{RAP}_P$ does not lie in one of the white sectors.

Within the *big* sector a further distinction can be made: if $\mathbf{RAP}_P$ lies inside the triangle or not. Both the questions in which sector

$\mathbf{RAP}_P$ lies, and if it is located within the triangle can be answered by inspecting the same quantities. For that a local two dimensional coordinate system is defined, with the edges of the triangle defining the axes. This coordinate system is not necessarily orthogonal, nor is the basis, consisting of the vectors from **C** to the other corners, imperatively normalized. Nevertheless, in this coordinate system - depicted in Figure 3.5, the point $\mathbf{RAP}_P$ can be written as a linear combination of the basis vectors, with p and q as constants.

$$\mathbf{RAP}_P = \mathbf{C} + p(\mathbf{A} - \mathbf{C}) + q(\mathbf{B} - \mathbf{C}) \tag{3.16}$$

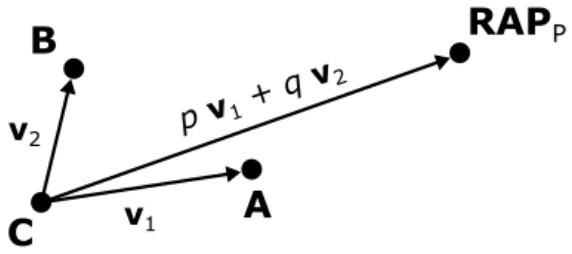


Figure 3.5.: Local coordinate system based on the triangle **A**-**B**-**C**.

Subtracting C and renaming

$$\begin{aligned} \mathbf{v}_0 &\equiv \mathbf{RAP}_P - \mathbf{C}\,, \\ \mathbf{v}_1 &\equiv \mathbf{A} - \mathbf{C}\,, \\ \mathbf{v}_2 &\equiv \mathbf{B} - \mathbf{C} \end{aligned}$$

leads to the equation

$$\mathbf{v}_0 = p\,\mathbf{v}_1 + q\,\mathbf{v}_2\,. \tag{3.17}$$

Taking the inner product of both sides once with $\mathbf{v}_1$ and once with $\mathbf{v}_2$ yields a linear system of equations for p and q.

$$\mathbf{v}_0 \cdot \mathbf{v}_1 = p\,\mathbf{v}_1 \cdot \mathbf{v}_1 + q\,\mathbf{v}_2 \cdot \mathbf{v}_1 \,, \tag{3.18}$$
$$\mathbf{v}_0 \cdot \mathbf{v}_2 = p\,\mathbf{v}_1 \cdot \mathbf{v}_2 + q\,\mathbf{v}_2 \cdot \mathbf{v}_2 \,. \tag{3.19}$$

It has to be kept in mind, that $\mathbf{v}_1$ and $\mathbf{v}_2$ are in most cases whether orthogonal nor normalized. Solving by substitution leads to

$$p = \frac{(\mathbf{v}_0 \cdot \mathbf{v}_1)\,(\mathbf{v}_2 \cdot \mathbf{v}_2) - (\mathbf{v}_0 \cdot \mathbf{v}_2)(\mathbf{v}_2 \cdot \mathbf{v}_1)}{(\mathbf{v}_1 \cdot \mathbf{v}_1)\,(\mathbf{v}_2 \cdot \mathbf{v}_2) - (\mathbf{v}_1 \cdot \mathbf{v}_2)(\mathbf{v}_2 \cdot \mathbf{v}_1)} \,, \tag{3.20}$$
$$q = \frac{(\mathbf{v}_0 \cdot \mathbf{v}_2)\,(\mathbf{v}_1 \cdot \mathbf{v}_1) - (\mathbf{v}_0 \cdot \mathbf{v}_1)(\mathbf{v}_1 \cdot \mathbf{v}_2)}{(\mathbf{v}_1 \cdot \mathbf{v}_1)\,(\mathbf{v}_2 \cdot \mathbf{v}_2) - (\mathbf{v}_1 \cdot \mathbf{v}_2)(\mathbf{v}_2 \cdot \mathbf{v}_1)} \,. \tag{3.21}$$

With p and q known, it can be deduced if $\mathbf{RAP}_P$ lies in the big or small sector of the triangle. Moreover, if it lies in the big sector, one can inspect if it is within the triangle. The condition for $\mathbf{RAP}_P$ lying in the big sector is that p and q are positive. There is no need to check both parameters, since they have the same sign by construction. The conditions that the parameters have to fulfill for $\mathbf{RAP}_P$ to be inside the triangle are

$$p,\, q \geq 0 \,, \tag{3.22}$$
$$p + q \leq 1 \,. \tag{3.23}$$

It is important to note that the second condition is only valid, if the basis vectors are not manually normalized.
As mentioned above, another aspect influencing the sign of the triangle is the position of the projection point on the edge. The sign depends on the direction of the vector pointing from the triangle corner to the projection point and two cases are differentiated. They are best described using the inner product and the sign function σ. With $\mathbf{X}$ being the triangle vertex that is part of the sub-triangle, $\mathbf{Y}$ the second vertex on the edge and $\mathbf{RAP}_{XY}$ the projection point onto that edge, the condition reads mathematically

$$\sigma((\mathbf{RAP}_{XY} - \mathbf{X}) \cdot (\mathbf{Y} - \mathbf{X})). \tag{3.24}$$

This formula returns the information if the vectors $(\mathbf{RAP}_{XY} - \mathbf{X})$ and $(\mathbf{Y} - \mathbf{X})$ are parallel (+1) or anti-parallel (-1). This corresponds to $\mathbf{RAP}_{XY}$ being located on same side of the corner $\mathbf{X}$ as the second vertex on that edge $\mathbf{Y}$ or on the opposite side. A graphical depiction is given in Figure 3.6.

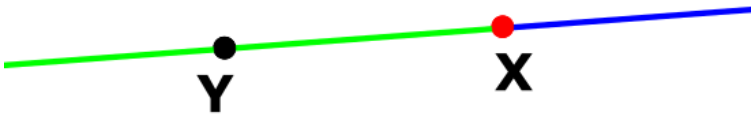


Figure 3.6.: Visualization of the possible location of the projection point $\mathbf{RAP}_{XY}$ on the triangle edge defined by the vertices $\mathbf{X}$ and $\mathbf{Y}$. The point $\mathbf{X}$ is part of the inspected sub-triangle. If $\mathbf{RAP}_{XY}$ lies on the green section of the edge, equation (3.24) equals +1, while if $\mathbf{RAP}_{XY}$ lies on the blue section of the edge, equation (3.24) equals -1.

In Table 3.2 the complete formulas to obtain the signs of the six sub-triangles are listed. The sub-triangle number n refers to the order established in Table 3.1.

with

$$\sigma(BigSmall) = \begin{cases} +1, & \text{if } \mathbf{RAP}_P \text{ in big sector} \\ -1, & \text{if } \mathbf{RAP}_P \text{ in small sector} \end{cases},$$

and

Table 3.2.: Formulas for the signs of each sub-triangle occurring in the geometric decomposition. n gives reference to the numbering of the sub-triangles established in Table 3.1. These equations only give the correct result, if the original facet is a right or acute triangle.

n	Formula for the sign of the sub-triangle
1	always +1
2	always +1
3	$\sigma((\mathbf{A}-\mathbf{RAP}_{AC})\cdot(\mathbf{A}-\mathbf{C}))\,\sigma(BigSmall)$
4	$\sigma((\mathbf{B}-\mathbf{RAP}_{BC})\cdot(\mathbf{B}-\mathbf{C}))\,\sigma(BigSmall)$
5	$\sigma((\mathbf{A}-\mathbf{RAP}_{AB})\cdot(\mathbf{A}-\mathbf{B}))\,\sigma(BigSmall)\,\sigma(InOut)$
6	$\sigma((\mathbf{B}-\mathbf{RAP}_{AB})\cdot(\mathbf{B}-\mathbf{A}))\,\sigma(BigSmall)\,\sigma(InOut)$

$$\sigma(InOut) = \begin{cases} -1, & \text{if } \mathbf{RAP}_P \text{ inside triangle} \\ +1, & \text{if } \mathbf{RAP}_P \text{ outside triangle} \end{cases}.$$

Calculating the Geometry Parameters and Local Basis

Once the type and signs of the sub-triangles are known, the geometry parameters can be extracted. This has to be done after the inspection of the sub-triangle types, because the parameters a and b as well as the local basis are assigned differently depending on the Rubeck type of the sub-triangle. Based on the type, the geometry parameters a, b and c as well as the local basis vectors $\mathbf{x}_{local}, \mathbf{y}_{local}$ and $\mathbf{z}_{local}$ can be calculated. With $\mathbf{X}$ being the triangle vertex that is part of the sub-triangle and $\mathbf{Y}$, the other corner on that edge, the geometry parameters are defined by

$$a = \begin{cases} \|\mathbf{RAP}_{XY} - \mathbf{RAP}_P\|, & \text{if type A} \\ \|\mathbf{X} - \mathbf{RAP}_{XY}\|, & \text{if type B} \end{cases},$$

$$b = \begin{cases} \|\mathbf{X} - \mathbf{RAP}_{XY}\|, & \text{if type A} \\ \|\mathbf{RAP}_{XY} - \mathbf{RAP}_P\|, & \text{if type B} \end{cases},$$

$$c = \|\mathbf{P} - \mathbf{RAP}_P\| \ ,$$

and the normalized local basis vectors by

$$\mathbf{x}_{local} = \begin{cases} \frac{\mathbf{RAP}_{XY} - \mathbf{RAP}_P}{\|\mathbf{RAP}_{XY} - \mathbf{RAP}_P\|}, & \text{if type A} \\ \\ \frac{\mathbf{X} - \mathbf{RAP}_{XY}}{\|\mathbf{X} - \mathbf{RAP}_{XY}\|}, & \text{if type B} \end{cases} ,$$

$$\mathbf{y}_{local} = \begin{cases} \frac{\mathbf{X} - \mathbf{RAP}_{XY}}{\|\mathbf{X} - \mathbf{RAP}_{XY}\|}, & \text{if type A} \\ \\ \frac{\mathbf{RAP}_{XY} - \mathbf{RAP}_P}{\|\mathbf{RAP}_{XY} - \mathbf{RAP}_P\|}, & \text{if type B} \end{cases} ,$$

$$\mathbf{z}_{local} = \frac{\mathbf{P} - \mathbf{RAP}_P}{\|\mathbf{P} - \mathbf{RAP}_P\|} \ .$$

Calculating the Partial Field in the Local Coordinate Frame and transform them into the Global Coordinate System

With all geometry parameters, types and local axis known, the magnetic field can be calculated first in the local coordinate system of every sub-triangle and then in the global coordinate frame of the whole system. The calculation in the local coordinate system follows equations (2.43) to (2.45) or (2.46) to (2.48), depending on the Rubeck type of the sub-triangle. The derivation of the transformation into the global coordinate system is presented in the following.
Starting in the local coordinate frame, denoted by the subscript l, the magnetic field of one sub-triangle can be written as

$$\mathbf{H}_l = \sum_i H_i \, \mathbf{e}_{i,l} = H_x \, \mathbf{e}_{x,l} + H_y \, \mathbf{e}_{y,l} + H_z \, \mathbf{e}_{z,l} \tag{3.25}$$

with H_i being the result of the field equations and $\mathbf{e}_{i,l}$ the local basis vectors. Since the global coordinate system $\mathbf{e}_{i,g}$ is a complete basis, the local unit vectors can be written as a linear combination of the global ones.

$$\mathbf{e}_{i,l} = \frac{\sum_j (\mathbf{e}_{i,l} \cdot \mathbf{e}_{j,g})\, \mathbf{e}_{j,g}}{\sqrt{\sum_j (\mathbf{e}_{i,l} \cdot \mathbf{e}_{j,g})^2}} \tag{3.26}$$

In the present case, also the local basis is complete and orthogonal and so the denominator in equation (3.26) equals 1 and will therefore be omitted. Using the notation in the global coordinate frame, where the i^{th} component of the j^{th} global basis vector is a $\delta_{i,j}$, the local coordinate basis can be written as:

$$\mathbf{e}_{i,l} = \begin{pmatrix} \mathbf{e}_{i,l} \cdot \mathbf{e}_{x,g} \\ \mathbf{e}_{i,l} \cdot \mathbf{e}_{y,g} \\ \mathbf{e}_{i,l} \cdot \mathbf{e}_{z,g} \end{pmatrix} . \tag{3.27}$$

Substituting this expression into equation (3.25) yields the magnetic field in the global coordinate frame

$$\mathbf{H}_g = H_x \begin{pmatrix} \mathbf{e}_{x,l} \cdot \mathbf{e}_{x,g} \\ \mathbf{e}_{x,l} \cdot \mathbf{e}_{y,g} \\ \mathbf{e}_{x,l} \cdot \mathbf{e}_{z,g} \end{pmatrix} + H_y \begin{pmatrix} \mathbf{e}_{y,l} \cdot \mathbf{e}_{x,g} \\ \mathbf{e}_{y,l} \cdot \mathbf{e}_{y,g} \\ \mathbf{e}_{y,l} \cdot \mathbf{e}_{z,g} \end{pmatrix} + H_z \begin{pmatrix} \mathbf{e}_{z,l} \cdot \mathbf{e}_{x,g} \\ \mathbf{e}_{z,l} \cdot \mathbf{e}_{y,g} \\ \mathbf{e}_{z,l} \cdot \mathbf{e}_{z,g} \end{pmatrix} \tag{3.28}$$

The last expression, needed to be calculated, is the inner product of the local and global basis vectors. In the present case, the local basis was derived from points in the global coordinate frame. Therefore, the inner product of any local basis vector with the i^{th} global basis vector is equivalent to the i^{th} component this local basis vector, denoted here by square brackets. Incorporating this knowledge into (3.28) yields

$$\mathbf{H}_g = H_x \begin{pmatrix} \mathbf{e}_{x,l}[0] \\ \mathbf{e}_{x,l}[1] \\ \mathbf{e}_{x,l}[2] \end{pmatrix} + H_y \begin{pmatrix} \mathbf{e}_{y,l}[0] \\ \mathbf{e}_{y,l}[1] \\ \mathbf{e}_{y,l}[2] \end{pmatrix} + H_z \begin{pmatrix} \mathbf{e}_{z,l}[0] \\ \mathbf{e}_{z,l}[1] \\ \mathbf{e}_{z,l}[2] \end{pmatrix} \quad (3.29)$$

where the Python convention of starting indexing at 0 is used. Having transformed the field of each sub-triangle into the global coordinate frame, the field of the original facet can be calculated by adding the fields of the sub-triangles with their respective sign, specified in Table 3.2.

3.3. Extracting the Inputs and Vectorization

The final section of this chapter treats the peculiarities of vectorizing the proposed code. Thanks to the utilization of NumPy arrays and NumPy universal functions, the entire algorithm presented in this chapter is already built for vectorized input. In the following, the way the input arguments of the field-calculating function are manipulated to enable the calculation of the magnetic field of a multi-facet system in many observation points will be presented. Prior to that, the input structure of the code listed in Appendix B will be discussed. This function has three inputs, which are consistent with the existing Magpylib package. These inputs are MAG, DIM and POSO and are discussed separately in the upcoming paragraphs.

MAG abbreviates *Magnetization*. Since the goal of this code is to calculate the magnetic field of facet bodies, the decision was made to select the magnetization of the simulated magnet as an input, instead of the *surface magnetic charge* σ_m - which is in most cases not directly accessible by the user. The derivation of σ_m from the magnetization is possible through equation (2.40) and the *outward pointing* surface normal vector $\mathbf{n}$ used in this equation will be

retrieved from the input DIM in the next paragraph. MAG is of size $(k \times 3)$, specifying the three components of the magnetization of the simulated body for the k facets of that body. If one wants to calculate the magnetic field of a multi-magnet setup, the MAG input is extended to include k_i entries along the 0^{th} dimension for the k_i facets of the i^{th} body.

DIM gives the dimensional input, i.e. the location of the triangle vertices and an implicit specification of the outward pointing surface normal vector. DIM is of size $k \times 4 \times 3$ and contains 4 points for each of the k facets. The first three points define the vertices of the triangle $\mathbf{T}_1, \mathbf{T}_2$ and $\mathbf{T}_3$. The fourth point was chosen to represent a point *lying within the magnet*, to which the facet belongs. This approach was chosen for two reasons. One reason was, that knowing one point inside the magnet provides sufficient information[4] to deduce the *outward pointing surface normal vector* $\mathbf{n}$. The second one was, that the position of a point lying within the magnet is easier to specify for the end user than the *outward pointing* surface normal $\mathbf{n}$. The calculation of $\mathbf{n}$, is a multi-step process. First, the *unoriented* surface normal $\mathbf{n}_u$ is calculated by taking the cross product of two edge vectors $(\mathbf{T}_1 - \mathbf{T}_2)$ and $(\mathbf{T}_1 - \mathbf{T}_3)$

$$\mathbf{n}_u = (\mathbf{T}_1 - \mathbf{T}_2) \times (\mathbf{T}_1 - \mathbf{T}_3) \,. \tag{3.30}$$

Then a vector, pointing *out* of the magnet, is defined, called $\mathbf{v}_{out}$. This vector starts at the point inside the magnet $\mathbf{P}_{in}$ and ends at one of the triangle vertices, in this case $\mathbf{T}_1$.

$$\mathbf{v}_{out} = \mathbf{T}_1 - \mathbf{P}_{in} \tag{3.31}$$

The outward pointing surface normal vector $\mathbf{n}$ can then be calculated by multiplying the normalized unoriented plane normal $\mathbf{n}_u$ with the sign of the inner product between $\mathbf{n}_u$ and the outward

[4] if the body is convex

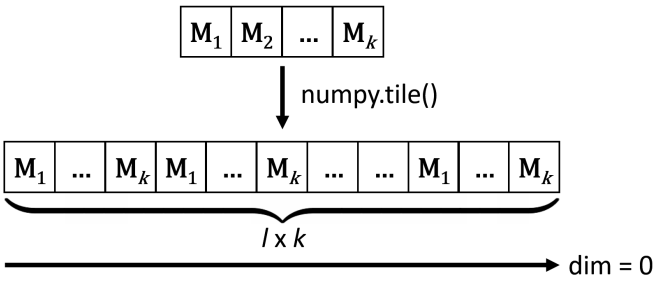
pointing vector from equation (3.31). Using the sign function σ this reads

$$\mathbf{n} = \sigma\left(\mathbf{n}_u \cdot \mathbf{v}_{out}\right) \frac{\mathbf{n}_u}{\|\mathbf{n}_u\|} \,. \tag{3.32}$$

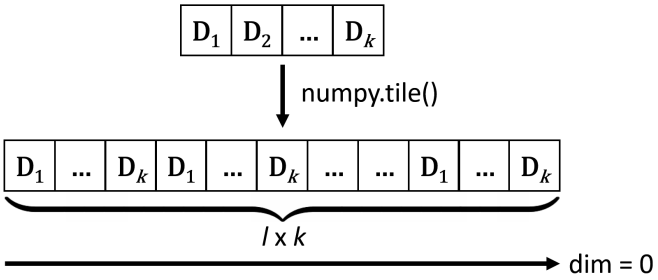
This way, by providing the three vertices of a triangle and a point inside the magnet as an input, the *outward pointing* surface normal vector, can be calculated. Furthermore, using (2.40) the surface magnetic charge σ_m of the facet can be computed.

POSO is the simplest of the three inputs. It is an $k \times 3$ array, listing the observation points.

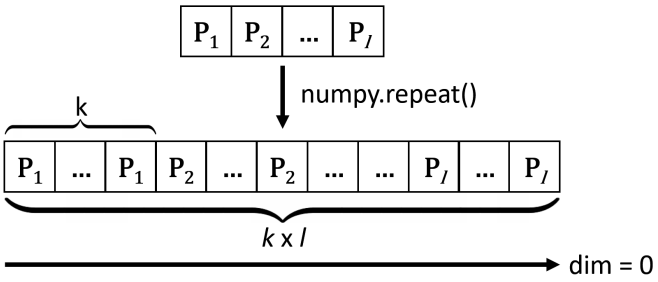
Vectorization In the vectorized case of multiple facets and multiple observation points, the inputs have to be modified so that the resulting arrays of MAG, DIM and POSO have the correct structure to be plugged into the the algorithm described in section 3.2. This restructuring has to ensure that in the end, the field of every facet in every observation point is calculated. In the general case, there are k facets belonging to potentially more than one magnet, and l observation points. Using the abbreviations M for MAG, D for DIM and P for POSO, the manipulation of the input arrays with *numpy.tile()* and *numpy.repeat()* is visualized in Figure 3.7. After the manipulation of all three inputs (MAG, DIM, POSO), the i^{th} entry along the 0^{th} dimension corresponds to the calculation of the magnetic field of one facet in one observation point. The whole arrays then describe the calculation of the magnetic field of every facets in all observation points.



(a) Using *numpy.tile()* to stack the input array describing the magnetization (MAG) l times along the 0^{th} dimension.



(b) Using *numpy.tile()* to stack the input array describing the facets (DIM) l times along the 0^{th} dimension.



(c) Using *numpy.repeat()* to repeat the input array describing the observation points (POSO) k times along the 0^{th} dimension.

Figure 3.7.: Manipulation of the input arrays to enable vectoriziation. k gives the number of facets, which contribute to the magnetic field and l is the number of observation points, in which the magnetic field is computed.

4. Validation

In this chapter the validation of the proposed code will be presented. It involves three distinct setups that will showcase the correctness and usefulness of the facet code. The first of these setups is a magnetic cuboid. This constitutes a case, in which analytical solutions are already available and therefore, the magnetic field calculated by the facet code can be verified. The second case is the computation for a triangular prism. With this setup the advantages of the newly introduced facet solution over existing analytical solutions will be shown. The third case is the comparison to a state of the art finite-element-method (FEM) simulation, describing a complex array of permanent magnets. This should provide further confidence into the general applicability of the facet solution and give an estimate of the performance increase that is to be expected.

4.1. Validation in the Cubic Magnet Case

The first configuration in which the proposed facet code was validated is an ideal permanent magnet[1] of cubic form. The facet solution was compared to Magpylib [14], which represents an approach using analytical formulas for cuboid magnets. In preparation of the facet solution, the surface of this cube was divided into 12 right triangles. For the comparison of the calculated magnetic fields, it was chosen to define a set of observation points, located in a plane above the magnet. In each of these observation points the relative

[1] without demagnetization

deviation of the norm of the magnetic field and the angle between the fields calculated by both solutions was evaluated. Furthermore, a simple timing of the computation time was performed.

4.1.1. Setup of the Cubic Magnet

Since cuboid magnets are already included in the Magpylib package, the calculation of the reference solution only needed the creation of a *Box* magnet source with the appropriate geometric parameters and magnetization. In this case the dimensions of the magnet were set to

$$DIM = (1, 1, 1)\,\text{mm}\ ,$$

and the magnetization was defined along the +z axis

$$MAG = (0, 0, 1000)\,\text{mT}\ .$$

The optional parameters were not set, which means that the center of the cube was in the origin and no rotations were applied. The plane of the observation points was chosen to be the $z = 1\,\text{mm}$ plane, with the x- and y-coordinates of the observations point lying on an equally spaced grid. The grid reached from -5 mm to +5 mm and had 50 grid points along each axis. One slight adaptation from a perfectly symmetrical grid was that the x-coordinates were shifted by 10 nm in $+x$ direction to avoid a bug in the code. How this bug manifests will be discussed in the upcoming section on known issues in section 4.4.

4.1.2. Comparison in the Cubic Magnet Case

In the following Figure 4.1 the relative deviation from the Magpylib cuboid solution and the angle between the two results is shown.

The relative deviation of the norm of the magnetic fields $\mathbf{H}$ was calculated by

$$\Delta_{rel} = \frac{\left\|\mathbf{H}_{facet} - \mathbf{H}_{magpylib}\right\|}{\left\|\mathbf{H}_{magpylib}\right\|} . \tag{4.1}$$

The angle between the two solutions was calculated by

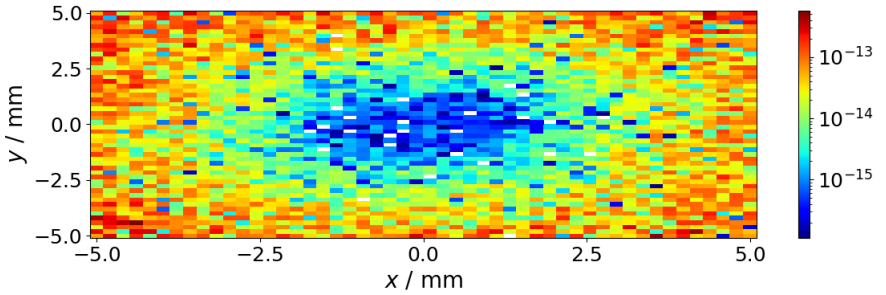
$$\alpha = \arccos\left(\frac{\mathbf{H}_{magpylib} \cdot \mathbf{H}_{facet}}{\| \mathbf{H}_{magpylib} \| \| \mathbf{H}_{facet} \|}\right) . \tag{4.2}$$

Since the angular deviation is displayed on a logarithmic scale, the absolute value of α was taken before plotting.

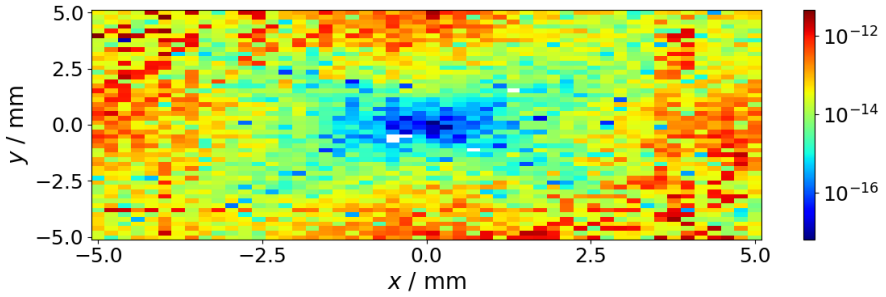
These figures show, that the result obtained by the facet solution is in good agreement with the already published Magpylib cuboid solution. An upper limit for the relative deviation of the total magnetic field in the examined plane is 10^{-12}. The orientational deviation in this setup is smaller than 10^{-11}. Moreover, the time it took to perform the calculation of the magnetic fields was extracted, using the *perf_counter* function form the *time* library. Only the code line with the call to the field calculating function was timed. For this configuration the facet code was slower than the Magpylib code by about a factor of 50. This performance result was to be expected, since the facet code involves many geometric operations.

4.2. Validation in the Triangular Prism Case

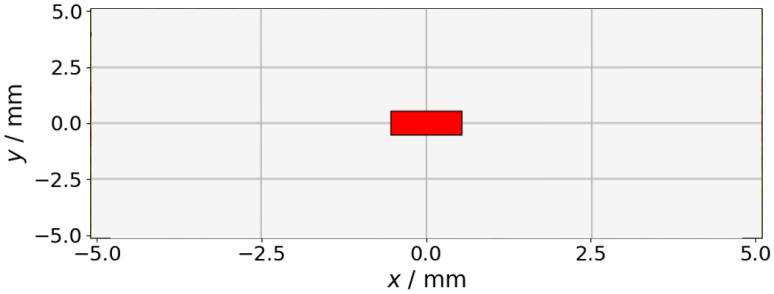
After showing that the newly introduced facet solution yields results in accordance with the already published Magpylib solution, the next step was to show its advantages over the cuboid solution. Therefore, the field of a prism with a right triangle as a base was calculated. For the facet solution, the surface of the prism was subdivided into 8 right triangles. For the Magpylib solution,



(a) Relative difference in the norm of the magnetic field on a logarithmic scale.



(b) Angular difference between the magnetic fields on a logarithmic scale.



(c) Top view of magnetic cube. Note the unequal axis representation.

Figure 4.1.: Comparison of the difference in the norm 4.1a and the orientation 4.1b between the magnetic fields calculated by the Magpylib cuboid solution and the proposed facet code for the cubic case. White areas indicate identical results. In Subfigure 4.1c the magnetic cube is depicted for reference.

the prism was approximated by n cuboids. The comparison not only involved the evaluation of the difference in the norm and

direction of the magnetic field, but also a convergence analysis of the approximation by cuboids. Furthermore, each step of this analysis was compared to the performance of the facet solution.

4.2.1. Setup of the Triangular Prism Magnet

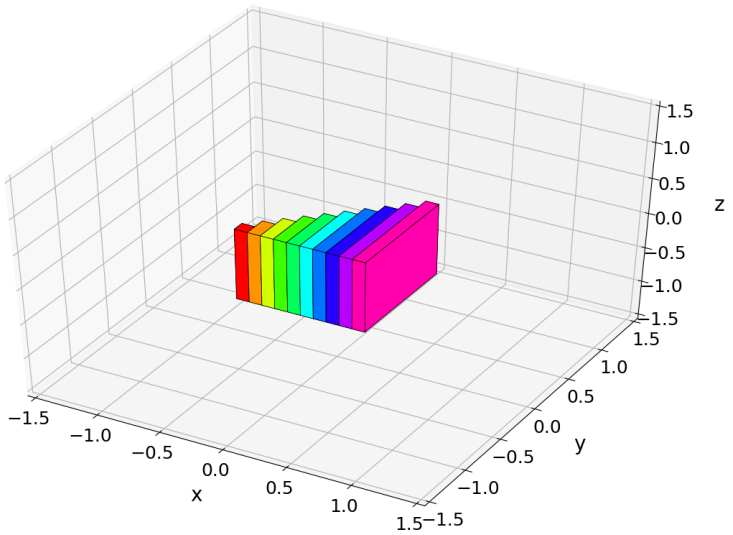
As mentioned above a permanent magnet with the shape of a triangular prism was simulated. The geometry of the prism was defined by the side lengths of the right base triangle and the height. For this comparison the side lengths of the base were chosen to be 1 mm each, and the height of the prism was also set to 1 mm. Once again the magnetization was defined as 1000 mT in $+z$ direction and the center of the prism was located at $(+\frac{1}{6}, -\frac{1}{6}, 0)$ mm. An approximation of the prism by 10 cuboids is shown in figure 4.2.

4.2.2. Convergence Analysis Triangular Prism

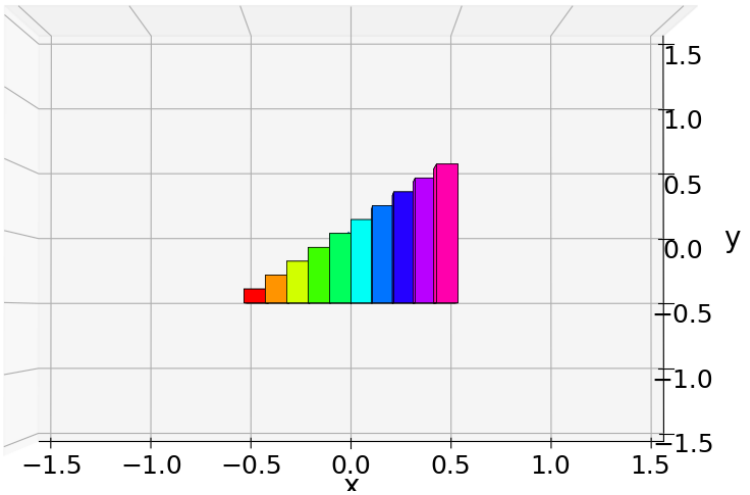
The first step in this comparison was to analyse the convergence of the approximation by cuboids. Therefore, a line above the magnet was defined and the norm of the field along this line was calculated, once by the facet solution and once for each discrete setup. Then the relative difference to the facet solution was calculated by

$$\Delta_n = \frac{\left\|\mathbf{H}_n - \mathbf{H}_{facet}\right\|}{\left\|\mathbf{H}_{facet}\right\|} \tag{4.3}$$

with the subscript n denoting the number of cuboids involved in the approximation. This is a suitable convergence criterion, since it was shown in section 4.1.2 that the facet solution gives the correct result for the field calculation. With increasing number of cuboids the approximation should converge to the value of the facet solution, and Δ_n should fall below $\approx 10^{-10}$. For this comparison the x-coordinates of the observation points were equally spaced between -5 mm and +5 mm with the y- and z-coordinate fixed to 0 and +1 mm



(a) 3D view of the approximation of the triangular prism by $n = 10$ cuboids.



(b) Top view of the approximation of the triangular prism by $n = 10$ cuboids.

Figure 4.2.: Approximation of a triangular prism by $n = 10$ cuboids.

respectively. In figure 4.3 the relative difference from the facet solution is plotted for increasingly fine approximations. Each line is labeled with the number of cuboids used in the approximation, n.

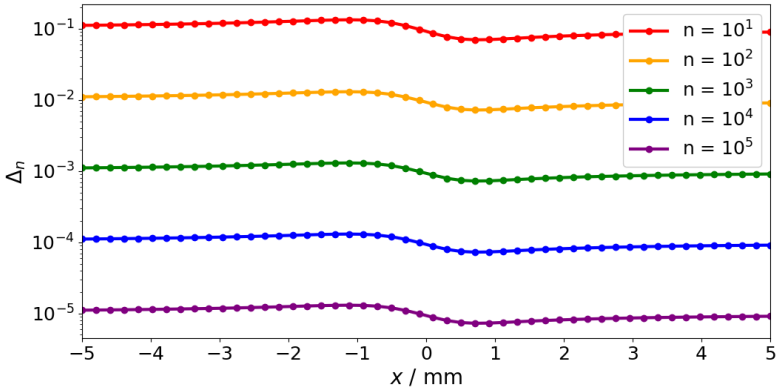


Figure 4.3.: Relative deviation from the facet solution for several degrees of discretization. n is the number of cuboids involved in the discretization.

The results shown in figure 4.3 are as expected: with a bigger number of cuboids approximating the prism, the deviation from the analytical solution gets smaller. Nevertheless, even with 10^5 cuboids, the deviation is still in the range of 10^{-5}, which indicates an insufficiently fine approximation. Furthermore, the indirect proportionality between n and the relative difference in this range is evident. Another important aspect of this comparison was the time it took each method to calculate the magnetic field. In the following bar diagram, figure 4.4, the timing of the field calculating function for each discrete setup, t_n is displayed in multiples of the duration of the facet calculation t_{facet}. This quotient is called τ

$$\tau = \frac{t_n}{t_{facet}} \, .$$

The performance comparison shown reflects the results of only one simulation run. Nevertheless, subsequent iterations of this comparison showed similar results and therefore, a sophisticated statistical analysis was omitted.

As is clearly visible, when aiming for a combination of maximum accuracy and performance for non-cuboid magnet shapes, the facet

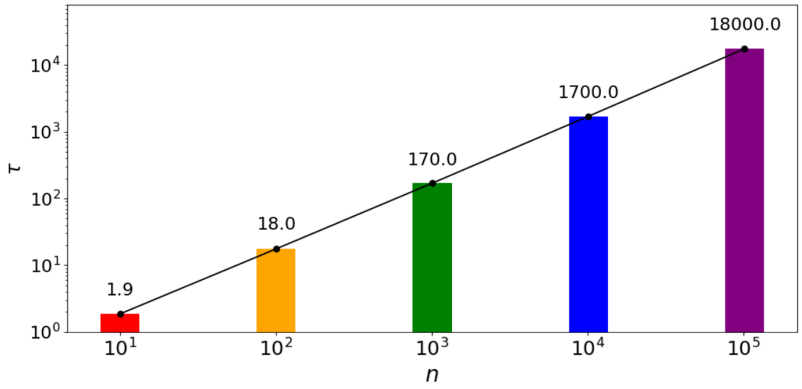


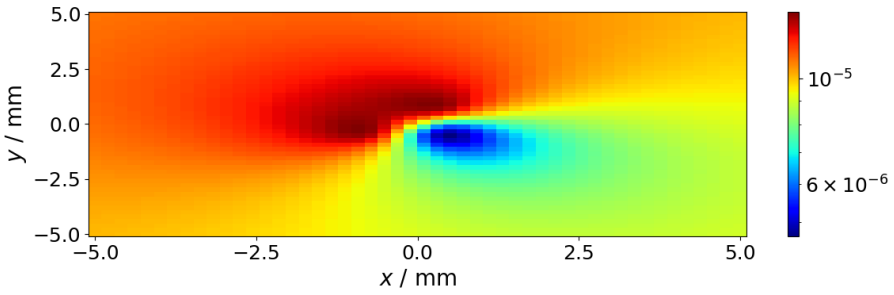
Figure 4.4.: Performance analysis of the cuboid discretization. Each bar gives the time it took to simulate an array of n cuboids, approximating the triangular prism in multiples of the computation time of the facet solution. The bars are annotated by the rounded values of the quotient τ.

solution introduced in this thesis prevails over the existing cuboid solution. Furthermore, it also allows for more complex shapes of magnets, which is a major benefit.

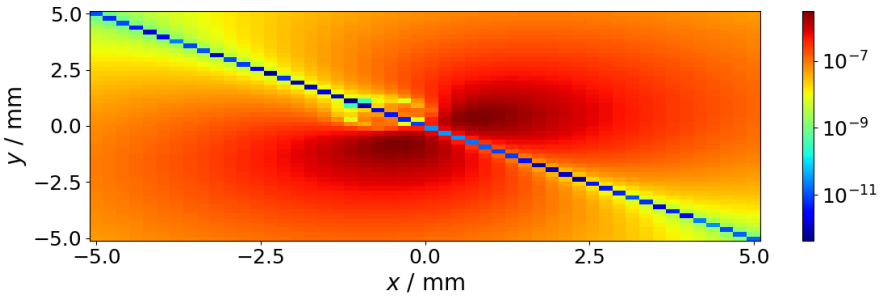
4.2.3. Comparison in the Triangular Prism Case

For $n = 10^5$ a more detailed analysis of the deviation from the facet solution was performed. In figure 4.5 the relative and orientational deviation of the discretized-cuboid solution to the facet solution are shown. The 50×50 observation points in which the fields were calculated were located on the same shifted grid as in the cubic case. The x- and y-coordinate of the observation points were equally spaced between -5 and +5 mm, while the z-coordinate was fixed to 1 mm. Again the x-coordinate was shifted by 10 nm to avoid a special case.

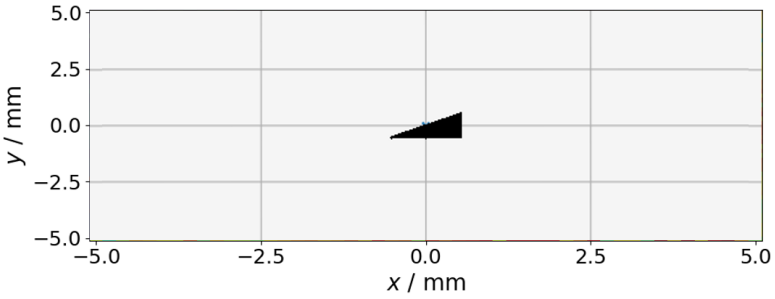
As was to be expected after the convergence analysis, the relative difference from the approximation by $n = 10^5$ cuboids to the facet



(a) Relative difference in the norm of the magnetic field.



(b) Angular difference between the magnetic fields.



(c) Top view of magnetic triangular prism. Note the unequal axis representation.

Figure 4.5.: Comparison of the difference in the norm 4.5a and the angular deviation 4.5b between the magnetic fields calculated by the Magpylib cuboid approximation with $n = 10^5$ and the proposed facet code for the triangular prism. In subfigure 4.5c the magnetic triangular prism is depicted for reference.

solution is in the range from 10^{-5} to 10^{-6}, which is significantly worse than in the cubic case (see Figure 4.1).

4.3. Complex Magnet Array

The last validation step presented in this thesis was the calculation of a complex magnet array and the comparison of the result to a state of the art FEM simulation. Before discussing the specific example, the following section briefly introduces the basic concepts of the finite element method.

4.3.1. Finite Element Method

The *finite element method*, FEM in short, is a numerical procedure, able to find approximate solutions for complex problems in physics and engineering. The basis for this method is the subdivision of space into a mesh of nodes and the numerical solution of a system of equations, resulting from this discretization.

A basic ingredient of the finite element method is the *finite element approximation*. It is based on the fact that one can approximate any desired function to arbitrary accuracy by a sufficiently large set of simple basis functions. In the case of FEM these functions are called *shape functions* and their defining characteristic is that they are only non-zero in a small region of space. This region is called the *finite element*. The center of each finite element lies on one node and it is bound by the nearest neighbors of this node. For each node a shape function is defined which has to fulfill two conditions. First, the value of the function has to be 1 at its node and secondly it has to be zero on every other node. A visualization of a one dimensional case with linear shape functions is shown in figure 4.6.

Having defined such a basis of shape functions, it is possible to approximate any function $\varphi(\mathbf{r})$ in a given region by a linear combination of these form functions $\psi_i(\mathbf{r})$ with suitable weights ω_i

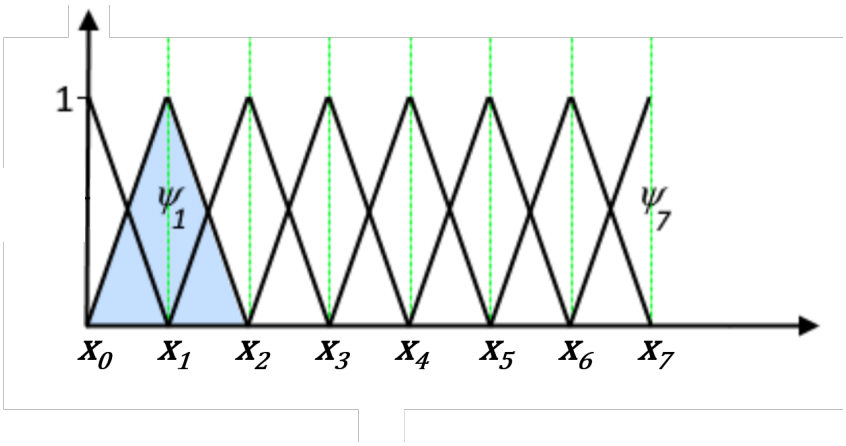


Figure 4.6.: Set of shape functions ψ_i on several nodes x_i. Adapted from Comsol multiphysics cycopedia [49]

$$\varphi(\mathbf{r}) \approx \sum_i \omega_i \, \psi_i(\mathbf{r}). \tag{4.4}$$

Each of these form functions $\psi_i(\mathbf{r})$ corresponds to a node in the mesh and fulfills the condition listed above. To achieve a better approximation, the number of nodes has to be increased, which is called *mesh refinement*. Having defined a set of nodes and shape functions, the unknown quantities left in equation (4.4) are the weights ω_i. Inspection of formula (4.4) at an arbitrary node j reveals that the weight ω_j has to be equal to the value of the desired function at this node $\varphi(\mathbf{r}_j)$.

$$\varphi(\mathbf{r}_j) \approx \sum_i \omega_i \, \psi_i(\mathbf{r}_j) \tag{4.5}$$

$$\varphi(\mathbf{r}_j) \approx \sum_i \omega_i \, \delta_{ij} = \omega_j \tag{4.6}$$

Therefore, the problem of correctly approximating the original function reduces to finding the weights of this linear combination of basis functions.

In order to solve this in the context of physical systems, one has to first discuss the *weak formulation* of differential equations. In the following the introduction of the weak formulation is motivated, and then a specific example is discussed. Often the derivation of the differential equation is too strict, in the sense that it excludes physically possible solutions on the basis of insufficient differentiability [50]. In real systems, such discontinuities in a derivative can be caused by different material parameters when crossing a material boundary. The physical solution to such a problem will not satisfy the differential equation and therefore, a less strict version of the equations describing the system is needed. The weak formulation of a differential equation exactly does that. To attain the weak form of a differential equation, one has to multiply the equation by a so called test function and thereafter integrate over the whole region of interest. By doing so one loosens the requirement of the solution to fulfill the differential equation in every point, and replaces it by the condition that the solution has to fulfill the equation in an integral sense [49]. As an example for this procedure one can examine the combination of Gauß' law for magnetism (2.23), the constitutive relation of magnetism (2.27) and the definition of the magnetic scalar potential Φ_m (2.33)

$$-\nabla \cdot (\mu_0(\nabla\Phi_m + \mathbf{M})) = 0 \tag{4.7}$$

with $\mathbf{M}$ being the magnetization. To find the weak formulation of this differential equation, one has to multiply equation (4.7) with a test function φ and integrate over the volume of interest Ω.

$$\int_\Omega -\nabla \cdot (\mu_0(\nabla\Phi_m + \mathbf{M}))\varphi \,\mathrm{d}V = 0 \tag{4.8}$$

The left side can be integrated by parts to transfer the derivative to the test function. This, with additional application of Gauß' theorem to the resulting second term gives

$$\int_{\Omega} (\mu_0(\nabla\Phi_m + \mathbf{M})) \cdot \nabla\varphi \, \mathrm{d}V + \int_{\partial\Omega} -(\mu_0(\nabla\Phi_m + \mathbf{M})) \cdot \mathbf{n}) \, \varphi \, \mathrm{d}S = 0 \tag{4.9}$$

This is the weak formulation of the differential equation (4.7). A magnetic scalar potential Φ_m, which fulfills this equation for every suitable test function, is called *weak solution*. In this case the *Galerkin method* can be applied, which works by expressing the desired solution Φ_m and the test function φ in terms of the same basis ψ_i. Explicitly these definitions read

$$\Phi_m(\mathbf{r}) \approx \sum_i \Phi_{m,i} \, \psi_i(\mathbf{r}) \ , \tag{4.10}$$

$$\varphi(\mathbf{r}) \approx \sum_i \varphi_i \, \psi_i(\mathbf{r}) \ . \tag{4.11}$$

Equation (4.9) has to hold for arbitrary test functions φ. Because of equation (4.11), this requirement is fulfilled if it holds for every basis functions ψ_j separately. Inserting (4.10) and (4.11) into equation (4.9) yields

$$\begin{aligned} &\sum_i \int_{\Omega} (\mu_0(\Phi_{m,i}\nabla\psi_i + \mathbf{M})) \cdot \nabla\psi_j \, \mathrm{d}V \\ +&\sum_i \int_{\partial\Omega} -(\mu_0(\Phi_{m,i}\nabla\psi_i + \mathbf{M})) \cdot \mathbf{n}) \, \psi_j \, \mathrm{d}S = 0 \end{aligned} \tag{4.12}$$

Evaluating (4.12) on each of the m nodes constitutes one equation in a system of $m \times m$ unique equations. The unknowns in these equations are the $\Phi_{m,i}$ s - the values of the desired solution function on the respective node. The only thing left is to solve this system of equations. While not being a trivial task, it is in many cases possible to apply numerical methods to solve such a system of equations. Additional introduction and discussion of the finite element method can be found in [51] and [52].

4.3.2. Setup of the Complex Magnet Array

Having introduced the basics of FEM, the last validation step of the facet solution can be discussed. For that reason a complex magnet array, spelling "SAL+TUG", was defined. Each of the letters constitutes an ideal permanent magnet with a magnetization of 1000 mT in +z-direction. A depiction of this setup is given in figure 4.7. For the facet solution, the surface of this magnet array was divided into a total of 236 facets. The FEM simulation was performed with *Ansys© Maxwell*.

4.3.3. Comparison in the Complex Magnet Array Case

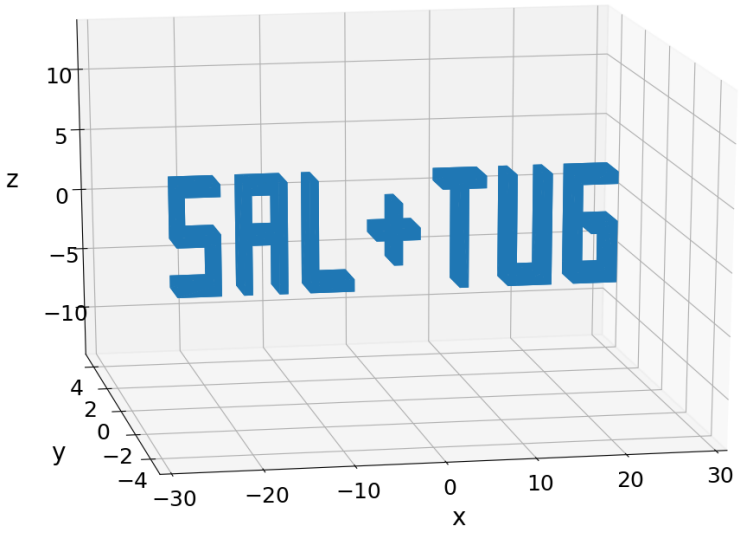
To compare the facet solution to the FEM simulation it was decided to compute the norm of the magnetic field along a line above the array. The line spanned from -30 mm to +30 mm in x-direction and had y- and z-coordinates of 0 mm and 4.61 mm respectively. Along this line 264 equally spaced observation points were defined. In figure 4.8, the result of both the FEM as well as the facet simulation are shown.

As can be seen, the two methods show good agreement, which further underscores the validity of the facet solution. A better understanding of the difference between the two solutions can be obtained by inspecting the relative difference. In figure 4.9 the relative difference, calculated by

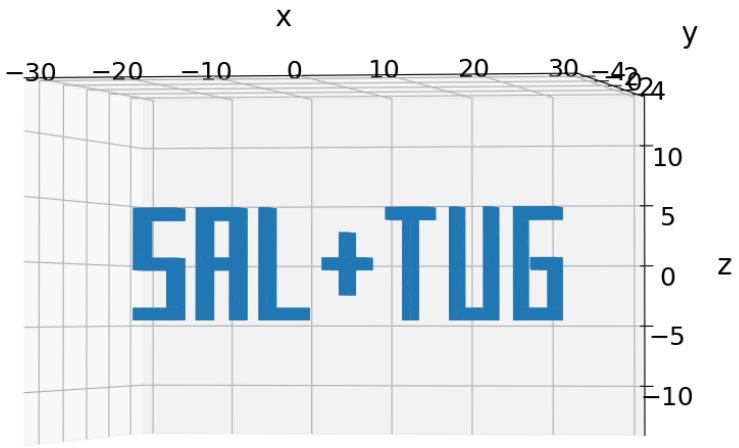
$$\Delta_r = \frac{\| \mathbf{H}_{facet} - \mathbf{H}_{FEM} \|}{\| \mathbf{H}_{facet} \|} \tag{4.13}$$

is shown.

This analysis shows that both methods agree withing 1%. It has to be noted, that the better interpretation of this difference is that the FEM is within 1% of the facet solution, since the facet solution



(a) View of the complex magnet array.



(b) View of the complex magnet array from a different angle.

Figure 4.7.: Depiction of the complex magnet array used in the simulation.

is based on analytical formulas. Accuracy was not the primary interest for this comparison. This was the performance of each

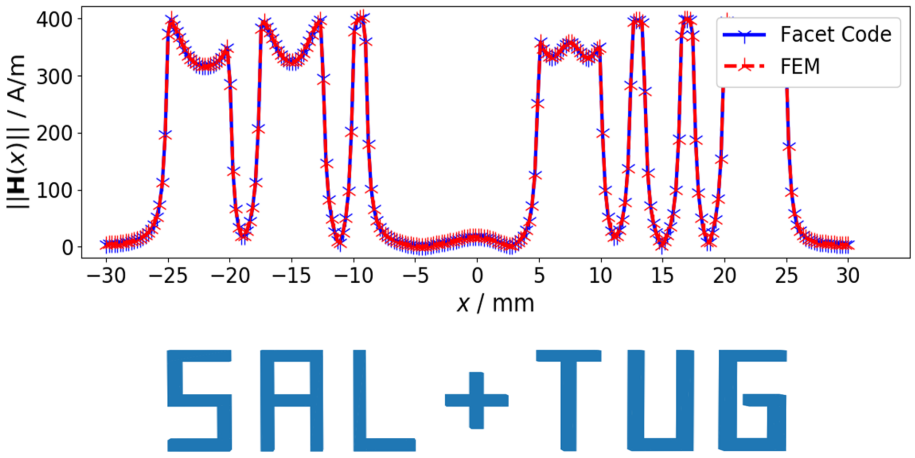


Figure 4.8.: Values of the norm of the magnetic field **H** on 264 points along the line $y = 0\,\text{mm}$, $z = 4.61\,\text{mm}$. Calculated by the proposed facet code (blue) and by Ansys© Maxwell (red). Below the plot of the norm of the magnetic field, a front view of the magnet array is given on the same x-positions for reference.

method, and in this regard the facet solution showed its benefits. One simulation run of the FEM took

$$t_{FEM} = 14\,\text{min}\ 17\,\text{s}\ ,$$

while the facet solution only took

$$t_{facet} = 0.51\,\text{s}\ .$$

For the presented case these results constitute a speedup by a factor of

$$\frac{t_{FEM}}{t_{facet}} \approx 1700\ . \tag{4.14}$$

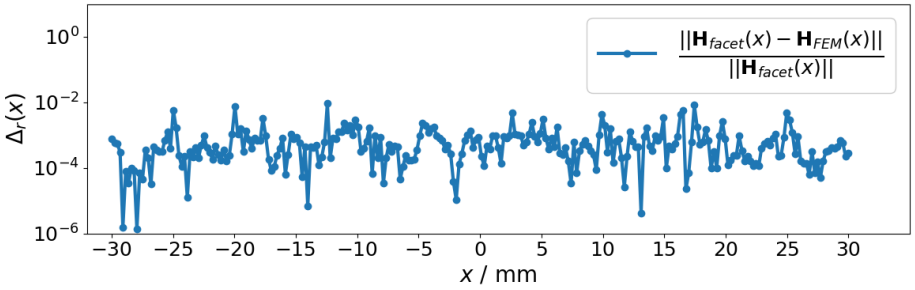


Figure 4.9.: Relative difference Δ_r between the FEM simulation and the facet solution, for 264 points along the line $y = 0\,\text{mm}$, $z = 4.61\,\text{mm}$.

A thorough discussion on how to interpret this performance increase is warranted by the fundamental difference of the two methods analyzed. In the following this discussion is started by contrasting the two methods and listing the performance affecting aspects of the examined system for each approach.

The performance of the FEM is closely tied to the desired quality of the approximating solution. As mentioned in Section 4.3.1 a more accurate solution is obtained if a finer mesh and therefore, more nodes are analysed. Since a larger number of nodes corresponds to a larger system of equations that has to be solved, achieving a more accurate FEM solution takes more time. Another aspect affecting the performance of the FEM in the discussed scenario is the the *air gap* between the observation points and the magnet. The number of nodes scales with the simulated volume, and since this volume increases with a larger air gap, the computation time becomes longer. In the presented case, the air gap was deliberately chosen to be very small, in order to minimize the disadvantage of the FEM.

The performance of the facet solution is dependent on the number of input facets and the number of observation points. As mentioned in section 3.3 for k facets and l observation points, the proce-

dure described in 3.2 has to be performed $k \times l$ times. Because the whole code is vectorized, the computation time is not proportional to the number of inputs. In Figure 4.10 the time it took the facet code to calculate the field of one facet in l observation points is shown, to emphasize this characteristic of the presented code. The final caveat needed to be addressed in the performance comparison is the shape of the magnet array in figure 4.7. This shape is very beneficial to the facet solution, since there are large even surfaces which are easily describable by triangles. When dealing with round surfaces, a similar problem like the one discussed in section 4.2 would occur. To accurately describe a round surface by triangles, many facets would be needed, which would result in a trade-off between computation time and minimizing the approximation error.

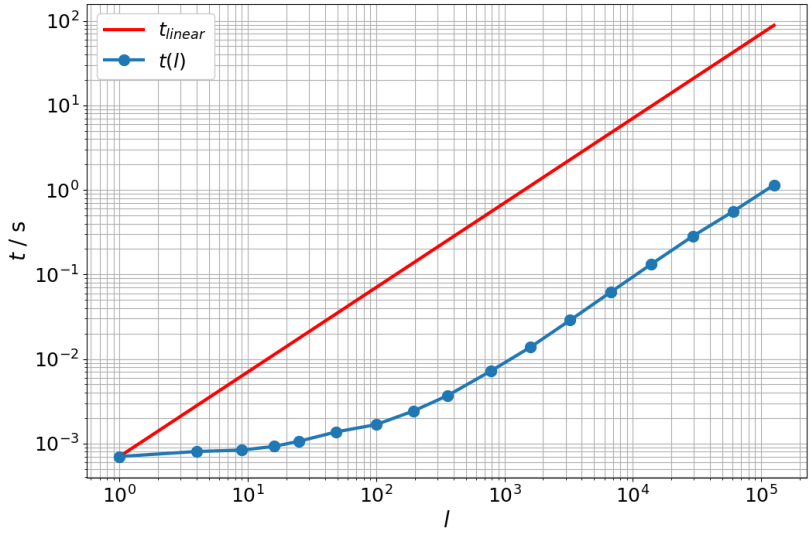


Figure 4.10.: Double-logarithmic plot showing the scaling of the computation time t. Computed was the magnetic field of one facet on a grid containing l points. In blue the timing of the vectorized code is shown $t(l)$. For reference the red line shows a linear increase t_{linear}, following formula (4.15).

The linear increase shown in Figure 4.10 represents the linear slope

$$t_{linear}(l) = l \cdot t(l = 0) \ . \tag{4.15}$$

The plot shows the scaling advantages of the vectorized code, and is in line with the findings in the paper introducing Magpylib [14, Fig.6(b)].

4.4. Known Issues

Before concluding this thesis, it is necessary to discuss known shortcomings of the introduced code. During the testing and validation of the results, two separate issues were identified.

In-plane Observation Point The first issue occurs, when the observation point lies in the plane of the facet. In such a case the distance from the observation point to the plane is zero. This leads to divergent terms in the equations for the **H**-field (2.43) to (2.45), which was already addressed in the original paper by Rubeck et al. [47]. For the case of $c = 0$, they write:

> The normal component of the magnetic field H_z is discontinuous in crossing the surface . It is reasonable to take $H_z = 0$ for $c = 0$. It must also be pointed out that the field components H_x and H_y are undefined for $c = 0$ (the plane of the polygon). The divergent terms cancel analytically when all the right triangles of a polygon are combined together. It was analytically verified on the case of the rectangle and numerically for other polygonal shapes (the divergent terms are simply not calculated).

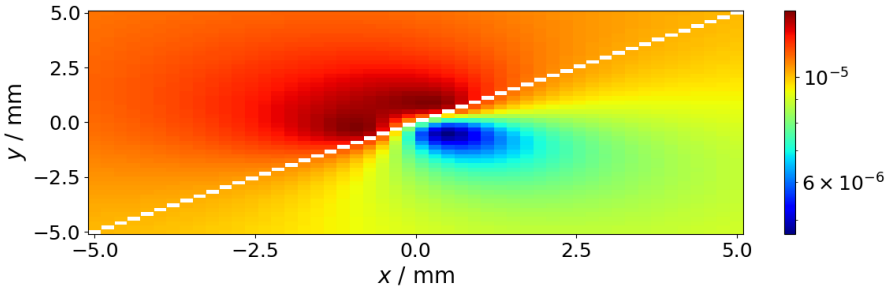
In the current implementation, presented in this work, this case was handled by returning a warning to the user. This approach was taken because the whole calculation, for possibly multiple facets and many observation points, is performed in a vectorized fashion.

This special case only renders the calculation in the specific observation point invalid, but does not effect other observation points. One way to correctly handle this special case without significant performance decrease, would be to employ Boolean indexing. However, this would add another layer of complexity to the code and unfortunately there was too little time to implement and thoroughly test this approach. Adding the capability to deal with in-plane observation points is planned for the future, and would increase the applicability of the proposed solution. An example where this issue manifests is the case of the triangular prism from section 4.2. In the presented validation the grid of the observation points was shifted slightly. In Figure 4.11 the calculation of the relative difference with equation for both the symmetrical as well as the shifted grid is shown.

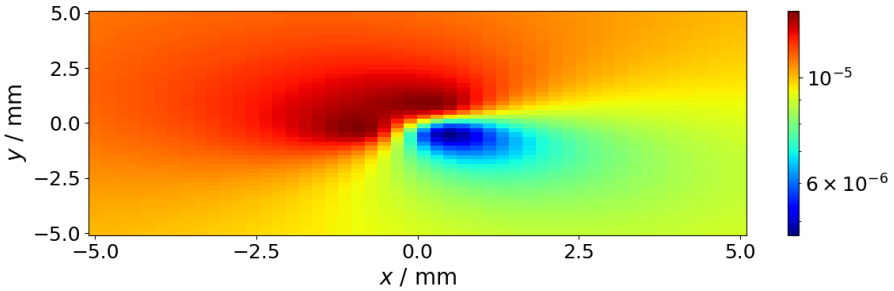
In this case, the observation points in the plane

$$y = \frac{b}{a}\, x \tag{4.16}$$

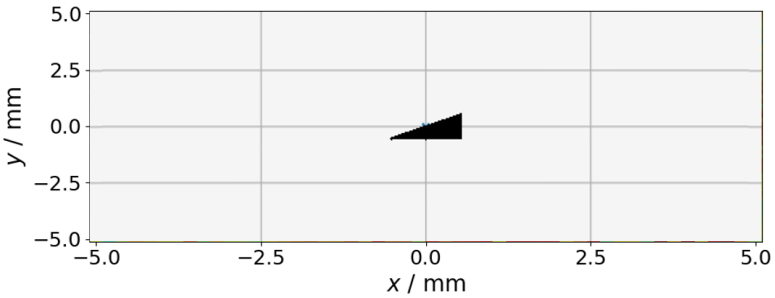
lie exactly in the plane of one of the facets, and therefore represent the special case, where $c = 0$, In this case the calculation fails, and several Python exceptions are raised, which was expected.



(a) Special case with in-plane observation point. Calculation on a perfectly symmetrical grid.



(b) Calculation on a equally spaced grid, but the x-component of the observation point is shifted by 10 nm.



(c) Location of the triangular prism.

Figure 4.11.: Visualization of the $c = 0$ special case. Subfigure 4.11a shows a perfectly symmetrical grid, while Subfigure 4.11b is shifted by 10 nm in $+x$-direction. The white points in figure are the points where the calculation failed. Subfigure 4.11c shows a top view of the triangular prism. Note the unequal axis representation.

Bug encountered during Validation As mentioned in the presented validation the x-coordinates of the observation points were

shifted slightly away from a perfectly symmetrical grid. Calculating the magnetic field of the cubic setup on a symmetrical grid, without the shift, resulted in subfigure 4.12a. For comparison the original calculation on the shifted grid is shown in subfigure 4.12b.

Some data points appear in white, because the relative difference vanishes and a logarithmic scale was used. Figures 4.12a shows, that the calculation tends to fail along the plane

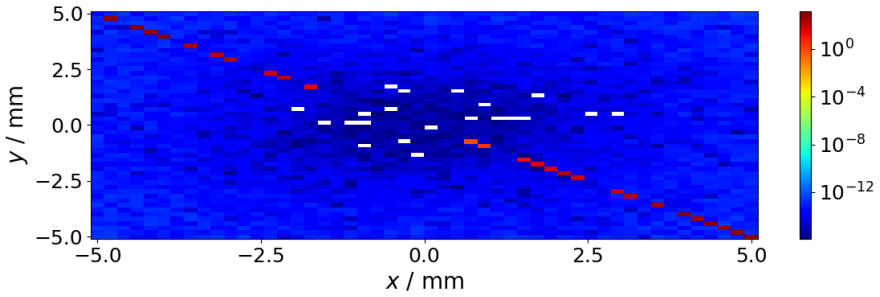
$$y = -\frac{b}{a}\,x \tag{4.17}$$

in the chosen coordinate frame. This behaviour was encountered for different parameters a and b. It is noteworthy, that the calculation produces an apparently correct result on the second diagonal

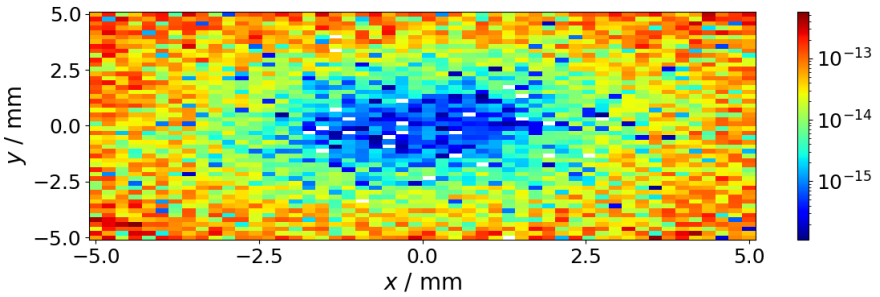
$$y = \frac{b}{a}\,x \tag{4.18}$$

and that no Python exceptions were raised in this case.

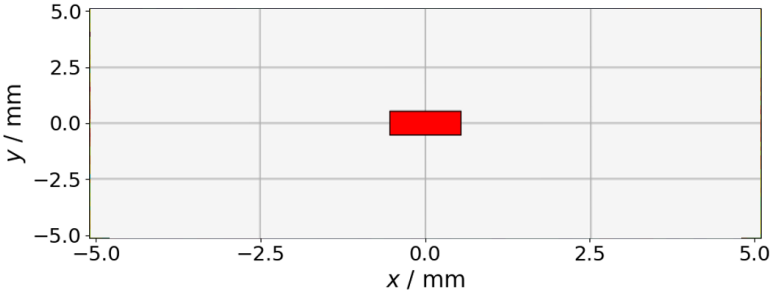
This concludes the discussion of the results and the validation of the proposed facet code. The next section will provide a summary and conclusion of the insights achieved in this thesis.



(a) Special case, which leads to false results. Calculation on a perfectly symmetrical grid.



(b) Calculation on a equally spaced grid, but the x-component of the observation point is shifted by 10 nm.



(c) Location of the cube

Figure 4.12.: Visualization of bug in the cubic magnet case. Subfigure 4.12a shows a perfectly symmetrical grid, while subfigure 4.12b is shifted by 10 nm in $+x-$direction. Note the different ranges in the two plots. The white points in figures are the points where relative difference was 0. Subfigure 4.11c shows a top view of the cube. Note the unequal axis representation.

5. Conclusion

This chapter presents a summary and discussion of the key findings of this thesis. It will focus on the insights gathered from the presented validation and provide context and interpretation.

The proposed method for the analytical calculation of the magnetic field of hard magnetic facet bodies is based on formulas (2.43) to (2.48), published by the group of Rubeck [47]. Based on this theoretical ground an algorithm is introduced, which by means of a geometric decomposition, is able to compute the magnetic field of a *homogeneously magnetically charged* right or acute triangle. By exploitation of the superposition principle and the use of efficient code through vectorization and the NumPy library, it was possible to write a function, that is able to perform the field computation of hard magnetic facet bodies in a rapid fashion.

The correctness of the results produced by the facet code were analysed via comparison to another already published analytical solution - the Magpylib *cuboid* [14]. This comparison is described in section 4.1.2 and figure 4.1 and shows the excellent accordance of the two methods. Although no extensive performance comparison was performed, a simple timing revealed a performance advantage of the Magpylib solution in the range of one order of magnitude. One source of the slightly inferior performance of the facet code is the necessity of the geometric decomposition. While every step in this decomposition is computationally light-weight, the amount of steps prior to the evaluation of field calculating functions (2.43) to (2.48) produces significant performance overhead cost. The time consumed by the geometric decomposition was analyzed by the use of *profiling* during code optimization. A detailed discussion of

the results of this profiling was beyond the scope of this work and is therefore not presented explicitly.

A showcase of the advantages of the newly introduced facet code over existing solutions, was the analysis of a magnetic triangular prism. This setup could only be approximated with the Magpylib cuboid solution and the investigation of this case revealed the advantages of the proposed facet code. While it is not possible to achieve an accurate and fast result with the cuboid approximation, the facet solution retains both the performance as well as the accuracy of the analytical formulas. In figure 4.4 the computation speed advantage of the facet solution over the cuboid approximation is shown. Figures 4.3 and 4.5 provide a deeper analysis in terms of accuracy.

The final validation step was the comparison of the facet solution to a state of the art finite element method simulation of a complex magnet array, which is shown in figure 4.7. In figures 4.8 and 4.9 a comparison of the calculated magnetic field of this array is shown. The data presented in these figures further underscore the validity of the fields computed by the proposed facet solution. Furthermore, for this exact setup, a performance analysis revealed, that the facet code outperformed the state of the art FEM simulation by a factor of ≈ 1700. At the end of section 4.3.3 some context is given on how to interpret this performance increase. This discussion addresses the fundamental difference between the two calculation methods and highlights aspects of the presented case that affected the performance. For the facet code, the scaling with the number of inputs was explicitly mentioned. With the context from section 3.3, figure 4.10 shows the impact of vectorized code on the scaling with more inputs. Section 4.4 gives an overview of the two edge cases, in which the proposed facet code is known to produce the wrong result. One of them represents a limitation of the underlying analytical formulas, while the second one is probably related to the implementation and needs further analysis.

In this thesis a Python function is presented that is able to calculate the magnetic field of hard magnetic facet bodies. Due to

performance oriented programming and the application of analytical formulas, the presented code provides fast computation times without sacrificing accuracy. Future research, built on the foundation of the proposed facet code, could enable improvements in magnetic system design and applications of computational magnetism. Which applications this could be, will be discussed in the next chapter - the outlook.

6. Outlook

In this final chapter, an outlook to possible applications of the proposed facet code will be given.

As mentioned in the chapter **Motivation**, the availability of high-performance code to calculate the magnetic field of facet bodies enables new applications in the field of magnetic system design. Beyond the calculation of complex magnet shapes, the presented facet code could be used to expand the applicability of analytical methods beyond the case of ideal homogeneously magnetized permanent magnets. Using the facet solution it is possible to define small polyhedral elements, which discretize a given magnet volume. In that way, the calculation of inhomogeneous magnetization can be treated. Another application of this discretization would be the treatment of demagnetization. Given a magnetic material, build from discrete elements, for which the remnant permeability is known, the self interaction and therefore, the demagnetization can be approximated.

Appendix

Appendix A.

Derivation Rubeck Formulas

In chapter 2.3 formula (2.42) for the magnetic scalar potential Φ_m was derived. Since one is interested in the magnetic field, not the scalar potential, equation (2.33) can be applied

$$\mathbf{H} = \nabla \Phi_m \ .$$

Then, according to Rubeck et. al. [47] the three components of $\mathbf{H}$ for type A triangles can be written as:

$$\begin{aligned} H_{x,A}(\mathbf{r}) =& \frac{\sigma_m}{4\pi\mu_0} \iint_S \frac{\mathbf{r}\cdot\mathbf{i}}{r^3} \mathrm{d}S \\ =& \frac{\sigma_m}{4\pi\mu_0} \int_{x=0}^{x=a} \int_{y=0}^{y=bx/a} \frac{-x}{(x^2+y^2+c^2)^{3/2}} \mathrm{d}x\mathrm{d}y \ , \end{aligned} \tag{A.1}$$

$$\begin{aligned} H_{y,A}(\mathbf{r}) =& \frac{\sigma_m}{4\pi\mu_0} \iint_S \frac{\mathbf{r}\cdot\mathbf{j}}{r^3} \mathrm{d}S \\ =& \frac{\sigma_m}{4\pi\mu_0} \int_{x=0}^{x=a} \int_{y=0}^{y=bx/a} \frac{-y}{(x^2+y^2+c^2)^{3/2}} \mathrm{d}x\mathrm{d}y \ , \end{aligned} \tag{A.2}$$

$$\begin{aligned} H_{z,A}(\mathbf{r}) =& \frac{\sigma_m}{4\pi\mu_0} \iint_S \frac{\mathbf{r}\cdot\mathbf{k}}{r^3} \mathrm{d}S \\ =& \frac{\sigma_m}{4\pi\mu_0} \int_{x=0}^{x=a} \int_{y=0}^{y=bx/a} \frac{c}{(x^2+y^2+c^2)^{3/2}} \mathrm{d}x\mathrm{d}y \ , \end{aligned} \tag{A.3}$$

with $\mathbf{r} = -x\,\mathbf{i} - y\,\mathbf{j} + c\,\mathbf{k}$. In the following the integration of each of the components will be performed step by step. For $H_{x,A}$ one starts by pushing the prefactor to the left side:

$$\frac{4\pi\,\mu_0\,H_{x,A}(\mathbf{r})}{\sigma_m} = \int_{x=0}^{x=a}\int_{y=0}^{y=bx/a} \frac{-x}{(x^2+y^2+c^2)^{3/2}}\mathrm{d}y\mathrm{d}x\ . \tag{A.4}$$

Here two integrals can be defined and solved subsequently

$$I_{1,x}(x) = \int_{y=0}^{y=bx/a} \frac{1}{(x^2+y^2+c^2)^{3/2}}\mathrm{d}y\ , \tag{A.5}$$

$$I_{2,x} = \int_{x=0}^{x=a} -x\,I_{1,x}(x)\mathrm{d}x\ . \tag{A.6}$$

The integral $I_{1,x}$ can be solved by substituting

$$y = \sqrt{x^2+c^2}\,\tan(u)\ . \tag{A.7}$$

Therefore, the integration measure and the limits of the integration can be written as

$$\mathrm{d}y = \sqrt{x^2+c^2}\,(1+\tan^2(u))\mathrm{d}u\ , \tag{A.8}$$

$$u(y=0) = 0\ , \tag{A.9}$$

$$u\left(y = \frac{b\,x}{a}\right) = \arctan\left(\frac{b\,x}{a\,\sqrt{x^2+c^2}}\right)\ . \tag{A.10}$$

Putting this substitution back into $I_{1,x}$ yields

$$I_{1,x} = \int_0^{\arctan\left(\frac{b\,x}{a\sqrt{x^2+c^2}}\right)} \frac{\sqrt{x^2+c^2}\,(1+\tan^2(u))}{((x^2+c^2)+(x^2+c^2)\,\tan^2(u))^{3/2}}\,\mathrm{d}u\ . \tag{A.11}$$

Collecting terms and expressing the tangent by the quotient of sine and cosine results in

$$I_{1,x} = \frac{1}{x^2+c^2}\int_0^{\arctan\left(\frac{b\,x}{a\sqrt{x^2+c^2}}\right)} \frac{1}{\left(1+\frac{\sin^2(u)}{\cos^2(u)}\right)^{1/2}}\,\mathrm{d}u\ . \tag{A.12}$$

Expanding the fraction in the denominator, using the trigonometric identity then resolving the compound fraction finally leads to

$$I_{1,x} = \frac{1}{x^2+c^2}\int_0^{\arctan\left(\frac{b\,x}{a\sqrt{x^2+c^2}}\right)} \cos(u)\,\mathrm{d}u\ . \tag{A.13}$$

Integrating and evaluating the boundaries yields

$$I_{1,x} = \frac{1}{x^2+c^2}\,\sin\left(\arctan\left(\frac{b\,x}{a\sqrt{x^2+c^2}}\right)\right)\ . \tag{A.14}$$

Using

$$\sin(\arctan(x)) = \frac{x}{\sqrt{1+x^2}}\ , \tag{A.15}$$

and plugging this into equation (A.14) gives

$$I_{1,x} = \frac{1}{x^2+c^2}\,\frac{\frac{b\,x}{a\sqrt{x^2+c^2}}}{\sqrt{1+\frac{b^2\,x^2}{a^2\,(x^2+c^2)}}}\ . \tag{A.16}$$

Collecting terms simplifies the equation to

$$I_{1,x}(x) = \frac{b\,x}{(x^2+c^2)\,\sqrt{(a^2+b^2)\,x^2+a^2c^2}}\,. \tag{A.17}$$

Now that the first integration step is finished, the second integral can be calculated. Inserting (A.17) into (A.6) gives the starting point for the evaluation of the second integral.

$$I_{2,x} = -b\int_{x=0}^{x=a} \frac{x^2}{(x^2+c^2)\,\sqrt{(a^2+b^2)\,x^2+a^2c^2}}\mathrm{d}x\,. \tag{A.18}$$

This integral can again be solved by substitution. Replacing

$$\frac{x}{\sqrt{(a^2+b^2)\,x^2+a^2c^2}} = u\,, \tag{A.19}$$

and therefore

$$x = \frac{acu}{\sqrt{1-(a^2+b^2)\,u^2}}\,, \tag{A.20}$$

$$u(x=0) = 0\,, \tag{A.21}$$

$$u(x=a) = \frac{1}{\sqrt{a^2+b^2+c^2}}\,, \tag{A.22}$$

and

$$\mathrm{d}x = \frac{\left((a^2+b^2)\,x^2+a^2c^2\right)^{3/2}}{a^2c^2}\mathrm{d}u\,. \tag{A.23}$$

Combining the substitution of $\mathrm{d}x$ with the integral $I_{2,x}$ yields

$$I_{2,x} = -b \int_{x=0}^{1/\sqrt{a^2+b^2+c^2}} \frac{x^2 \left(\left(a^2+b^2\right) x^2 + a^2c^2\right)^{3/2}}{a^2c^2 \left(x^2+c^2\right) \sqrt{\left(a^2+b^2\right) x^2 + a^2c^2}} \, \mathrm{d}u \; . \tag{A.24}$$

Cancellation and pulling constant factors in front of the integral simplifies to

$$I_{2,x} = -\frac{b}{a^2c^2} \int_{x=0}^{1/\sqrt{a^2+b^2+c^2}} \frac{x^2 \left(\left(a^2+b^2\right) x^2 + a^2c^2\right)}{x^2+c^2} \, \mathrm{d}u \; . \tag{A.25}$$

Adding 0 to the numerator in the form of $\pm c \left(\left(a^2+b^2\right) x^2 + a^2c^2\right)$, enables splitting the integral into two parts:

$$\begin{aligned} I_{2,x} = -\frac{b}{a^2c^2} \int_{x=0}^{1/\sqrt{a^2+b^2+c^2}} & \frac{\left(x^2+c^2\right) \left(\left(a^2+b^2\right) x^2 + a^2c^2\right)}{x^2+c^2} \, \mathrm{d}u \\ & + \frac{b}{a^2c^2} \int_{x=0}^{1/\sqrt{a^2+b^2+c^2}} \frac{c^2 \left(\left(a^2+b^2\right) x^2 + a^2c^2\right)}{x^2+c^2} \, \mathrm{d}u \; . \end{aligned} \tag{A.26}$$

Canceling $x^2 + c^2$ in the first term and c^2 in the second one, combined with substituting x from (A.20) yields

$$\begin{aligned} I_{2,x} = -\frac{b}{a^2c^2} \int_{x=0}^{1/\sqrt{a^2+b^2+c^2}} & \left(\frac{a^2c^2u^2(a^2+b^2)}{1-u^2(a^2+b^2)} + a^2c^2 \right) \mathrm{d}u \\ & + \frac{b}{a^2} \int_{x=0}^{1/\sqrt{a^2+b^2+c^2}} \frac{\frac{a^2c^2u^2(a^2+b^2)}{1-u^2(a^2+b^2)} + a^2c^2}{\frac{a^2c^2u^2}{1-u^2(a^2+b^2)} + c^2} \, \mathrm{d}u \; . \end{aligned} \tag{A.27}$$

Cancellation of a^2c^2 in both integrals leads to

$$I_{2,x} = -b \int_{x=0}^{1/\sqrt{a^2+b^2+c^2}} \left(\frac{u^2(a^2+b^2)}{1-u^2(a^2+b^2)} + 1 \right) \mathrm{d}u \\ +b \int_{x=0}^{1/\sqrt{a^2+b^2+c^2}} \frac{\frac{u^2(a^2+b^2)}{1-u^2(a^2+b^2)} + 1}{\frac{a^2u^2}{1-u^2(a^2+b^2)} + 1} \mathrm{d}u \ . \tag{A.28}$$

Resolving the fractions and collecting terms simplifies to

$$I_{2,x} = b \int_{x=0}^{1/\sqrt{a^2+b^2+c^2}} \left(\frac{-1}{1-u^2(a^2+b^2)} + \frac{1}{1-u^2b^2} \right) \ . \tag{A.29}$$

Both the terms have the form

$$I = \int \frac{1}{1-m^2u^2} \mathrm{d}u \ , \tag{A.30}$$

which can be factorized and integrated like

$$I = \int \frac{1}{1-m^2u^2} \, \mathrm{d}u \tag{A.31}$$

$$= \int \frac{1}{(1+mu)\,(1-mu)} \, \mathrm{d}u \tag{A.32}$$

$$= \frac{1}{2} \int \frac{1}{1+mu} + \frac{1}{1-mu} \, \mathrm{d}u \tag{A.33}$$

$$= \frac{1}{2m} \ln \left(\frac{1+mu}{1-mu} \right) + C \ . \tag{A.34}$$

The integration of (A.29) and evaluation within its limits results in

$$I_{2,x} = \frac{-b}{2D_{ab}} \ln\left(\frac{D_{abc} + D_{ab}}{D_{abc} - D_{ab}}\right) + \frac{1}{2} \ln\left(\frac{D_{abc} + b}{D_{abc} - b}\right) , \qquad \text{(A.35)}$$

with the abbreviations

$$\begin{aligned} D_{abc} &= \sqrt{a^2 + b^2 + c^2} , \\ D_{ab} &= \sqrt{a^2 + b^2} , \\ D_{ac} &= \sqrt{a^2 + c^2} . \end{aligned}$$

Therefore, equation (A.1) evaluates to the formula found by Rubeck et.al

$$H_{x,A} = \frac{\sigma_m}{4\pi\mu_0} \left(\frac{-b}{2D_{ab}} \ln\left(\frac{D_{abc} + D_{ab}}{D_{abc} - D_{ab}}\right) + \frac{1}{2} \ln\left(\frac{D_{abc} + b}{D_{abc} - b}\right)\right) . \qquad \text{(A.36)}$$

Integration of equation for the y-component can be performed similarly to the derivation for the x-component. Starting by pushing the factors to the other side of equation (A.2)

$$\frac{4\pi\,\mu_0\,H_{y,A}(\mathbf{r})}{\sigma_m} = \int_{x=0}^{x=a} \int_{y=0}^{y=bx/a} \frac{-y}{(x^2 + y^2 + c^2)^{3/2}} \mathrm{d}y\mathrm{d}x . \qquad \text{(A.37)}$$

By exchanging the order of integration, and adjusting the integration limits, this can be written as

$$\frac{4\pi\,\mu_0\,H_{y,A}(\mathbf{r})}{\sigma_m} = \int_{y=0}^{y=b} \int_{x=0}^{x=ay/b} \frac{-y}{(x^2 + y^2 + c^2)^{3/2}} \mathrm{d}x\mathrm{d}y , \qquad \text{(A.38)}$$

which can be written as two integrals

$$I_{1,y}(y) = \int_{x=0}^{x=ay/b} \frac{-1}{(x^2+y^2+c^2)^{3/2}} \mathrm{d}x \ , \tag{A.39}$$

$$I_{2,y} = \int_{y=0}^{y=b} -y\, I_{1,y}(y) \mathrm{d}y \ . \tag{A.40}$$

Integral $I_{1,y}(y)$ can be solved by substituting

$$x = \sqrt{y^2+c^2}\tan(u)$$

and evaluates to

$$I_{1,y}(y) = \frac{ay}{(y^2+c^2)\sqrt{(a^2+b^2)y^2+b^2c^2}} \ . \tag{A.41}$$

The steps resemble the derivation from (A.5) to (A.17). Therefore, $I_{2,y}$ looks like

$$I_{2,y} = \int_{y=0}^{y=b} -y \frac{ay}{(y^2+c^2)\sqrt{(a^2+b^2)y^2+b^2c^2}} \mathrm{d}y \ , \tag{A.42}$$

and can be solved the same way as integral (A.18). Following the steps from above ((A.18) to (A.35)) the equation for $H_{y,A}$ is found

$$H_{y,\mathrm{A}} = \frac{\sigma_m}{4\pi\mu_0}\left\{ \frac{a}{2D_{ab}} \ln\left(\frac{D_{abc}+D_{ab}}{D_{abc}-D_{ab}}\right) - \frac{1}{2}\ln\left(\frac{D_{ac}+a}{D_{ac}-a}\right)\right\} \tag{A.43}$$

Integration of the z-component again starts with pushing the prefactors in equation (A.3) to the right-hand side:

$$\frac{4\pi\,\mu_0\,H_{z,A}(\mathbf{r})}{c\,\sigma_m} = \int_{x=0}^{x=a}\int_{y=0}^{y=bx/a}\frac{1}{(x^2+y^2+c^2)^{3/2}}\mathrm{d}y\mathrm{d}x\;, \quad \text{(A.44)}$$

which can be written as to integrals:

$$I_{1,z}(x) = \int_{y=0}^{\frac{bx}{a}}\frac{1}{(x^2+y^2+c^2)^{3/2}}\mathrm{d}y\;, \quad \text{(A.45)}$$

$$I_{2,z} = \int_{x=0}^{x=a} I_{1,z}(x)\mathrm{d}x\;. \quad \text{(A.46)}$$

The evaluation from $I_{1,z}(x)$ is equivalent to the derivation performed from (A.5) to (A.17) and results in

$$I_{1,z}(x) = \frac{b}{a}\frac{1}{(x^2+c^2)}\frac{x}{\sqrt{\frac{a^2+b^2}{a^2}\,x^2+c^2}}\;. \quad \text{(A.47)}$$

This expression can now be integrated over $\mathrm{d}x$ to calculate $I_{2,z}$

$$I_{2,z} = \int_{x=0}^{x=a}\frac{b}{a}\frac{1}{(x^2+c^2)}\frac{x}{\sqrt{\frac{a^2+b^2}{a^2}\,x^2+c^2}}\mathrm{d}x\;, \quad \text{(A.48)}$$

by substituting

$$\tan(u) = \frac{a}{b}\sqrt{\frac{a^2+b^2}{a^2}\,\frac{x^2}{c^2}+1}\;, \quad \text{(A.49)}$$

and hence

$$\frac{x}{\sqrt{\frac{a^2+b^2}{a^2}\,x^2+c^2}}\mathrm{d}x = \left(1+\tan^2(u)\right)\frac{abc}{a^2+b^2}\mathrm{d}u\;, \quad \text{(A.50)}$$

$$x^2 + c^2 = \left(1 + \tan^2(u)\right) \frac{b^2 c^2}{a^2 + b^2} \,. \tag{A.51}$$

With these substitutions (A.48) reads

$$I_{2,z} = \frac{b}{a} \frac{a^2 + b^2}{b^2 c^2} \frac{abc}{a^2 + b^2} \int_{u(x=0)}^{u(x=a)} \frac{1}{\left(1 + \tan^2(u)\right)} \left(1 + \tan^2(u)\right) \mathrm{d}u \,. \tag{A.52}$$

After cancelling fractions this simplifies to

$$I_{2,z} = \frac{c}{c^2} \int_{u(x=0)}^{u(x=a)} 1 \mathrm{d}u \,. \tag{A.53}$$

The integration limits follow from (A.49) and read

$$u(x = 0) = \arctan\left(\frac{a}{b}\right) \tag{A.54}$$

$$u(x = a) = \arctan\left(\frac{a}{bc} D_{abc}\right) \tag{A.55}$$

Evaluation of (A.53) with these limits and taking the prefactor into account yields the equation found in Rubeck's paper for $\mathbf{H}_z$

$$H_{z,A} = \frac{\sigma_m}{4\pi\mu_0} \left\{ \arctan\left(\frac{a\, D_{abc}}{bc}\right) - \frac{|c|}{c} \arctan\left(\frac{a}{b}\right) \right\} \tag{A.56}$$

The equations for type B triangles (2.46) to (2.48) can be derived analogously. The derivation only differs in the use of the geometry parameters a, b and c, according to figure 2.2.

Appendix B.

Code Listing

```
import numpy as np
# import numpy functions into the local namespace
#   for readability and for small performance improvement (no need to call numpy namespace)
from numpy import absolute,arctan2,argsort,array,cross,empty,empty_like,isclose,log
from numpy import newaxis,ones,pi,sign,size,sqrt,swapaxes,tile,zeros_like
from numpy.linalg import norm


def get_n_plane(T1,T2,T3,inner_point):
    '''Compute the normalized outward pointing normal vector to the plane defined by triangle T1, T2 and T3.

    Outward in this context is indirectly defined by the input "inner_point", which gives a point inside
    the magnet to which T1, T2 and T3 define a surface facet. In the workflow this function is used in, it is
    easier to give an inner point rather than an outer point.
    To calculate the normalized outward pointing normal vector, one first calculates one of the two plane normals,
    by taking the cross product of two plane vectors - in this case T1-T2 and T1-T3.
    This vector is the unoriented plane normal: un_plane
```

```
18      Then a vector is defined, that - by construction -
        points out of the magnet: vector_out.
19      Taking the sign of the inner product of the '
        unoriented' plane normal and the outwardpointing
        vector
20      indicates if they both point outward (+) or if
        un_plane points inward (-).
21      Given this information one can multiply the
        unoriented plane normal with this sign, and
        therefore
22      calcuate the outward pointing plane normal, which
        then only has to be normalized.
23
24      Inputs:         /kxl = number of facets x number of
        observation points/
25      T1:             (kxl,3) ndarray float64       First
        triangle point defining the plane
26      T2:             (kxl,3) ndarray float64
        Second triangle point defining the plane
27      T3:             (kxl,3) ndarray float64       Third
        triangle point defining the plane
28      inner_point:    (kxl,3) ndarray float64       Point
        inside the magnet, to which
29                                                      T1,
        T2 and T3 define a surface facet
30      Outputs:
31      n_plane         (kxl,3) ndarray float64
        normalized outward pointing, plane normal vector
32      '''
33
34      # get the unoriented plane normal | see eq.(3.30)
35      un_plane = cross(T1-T2, T1-T3)
36
37      # get an outward pointing vector | see eq.(3.31)
38      vector_out = T1-inner_point
39
40      # get the sign of the projection of vector_out onto
        un_plane
41      sig = sign(np.sum(un_plane*vector_out, axis=1,
        keepdims=True))
42
43      # correctly oriented (sign) and normalized plane
        normal | see eq.(3.32)
44      n_plane = sig*un_plane/norm(un_plane, axis=1,
```

```
    keepdims=True)
45
46      return n_plane
47
48
49  def orthogonal_projection_plane_3D(T1,P,n_plane):
50      '''Get the orthogonal projection of a point onto a
    plane in 3D.
51
52      This is done by subtracting the component
    orthogonal to the plane from the vector P-X, where
53      X is an arbitrary point on the plane (in this case
    T1).
54      The plane is defined by T1 and the normalized plane
        normal vector n_plane.
55
56      Inputs:     /kxl = number of facets x number of
    observation points/
57      T1:         (kxl,3) ndarray float64;      First
    triangle point defining the plane
58      P:          (kxl,3) ndarray float64;      Point to
    be orthogonally projected onto the plane defined
59                                                  by T1
    and n_plane.
60      n_plane:    (kxl,3) ndarray float64;
    normalized plane normal vector
61
62      Output:
63      RAP_plane:  (kxl,3) ndarray float64;
    Projection of P onto the plane
        '''
65      # calculate the component of P-T1 that is
    orthogonal to the plane
66          # -> the projection of P-T1 onto n_plane
67      h = np.sum((P-T1)*n_plane, axis=1,keepdims=True)
68
69      # calculate the orthogonal projection of P onto the
        plane | see eq.(3.1)
70      RAP_plane = P-h*n_plane
71
72      return(RAP_plane)
73
74
75  def getOrAngle3D(V1,V2,n_plane):
```

```
76      """Calculate the oriented angle FROM V1 TO V2 for 3
        D vectors.
77
78      See subsection section 3.2.2
79      Inputs:         /kxl = number of facets x number of
        observation points/
80      V1:             (kxl,3) ndarray float 64;     first
        input vector
81      V1:             (kxl,3) ndarray float 64;
        second input vector
82      n_plane:        (kxl,3) ndarray float 64;
        normalized outward-pointing plane normal vector
83
        defining the "up direction"
84
85      Outputs:
86      oriented_angle: (kxl,) numpy float64;
        oriented angle from V1 to V2 in radians
87                                                      in
        range [-pi;pi]
88      """
89      #precalculate the arguments of arctan2 for
        readability
90      #numerator
91      arg1 = np.sum(cross(V1, V2)*n_plane,axis=1)
92      #denominator
93      arg2 = np.sum(V1*V2,axis=1)
94      #calculate the oriented angle in the right quadrant
        with arctan2 | see eq.(3.12)
95      oriented_angle = arctan2(arg1, arg2)
96
97      return oriented_angle
98
99
100 def triangle_sort(T1,T2,T3,RAP_P,n_plane,num):
101     '''Function to consistently sort three input points
        representing the corners of a triangle.
102
103     This function assures that T1,T2 and T3 are
        consistently sorted like follows:
104     In the local coordinate system with n_plane
        pointing in +z direction,
105     the points are sorted so that when looking from
        RAP_P towards the triangle
```

```
106     A is the far left point, B is the far right point
    and C is the middle point.
107
108     Inputs:     /kxl = number of facets x number of
    observation points/
109     T1:         (kxl,3) ndarray float64;      First
    triangle point
110     T2:         (kxl,3) ndarray float64;      Second
    triangle point
111     T3:         (kxl,3) ndarray float64;      Third
    triangle point
112     RAP_P:      (kxl,3) ndarray float64;      Right
    angle point of the global observation point
113                                                 onto
    the triangle plane
114     n_plane:    (kxl,3) ndarray float 64;
    normalized plane normal vector
115     num:        kxl     int
116
117     Outputs:
118     A:          (kxl,3) ndarray float64;      Leftest
    triangle point
119     B:          (kxl,3) ndarray float64;      Rightest
    triangle point
120     C:          (kxl,3) ndarray float64;      Middle
    triangle point
121     '''
122
123     #put the points into an array to sort them
    afterwards (see numpy argsort)
124     points = array([T1,T2,T3])
125
126     #calculate the angles between RAP_P-T1 and RAP_P-T1
    , RAP_P-T2 and RAP_P-T3 respectively
127     angles = zeros_like(T1)
128     # angles[:,0] always 0 by construction
129     angles[:,1] = getOrAngle3D(RAP_P-T1, RAP_P-T2,
    n_plane)
130     angles[:,2] = getOrAngle3D(RAP_P-T1, RAP_P-T3,
    n_plane)
131     # get the indices that sort the angles-array
132     angle_indices = argsort(angles, axis=1)
133
134     # ran = range(num) for correctly indexing the axis
```

```
    =1 dimension of points
135     ran = range(num)
136     # asign the points in the described way
137     A = points[angle_indices[:,2],ran,:]
138     B = points[angle_indices[:,0],ran,:]
139     C = points[angle_indices[:,1],ran,:]
140 
141     return A,B,C
142 
143 
144 def orthogonal_projection_tria_edges_3D(A,B,C,P):
145     '''Returns the orthogonal projection of a point P
    onto all three edges of the triangle A-B-C.
146 
147     This is done by first calculating the normalized
    triangle edge vectors and
148     the vectors pointing from the corners to the point
    P. Then the inner product
149     of the corner-to-P vectors with the normalized edge
     vectors gives the distance the
150     projection point is from the corner. Going this
    distance from the corner point in the direction
151     of the normalized edge vector yields the desired
    projection-/right-angle-point.
152 
153     Inputs:     /kxl = number of facets x number of
    observation points/
154     A:          (kxl,3) ndarray float64;      Leftest
    triangle point
155     B:          (kxl,3) ndarray float64;      Rightest
    triangle point
156     C:          (kxl,3) ndarray float64;      Middle
    triangle point
157     P:          (kxl,3) ndarray float64;
    Observation point
158 
159     Outputs:
160     V5          (3*kxl,3) ndarray float64;    Array
    containing the right angle points:
161         RAP_AB:     (kxl,3) ndarray float64;
    orthogonal projection of P onto A-B
162         RAP_AC:     (kxl,3) ndarray float64;
    orthogonal projection of P onto A-C
163         RAP_BC:     (kxl,3) ndarray float64;
```

```
    orthogonal projection of P onto B-C
    '''

    # normalization of triangle edge vectors
    #   array of edge vectors
    V1 = array([(A-B), (A-C), (B-C)])
    #   array of normalized edge vectors
    V2 = V1/norm(V1, axis=2, keepdims=True)

    # projection on edge lines
    #   array of corner points
    V3 = array([A, A, B])
    #   array of vectors pointing from corners to P
    V4 = array([(P-A), (P-A), (P-B)])
    #   calculating the projection point | see eq
    .(3.14)
    V5 = V3 + np.sum(V4*V2, axis=2, keepdims=True)*V2

    return V5


def InOut_BigSmall_Fun(A,B,C,RAP_P,num_facets):
    '''Determine if RAP_P lies inside our outside the
    triangle A,B,C and if RAP_P lies in a the big or
    small sector of this triangle.

    RAP_P = RAP_P
    ------- InOut -------
    define a local coordinate system by putting C into
    the origin and defining
    A-C and B-C as the local basis.
    Then RAP_P can be written as
      # RAP_P = C + p*(A-C) + q*(B-C)
    subtracting C and renaming RAP_P-C -> v0; A-C -> v1
    ; B-C -> v2 yields:
    v0 = p*v1+q*v2
    taking this equation and performing the dot product
    (.) once with v1 and once with v2 results in the
    following
    system of equations :
    # v0.v1 = p*v1.v1+q*v2.v1
    # v0.v2 = p*v1.v2+q*v2.v2

    solving this yields:
```

```
200     # p = [(v0.v1)*(v2.v2)-(v0.v2)(v2.v1)] / [(v1.v1)*(
        v2.v2)-(v1.v2)(v2.v1)]
201     # q = [(v0.v2)*(v1.v1)-(v0.v1)(v1.v2)] / [(v1.v1)*(
        v2.v2)-(v1.v2)(v2.v1)]
202     the conditions for RAP_P lying in the triangle is
        that all of the following are true
203       # p > 0
204       # q > 0           # q always has same sign as p
205       # p + q < 1
206
207     ------- BigSmall -------
208     Big sector: RAP_P lies in the same sector as the
        triangle
209     Small sector: RAP_P lies in the opposite sector the
        triangle
210
211     This is equivalent to p > 0.
212
213     Inputs:     /kxl = number of facets x number of
        observation points/
214     A:          (kxl,3) ndarray float64;      Leftest
        triangle point
215     B:          (kxl,3) ndarray float64;      Rightest
        triangle point
216     C:          (kxl,3) ndarray float64;      Middle
        triangle point
217     RAP_P:      (kxl,3) ndarray float64;
        Projection of the observation point onto the
        triangle plane
218
219     Outputs:
220     InOut:      (kxl,1) ndarray float64;      Array
        describing if RAP_P is inside(-1) or outside(1) the
        triangle
221     BigSmall:   (kxl,1) ndarray float64;      Array
        describing if RAP_P is in the big (1) or small (-1)
        sector of the triangle
222     '''
223     # define the vectors
224     #   must NOT be normalized. Else condition is
        meaningless
225     v0 = RAP_P-C
226     v1 = (A-C)
227     v2 = (B-C)
```

```
228
229     # precalculate some dot products
230     v0_v1 = np.sum(v0*v1, axis=1)
231     v0_v2 = np.sum(v0*v2, axis=1)
232     v1_v2 = np.sum(v1*v2, axis=1)
233     v1_v1 = np.sum(v1**2, axis=1)
234     v2_v2 = np.sum(v2**2, axis=1)
235
236     # calculate the constants p and q
237     p = (v0_v1*v2_v2-v0_v2*v1_v2)/(v1_v1*v2_v2-v1_v2
    **2)
238     q = (v0_v2*v1_v1-v0_v1*v1_v2)/(v1_v1*v2_v2-v1_v2
    **2)
239
240     # allocate output
241     InOut = ones((num_facets,1))
242     BigSmall = -ones((num_facets,1))
243
244     # bigsmall condition
245     bool_index = p>0
246     BigSmall[bool_index] = 1 #Big
247
248     # inout condition
249     bool_index2 = np.logical_and(bool_index,(p+q)<1)
250     InOut[bool_index2] = -1   # inside
251
252     return InOut, BigSmall
253
254
255 def getLenUnit(V):
256     ''' compute length and unit vector of given input
    vector '''
257     # calculate the side length
258     ab = norm(V,axis=2)
259     # catch ab == 0 case
260     bool_index = np.logical_not(isclose(ab,0))
261     # determine the normalized vector describing the
    triangle edge
262     #   keep the newaxis indexing, because for ab the
    dimension reduction is intended
263     # set xy to 0 if ab==0 and calculate normally
    otherwise
264     xy = np.zeros_like(V)
265     xy[bool_index] = V[bool_index]/ab[bool_index][:,
```

```
    newaxis]
266     return ab, xy
267
268
269 def get_Rubeck_parameters(A,B,C,P,RAP_P,RAP_AB,RAP_AC,
    RAP_BC,n_plane,num):
270     '''Calculate all parameters of the sub-triangles
    necessary to apply the Rubeck2013 formula.
271
272     These parameters are the geometry parameters a,b
    and c, if the sub-triangles are Type A or B,
273     and the sign to correctly sum up the partial fields
    .
274     For the calculation of the field of the (!)acute or
     right(!) A-B-C triangle in the observation point P
275     A-B-C has to be divided into 6 right triangles:
276         #RAP_P-RAP_AC-C
277         #RAP_P-RAP_BC-C
278         #RAP_P-RAP_AC-A
279         #RAP_P-RAP_BC-B
280         #RAP_P-RAP_AB-A
281         #RAP_P-RAP_AB-B
282     For each of these 6 triangles the geometry
    paramters a,b and c as well as
283     their Type (A or B) and their sign(+1 or -1) are
    calculated.
284     In addition to that also the local x-,y- and z-axes
     for each triangle are returned.
285     These are needed to transform the field from the
    local coordinate system into the global one.
286
287     Inputs:         /kxl = number of facets x number of
     observation points/
288     A:              (kxl,3) ndarray float64;
    Leftest triangle point (see sorting)
289     B:              (kxl,3) ndarray float64;
    Rightest triangle point
290     C:              (kxl,3) ndarray float64;
    Middle triangle point
291     P:              (kxl,3) ndarray float64;
    Observation point
292     RAP_P:          (kxl,3) ndarray float64;
    orthogonal projection of P onto triangle plane
293     RAP_AB:         (kxl,3) ndarray float64;
```

```
    orthogonal projection of P onto A-B
294  RAP_AC:         (kxl,3) ndarray float64;
    orthogonal projection of P onto A-C
295  RAP_BC:         (kxl,3) ndarray float64;
    orthogonal projection of P onto B-C
296  n_plane:        (kxl,3) ndarray float64;
    normalized outward-pointing plane normal vector
297
298  Outputs:
299  Rubeck_as:      (kxl,6) ndarray float64;
    Geometry parameters a for each sub-triangle
300  Rubeck_bs:      (kxl,6) ndarray float64;
    Geometry parameters b for each sub-triangle
301  Rubeck_cs:      (kxl,6) ndarray float64;
    Geometry parameters c for each sub-triangle
302  x_local:        (kxl,6,3) ndarray float64;    local
     x-axis for each sub-triangle
303  y_local:        (kxl,6,3) ndarray float64;    local
     y-axis for each sub-triangle
304  z_local:        (kxl,6,3) ndarray float64;    local
     z-axis for each sub-triangle
305  Rubeck_signs:   (kxl,6) ndarray float64;      Sign
    of each sub-triangle, denoting its contribution
306                                                  to
    the total triangle
307  maskA:          (kxl,6) ndarray bool;         Array
     containing the boolean info if a sub triangle is
    type A
308  maskB:          (kxl,6) ndarray bool;         Array
     containing the boolean info if a sub triangle is
    type B
309  '''
310
311  # order of subTriangles per facet:
312      #rap-rap_AC-C
313      #rap-rap_BC-C
314      #rap-rap_AC-A
315      #rap-rap_BC-B
316      #rap-rap_AB-A
317      #rap-rap_AB-B
318
319  # get Rubeck Types (for each subTriangle)
    ----------------- | see 3.2.5
320  # get the type by inspecting if the oriented plane
```

```
normal (n_plane) is parallel (Type A: type = 1) or
321  #   antiparallel (Type B: type = -1) to (RAP_XY-
RAP_P cross X/Y-RAP_XY) (-> order matters!)
322  #
323  # calculate this in fully vectorized fashion by
contructing large arrays on which the same
operations are performed
324  # V1 "first" leg of the triangle for cross product
325  V1 = array([RAP_AC-RAP_P,   RAP_BC-RAP_P,   RAP_AC-
RAP_P,   RAP_BC-RAP_P,   RAP_AB-RAP_P,   RAP_AB-
RAP_P])
326  # V2 "second" leg of the triangle for cross product
327  V2 = array([C-RAP_AC,       C-RAP_BC,       A-
RAP_AC,       B-RAP_BC,       A-RAP_AB,       B-
RAP_AB])
328  Rubeck_types = sign(np.sum(cross(V1, V2, axis=2)*
n_plane, axis=2)).T # transpose for dim consistency
329
330  # get a mask for type A and B to correctly asign
the parameters
331  maskA = Rubeck_types == +1
332  maskB = Rubeck_types == -1
333
334  # get Rubeck signs (for each subTriangle)
---------------- | see 3.2.5
335  Rubeck_signs= ones([num, 6])
336  # test inside or outside  and  Big or small sektor
337  InOut,BigSmall = InOut_BigSmall_Fun(A, B, C, RAP_P,
 num)
338  #rap-rap_AC-C and rap-rap_BC-C are always +1,
compute remaining 4
339  V3 = array([(A-RAP_AC)*(A-C), (B-RAP_BC)*(B-C), -(A
-RAP_AB)*(A-B)*InOut, -(B-RAP_AB)*(B-A)*InOut])
340  Rubeck_signs[:,2:] = BigSmall*sign(np.sum(V3, axis
=2)).T
341
342  # compute a,b,c and local unit vectors
--------------------
343  # triangle edge vectors (for all subTriangles)
344  V4 = swapaxes(array([RAP_AC-RAP_P, RAP_BC-RAP_P,
RAP_AC-RAP_P, RAP_BC-RAP_P, RAP_AB-RAP_P, RAP_AB-
RAP_P]), 0, 1)
345  V5 = swapaxes(array([C-RAP_AC, C-RAP_BC, A-RAP_AC,
B-RAP_BC, A-RAP_AB, B-RAP_AB]), 0, 1)
```

```
346     lengthV4,unitV4 = getLenUnit(V4)
347     lengthV5,unitV5 = getLenUnit(V5)
348 
349     # assign a-b-c parameters
350     Rubeck_as = empty([num,6])
351     Rubeck_bs = empty([num,6])
352     # distance P to RAP_P
353     Rubeck_cs_0 = norm(P-RAP_P, axis=1, keepdims=True)
354     if any(isclose(Rubeck_cs_0,0)): # find a better way
        to catch c == 0 special case
355         print('warning! small c in at least one facet')
356     # tile for arr shape (num,6)
357     Rubeck_cs = tile(Rubeck_cs_0,(1,6))
358 
359     # assign a and b paramters
360     Rubeck_as[maskA] = lengthV4[maskA]
361     Rubeck_as[maskB] = lengthV5[maskB]
362     Rubeck_bs[maskA] = lengthV5[maskA]
363     Rubeck_bs[maskB] = lengthV4[maskB]
364 
365     # local unit vectors
366     x_local = empty([num,6,3])
367     y_local = empty([num,6,3])
368     # zaxis is always normalized P-RAP_P.
369     #   tile for array shape (num,6,3)
370     z_local = tile(((P-RAP_P)/Rubeck_cs_0)[:,newaxis
        ,:],(1,6,1))
371 
372     # assign local x and y axis
373     x_local[maskA] = unitV4[maskA]
374     x_local[maskB] = unitV5[maskB]
375     y_local[maskA] = unitV5[maskA]
376     y_local[maskB] = unitV4[maskB]
377 
378     return Rubeck_as,Rubeck_bs,Rubeck_cs,x_local,
        y_local,z_local,Rubeck_signs,maskA,maskB
379 
380 
381 def Rubeck_formula_A(a,b,c):
382     '''Formula to calculate the field of a
        homogeneously charged right triangle of type A,
        after Rubeck2013.
383 
384     Function accepts array input
```

```
385     Inputs: /kxl = number of facets x number of
        observation points/
386     a:      (kxl,) ndarray float64;       side length
        of the right triangle
387     b:      (kxl,) ndarray float64;       side length
        of the right triangle
388     c:      (kxl,) ndarray float64;       distance of
        the observation point to the triangle corner
389
390     Outputs:
391     field:  (kxl,3) ndarray float64;     field vector
        in the observation point
392     '''
393
394     Dab = sqrt(a**2+b**2)
395     Dac = sqrt(a**2+c**2)
396     Dabc = sqrt(a**2+b**2+c**2)
397     Hx = (-b/2./Dab)*log((Dabc+Dab)/(Dabc-Dab)) + 1/2.*
        log((Dabc+b)/(Dabc-b))
398     Hy = (a/2./Dab)*log((Dabc+Dab)/(Dabc-Dab)) - 1/2.*
        log((Dac+a)/(Dac-a))
399     Hz = arctan2((a*Dabc),(b*c)) - absolute(c)/c*
        arctan2(a,b)
400     field = array([Hx,Hy,Hz]).T
401     return field
402
403
404 def Rubeck_formula_B(a,b,c):
405     '''Formula to calculate the field of a
        homogeneously charged right triangle of type B,
        after Rubeck2013.
406
407     Function accepts array input
408     Inputs: /kxl = number of facets x number of
        observation points/
409     a:      (kxl,) ndarray float64;       side length
        of the right triangle
410     b:      (kxl,) ndarray float64;       side length
        of the right triangle
411     c:      (kxl,) ndarray float64;       distance of
        the observation point to the triangle corner
412
413     Outputs:
414     field:  (kxl,3) ndarray float64;     field vector
```

```
    in the observation point
415     '''
416     Dab = sqrt(a**2+b**2)
417     Dbc = sqrt(b**2+c**2)
418     Dabc = sqrt(a**2+b**2+c**2)
419     Hx = (b/2./Dab)*log((Dabc+Dab)/(Dabc-Dab)) - 1/2.*
    log((Dbc+b)/(Dbc-b))
420     Hy = (-a/2./Dab)*log((Dabc+Dab)/(Dabc-Dab)) + 1/2.*
    log((Dabc+a)/(Dabc-a))
421     Hz = -arctan2((b*Dabc),(-a*c)) - absolute(c)/c*
    arctan2(b,a)
422     field = array([Hx,Hy,Hz]).T
423     return field
424
425
426 def Bfield_FacetV(MAG, DIM, POSO):
427     '''Calculate the field of homogeneously "
    magnetically charged" triangles defined by point
    arrays
428     T1-T2-T3 in the observation points P.
429
430     Therefore do the following steps (vectorized):
431         - calculate the orthogonal projection of P onto
     the triangle plane.
432         - sort the triangle corner points for
    consistency.
433         - calculate the orthogonal projection of P onto
     the triangle edges.
434             - now the input triangles is devided into 6
     smaller right triangles.
435         - get the geometry parameters, the local axes,
    the Rubeck Types and the sign of the sub-triangles.
436         - calculate the field in the local coordinate
    system for each sub-triangle.
437         - transform the local fields into the global
    coordinate frame.
438         - sum the sub-triangles back up according to
    their sign.
439     Inputs:             /k = number of facets; l =
    number of observation points/
440     MAG:                (k,3) ndarray float64;
    Magnetization of the body the facet belongs to
441                                                     in
    the global coordinate frame
```

```
442
    sigma is derived from MAG
443     DIM:                (k,4,3) ndarray float64;  Array
        describing the geometry of the system
444                                                       4:
    three triangle points and 1 point describing "inside
    "
445     POSO:               (1,3) ndarray float64;
    Observation Point
446
447     Outputs:
448     field_return:       (1,3) ndarray float64; total
    field of all facets in points P in global basis
449     '''
450
451     ####################################
452     # enables the vectorized calculation | See Section
    3.3
453     # number of input facets and observation points
454     num_facets_in = size(DIM, axis=0)
455     num_POSOs = size(POSO, axis=0)
456
457     # tile and repeat input according to figure 3.7
458     DIM = np.tile(DIM,[num_POSOs,1,1])
459     MAG = np.tile(MAG,[num_POSOs,1])
460     POSO = np.repeat(POSO,num_facets_in,axis=0)
461
462     # num = kxl
463     num = size(DIM, axis=0)
464     ####################################
465
466     # extract the input points
467     T1 = DIM[:,0]
468     T2 = DIM[:,1]
469     T3 = DIM[:,2]
470     inner_point = DIM[:,3]
471
472     # compute outward pointing plane normal vector |
    See Section 3.3 DIM
473     n_plane = get_n_plane(T1, T2, T3, inner_point)
474
475     # get right angle point (projection point) of P
    onto triangle plane | See 3.2.1
476     RAP_P = orthogonal_projection_plane_3D(T1, POSO,
```

```
    n_plane)
477
478     # sort T1,T2,T3 to enable Rubeck formula
    application | See 3.2.2
479     A,B,C = triangle_sort(T1, T2, T3, RAP_P, n_plane,
    num)
480
481     # get the orthogonal projections of P onto the
    triangle edges | See 3.2.3
482     RAP_AB,RAP_AC,RAP_BC =
    orthogonal_projection_tria_edges_3D(A, B, C, POSO)
483
484     # get the geometry parameters, the local axes, the
    Rubeck types and the signs | See 3.2.5
485     a,b,c,x_local,y_local,z_local,Rubeck_signs,maskA,
    maskB = \
486           get_Rubeck_parameters(A, B, C, POSO, RAP_P,
    RAP_AB, RAP_AC, RAP_BC, n_plane, num)
487
488     # initialize arrays to store the local fields of
    the sub-triangles
489     field_local_part = empty((num,6,3))
490
491     # calculate the field for both types for all sub-
    triangles
492     field_local_part[maskA] = Rubeck_formula_A(a[maskA
    ], b[maskA], c[maskA])
493     field_local_part[maskB] = Rubeck_formula_B(a[maskB
    ], b[maskB], c[maskB])
494
495     # transform field of each sub-triangles from
    respective local to global | See 3.2.5
496     field_global_part = empty_like(field_local_part)
497     field_global_part[:,:,0] = field_local_part[:,:,0]*
    x_local[:,:,0]+ \
498                                field_local_part[:,:,1]*
    y_local[:,:,0]+ \
499                                field_local_part[:,:,2]*
    z_local[:,:,0]
500
501     field_global_part[:,:,1] = field_local_part[:,:,0]*
    x_local[:,:,1]+ \
502                                field_local_part[:,:,1]*
    y_local[:,:,1]+ \
```

```
503                                 field_local_part[:,:,2]*
    z_local[:,:,1]
504
505     field_global_part[:,:,2] = field_local_part[:,:,0]*
    x_local[:,:,2]+ \
506                                 field_local_part[:,:,1]*
    y_local[:,:,2]+ \
507                                 field_local_part[:,:,2]*
    z_local[:,:,2]
508
509     # sum up the contributions from the sub-triangles (
    careful signs)
510     field_return = np.sum(field_global_part*
    Rubeck_signs[:,:,newaxis], axis=1)
511
512     # calculate prefactor for outputing the H-field
513     sigma = np.sum(n_plane*MAG, axis=1, keepdims=True)
514     prefactor = sigma/4/pi
515
516     # field of each facet in one point with correct
    unit
517     field_facets = prefactor * field_return
518
519     # sum up the field of all facets IN ONE POINT and
    store in an array
520         # one row for each oberservation position (POSO
    ) and one column for each dimension
521     field_total_array = np.empty([num_POSOs,3])
522
523     for i in range(num_POSOs):
524         field_total_array[i,:] = np.sum(field_facets[i*
    num_facets_in:(i+1)*num_facets_in,:],axis=0)
525     return field_total_array
```

Bibliography

[1] Udo Ausserlechner, ed. *Magnetic sensors and devices: Technologies and applications*. Devices, circuits, and systems. Boca Raton: CRC Press, 2018. ISBN: 9781498710978 (cit. on p. 1).

[2] Michael Ortner, Marcelo Ribeiro, and Dietmar Spitzer. "Absolute Long-Range Linear Position System With a Single 3-D Magnetic Field Sensor." In: *IEEE Transactions on Magnetics* 55.1 (2019), pp. 1–4. ISSN: 0018-9464. DOI: 10.1109/TMAG.2018.2870597 (cit. on p. 1).

[3] Daniel Cichon et al. "A Hall-Sensor-Based Localization Method With Six Degrees of Freedom Using Unscented Kalman Filter." In: *IEEE Sensors Journal* 19.7 (2019), pp. 2509–2516. ISSN: 1530-437X. DOI: 10.1109/JSEN.2018.2887299 (cit. on p. 1).

[4] Katie M. Popek, Thomas Schmid, and Jake J. Abbott. "Six-Degree-of-Freedom Localization of an Untethered Magnetic Capsule Using a Single Rotating Magnetic Dipole." In: *IEEE Robotics and Automation Letters* 2.1 (2017), pp. 305–312. DOI: 10.1109/LRA.2016.2608421 (cit. on p. 1).

[5] Infineon AG. *XENSIV—Sensing the world*. 2020. URL: https://www.infineon.com/dgdl/Infineon-SensorSelectionGuide-ProductSelectionGuide-v01_00-EN.pdf?fileId=5546d462636%20cc8fb0164229c09f51bbe (visited on) (cit. on p. 1).

[6] C.P.O Treutler. "Magnetic sensors for automotive applications." In: *Sensors and Actuators A: Physical* 91.1-2 (2001), pp. 2–6. ISSN: 09244247. DOI: 10.1016/S0924-4247(01)00621-5 (cit. on p. 1).

[7] Samuel Huber et al. "A Gradiometric Magnetic Sensor System for Stray-Field-Immune Rotary Position Sensing in Harsh Environment." In: *Proceedings* 2.13 (2018), p. 809. DOI: 10.3390/proceedings2130809 (cit. on p. 2).

[8] Wolfgang Granig, Stephan Hartmann, and Benno Köppl. "Performance and Technology Comparison of GMR Versus Commonly used Angle Sensor Principles for Automotive Applications." In: *SAE Technical Paper Series*. SAE Technical Paper Series. SAE International400 Commonwealth Drive, Warrendale, PA, United States, 2007. DOI: 10.4271/2007-01-0397 (cit. on p. 2).

[9] Jan K. Sykulski, ed. *Computational magnetics*. 1. ed. London: Chapman & Hall, 1995. ISBN: 978-0-412-58570-8 (cit. on p. 2).

[10] Michael Ortner. "Improving magnetic linear position measurement by field shaping." In: *2015 9th International Conference on Sensing Technology (ICST)*. IEEE, 8.12.2015 - 10.12.2015, pp. 359–364. ISBN: 978-1-4799-6314-0. DOI: 10.1109/ICSensT.2015.7438423 (cit. on p. 3).

[11] Perla Malagò et al. "Magnetic Position System Design Method Applied to Three-Axis Joystick Motion Tracking." In: *Sensors (Basel, Switzerland)* 20.23 (2020). DOI: 10.3390/s20236873 (cit. on p. 3).

[12] Fabrice Bernier et al. "Additive manufacturing of soft and hard magnetic materials used in electrical machines." In: *Metal Powder Report* 75.6 (2020), pp. 334–343. ISSN: 00260657. DOI: 10.1016/j.mprp.2019.12.002 (cit. on p. 3).

[13] E. A. Périgo et al. "Additive manufacturing of magnetic materials." In: *Additive Manufacturing* 30 (2019), p. 100870. ISSN: 22148604. DOI: 10.1016/j.addma.2019.100870 (cit. on p. 3).

[14] Michael Ortner and Lucas Gabriel Coliado Bandeira. "Magpylib: A free Python package for magnetic field computation." In: *SoftwareX* 11 (2020), p. 100466. ISSN: 2352-7110. DOI: 10.1016/j.softx.2020.100466. URL: http://www.

sciencedirect.com/science/article/pii/S2352711020300170 (cit. on pp. 4, 57, 75, 81).

[15] Michael Ortner. *Magpylib Docs*. 2020. URL: https://magpylib.readthedocs.io/en/latest/ (visited on) (cit. on p. 4).

[16] Travis E. Oliphant. *Guide to NumPy*. 2nd edition. Austin, Texas: Continuum Press, September 2015. ISBN: 9781517300074 (cit. on pp. 4, 15).

[17] John L. Hennessy, David A. Patterson, and Krste Asanović. *Computer architecture: A quantitative approach*. 5th ed. Waltham, MA: Morgan Kaufmann/Elsevier, 2012. ISBN: 9780123838728 (cit. on pp. 6, 7, 10).

[18] Bruce Jacob, Spencer Ng, and David T. Wang. *Memory systems: Cache, DRAM, disk*. Digital printing. Burlington, MA: Morgan Kaufmann Publishers, 2010. ISBN: 0123797519 (cit. on pp. 6–9).

[19] Ashok N. Kamthane and Raj Kamal. *Computer programming and IT*. Always Learning. New Delhi: Dorling Kindersley, 2012. ISBN: 9788131774694 (cit. on pp. 6, 7).

[20] Arthur W. Burks, Herman H. Goldstine, and John Neumann. "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument." In: *The Origins of Digital Computers*. Ed. by Brian Randell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 399–413. ISBN: 978-3-642-61814-7. DOI: 10.1007/978-3-642-61812-3{\textunderscore}32 (cit. on p. 6).

[21] Wm. A. Wulf and Sally A. McKee. "Hitting the memory wall." In: *ACM SIGARCH Computer Architecture News* 23.1 (1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588 (cit. on p. 7).

[22] Alexey Lyashko. *Mastering Assembly Programming*. Packt Publishing, Limited. ISBN: 1-78728-748-3 (cit. on p. 10).

[23] Ciji Isen, Lizy K. John, and Eugene John. "A Tale of Two Processors: Revisiting the RISC-CISC Debate." In: *Computer Performance Evaluation and Benchmarking*. Ed. by David Kaeli and Kai Sachs. Vol. 5419. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 57–76. ISBN: 978-3-540-93798-2. DOI: 10.1007/978-3-540-93799-9{\textunderscore}4 (cit. on p. 10).

[24] Nabil Nassif, Jocelyne Erhel, and Bernard Philippe. *Introduction to computational linear algebra*. Boca Raton, Florida: CRC Press, Taylor & Francis Group, 2015. ISBN: 9781482258691 (cit. on p. 11).

[25] Siddharta Govindaraj. *Test-driven Python development: Develop high-quality and maintainable Python applications using the principles of test-driven development*. Community experience distilled. Birmingham, UK: Packt Publishing, 2015. ISBN: 1783987936 (cit. on p. 11).

[26] Micha Gorelick and Ian Ozsvald. *High performance Python*. First edition. Sebastopol, CA: O'Reilly, 2014. ISBN: 1449361595 (cit. on p. 13).

[27] Jake Vanderplas. *Why Python is Slow: Looking Under the Hood*. 2014. URL: https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/ (visited on) (cit. on p. 13).

[28] Jake Vanderplas. *Python Data Science Handbook*. Sebastopol, CA: Oreilly & Associates Inc, 2016. ISBN: 9781491912058 (cit. on pp. 14, 16).

[29] *https://www.python.org/*. 2020. URL: https://www.python.org/ (cit. on p. 14).

[30] *PyPy*. 2020. URL: https://www.pypy.org/ (cit. on p. 14).

[31] *Cython: C-Extensions for Python*. 2020. URL: https://cython.org/ (cit. on p. 14).

[32] *F2Py: Fortran to Python interface generator*. 2020. URL: https://numpy.org/doc/stable/f2py/ (cit. on p. 14).

[33] *Numba: A high performance Python compiler*. 2020. URL: https://numba.pydata.org/ (cit. on p. 15).

[34] *LLVM: The LLVM Compiler Infrastructure*. 2020. URL: http://llvm.org/ (cit. on p. 15).

[35] *NumExpr: Fast numerical expression evaluator for NumPy*. 2020. URL: https://pypi.org/project/numexpr/ (cit. on p. 15).

[36] Charles R. Harris et al. "Array programming with NumPy." In: *Nature* 585.7825 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2 (cit. on p. 15).

[37] *NumPy: The fundamental package for scientific computing with Python*. 2020. URL: https://numpy.org/ (cit. on p. 15).

[38] David J. Griffiths. *Introduction to electrodynamics*. Fourth edition. Boston: Pearson, 2013. ISBN: 0-321-85656-2 (cit. on pp. 17, 20, 25).

[39] John David Jackson. *Classical electrodynamics*. 3rd ed. New York: Wiley, 1998. ISBN: 0-471-30932-X (cit. on pp. 17, 26).

[40] Edward M. Purcell. *Electricity and magnetism*. Third edition. Cambridge: Cambridge University Press, 2013. ISBN: 9781107014022 (cit. on pp. 17, 19, 21).

[41] Andrzej Herczyński. "Bound charges and currents." In: *American Journal of Physics* 81.3 (2013), pp. 202–205. ISSN: 0002-9505. DOI: 10.1119/1.4773441 (cit. on p. 18).

[42] Kjell Prytz. *Electrodynamics: The field-free approach : electrostatics, magnetism, induction, relativity and field theory / Kjell Prytz*. Undergraduate lecture notes in physics, 2192-4791. Cham: Springer, 2015. ISBN: 978-3-319-13171-9 (cit. on p. 20).

[43] Daniel C. Mattis. *The Theory of Magnetism I: Statics and Dynamics*. Vol. 17. Springer Series in Solid-State Sciences. Berlin, Heidelberg: Springer Berlin Heidelberg, 1981. ISBN: 978-3-642-83238-3 (cit. on p. 21).

[44] Harald Ibach and Ibach Harald. *The Solid-State Physics: An introduction to principles of materials science, includes 100 problems*. Springer. Springer-Verlag Berlin Heidelberg New York. ISBN: 9783540438700 (cit. on p. 23).

[45] Patrick Fazekas. *Lecture Notes on Electron Correlation and Magnetism*. Vol. 5. WORLD SCIENTIFIC, 1999. ISBN: 978-981-02-2474-5. DOI: 10.1142/2945 (cit. on p. 23).

[46] Giorgio Bertotti and I. D. Mayergoyz. *The science of hysteresis*. Elsevier Series in Electromagnetism. Amsterdam: Elsevier, 2006. ISBN: 978-0-12-480874-4 (cit. on p. 23).

[47] Christophe Rubeck et al. "Analytical Calculation of Magnet Systems: Magnetic Field Created by Charged Triangles and Polyhedra." In: *IEEE Transactions on Magnetics* 49.1 (2013), pp. 144–147. ISSN: 0018-9464. DOI: 10.1109/TMAG.2012.2219511 (cit. on pp. 24, 27, 30, 75, 81, 89).

[48] Adrian Leonhard. *Signed angle between two 3D vectors with same origin within the same plane*. Stack Overflow, 10/15/2020. URL: https://stackoverflow.com/questions/5188561/signed-angle-between-two-3d-vectors-with-same-origin-within-the-same-plane (visited on) (cit. on p. 41).

[49] COMSOL. *The Finite Element Method (FEM)*. 2016. URL: https://www.comsol.de/multiphysics/finite-element-method (visited on) (cit. on pp. 67, 68).

[50] Bettina Schieche. *The Strength of the Weak Form*. 2014. URL: https://www.comsol.com/blogs/strength-weak-form/ (visited on) (cit. on p. 68).

[51] Klaus-Jürgen Bathe. *Finite element procedures for solids and structures: Nonlinear analysis*. MIT video course. [Cambridge, Mass.]: MIT, Center for Advanced Engineering Study, 1986. ISBN: 0133014584 (cit. on p. 69).

[52] Gouri Dhatt, Emmanuel Lefrancois, and Gilbert Touzot. *Finite Element Method*. ISTE. London: Wiley, 2013. ISBN: 9781848213685 (cit. on p. 69).