Numerical Simulation of Weakly Coupled Multiphysical Problems in Electrical Engineering

Investigations into Data Structures, Programming Paradigms and Creation of a Prototypic Universal Simulation Framework

Dipl.-Ing. Michael Jaindl

Numerical Simulation of Weakly Coupled Multiphysical Problems in Electrical Engineering

Investigations into Data Structures, Programming Paradigms and Creation of a Prototypic Universal Simulation Framework

Doctoral Thesis

at

Graz University of Technology

submitted by

Dipl.-Ing. Michael Jaindl

Institute for Fundamentals and Theory in Electrical Engineering (IGTE),
Graz University of Technology
Graz, Austria

25th February 2011

Supervisor:	ao.Univ.-Prof. Dr. Christian Magele
External Assessor:	Prof. Dr. Piergiorgio Alotto

Graz University of Technology

Numerische Simulation von schwach gekoppelten multiphysikalischen Problemen in der Elektrotechnik

Untersuchungen an Datenstrukturen, Softwareentwicklungsparadigmen und Erstellung eines prototypischen, universellen Simulationswerkzeugs

Dissertation

an der

Technischen Universität Graz

vorgelegt von

Dipl.-Ing. Michael Jaindl

Institut für Grundlagen und Theorie der Elektrotechnik (IGTE),
Technische Universität Graz
Graz, Österreich

25. Februar 2011

Diese Arbeit ist in englischer Sprache verfasst.

Betreut durch:	ao.Univ.-Prof. Dr. Christian Magele
Externe Begutachtung:	Prof. Dr. Piergiorgio Alotto

TU Graz
Graz University of Technology

Abstract

Every engineering discipline faces the fact of ever-shortening time-to-market windows and development cycles. In order to counteract these, virtual prototyping, simulation and problem optimization are employed in a rapidly increasing number of cases. Yet, the key to efficient problem formulation by professionals still lies in the use of sophisticated simulation software capable of processing numerous diverse design and optimization tasks in a versatile way.

More often than not, different tools for different workflows need to be coordinated and interdepend on each other's data in the design process chain. When toolchains need to be run multiple times, as it is typically the case in numerical optimization, the lineup overhead tends to be tedious to both man and machine.

This thesis describes different aspects concerning the design of a software and data framework which tackles the problem of lining up software tools that may be incoherent in terms of data exchange and control mode. The resulting system covers all parts of multiphysical simulation problems that may arise in electrical engineering and its adjoining disciplines as an application of the finite element method.

Kurzfassung

So gut wie jede Ingenieursdisziplin sieht sich heutzutage stets kürzer werdenden Entwicklungszyklen und Produktvorlaufzeiten gegenüber. Um entsprechend mithalten zu können, werden zunehmend Virtuelle Prototypenentwicklung, Simulation und Problemoptimierung angewandt. Dennoch liegt der Schlüssel zu effizienter Problemformulierung durch Professionisten nach wie vor in der Verwendung ausgeklügelter Simulationssoftware, die es erst ermöglicht, eine Vielzahl unterschiedlicher Design- und Optimierungsaufgaben bearbeiten zu können.

Häufig müssen in so einem Entwicklungsprozess höchst unterschiedliche Werkzeuge für verschiedene Abläufe koordiniert werden, die zusätzlich auch noch wechselweise von produzierten Daten abhängig sind. Werden solche Prozessketten mehrfach durchlaufen, wie es in der numerischen Optimierung typischerweise der Fall ist, summiert sich der Koordinations- und Kommunikationsaufwand für Mensch wie Maschine.

Diese Arbeit beschreibt verschiedene Aspekte hinsichtlich des Designs einer Software- und Datenumgebung, die das Problemfeld der Aufreihung von - in Bezug auf Datenaustausch und Steuerung sehr inkohärenten - Softwarewerkzeugen aufgreift. Das resultierende System deckt somit alle Teile von multiphysikalischen Simulationsproblemen ab, die in der Elektrotechnik und ihren benachbarten Disziplinen als Anwendung der Methode der Finiten Elemente entstehen mögen.

Statutory Declaration

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 marked all material which has been quoted either literally or by content from the used sources.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen / Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, 14.02.2011

Acknowledgements

There's surely a number of people I would like to extend my thanks to. Outstanding of these are undoubtedly CHRISTIAN MAGELE and PIERGIORGIO ALOTTO. Enjoying all possible freedom along with top-notch personal support and guidance really gave me great times while preparing this thesis. I definitely will miss the scintillatingly witty atmosphere at the IGTE.

Also, RALPH KUTSCHERA and his meticulous, yet ever-conscious work on his master thesis had their share in the successful completion of this project, and he equally deserves gratitude and recognition.

Anyway, all of that would have proven insufficient without my life's centre of gravity.
To JULIA*, my love, and motivation for so many things, I would like to dedicate this work.*

Michael Jaindl
Graz, Austria, February 2011

Credits

- Parts of the prototype described in this thesis were developed in cooperation with Ralph Kutschera as part of his master thesis work [37].
- This thesis was written using Keith Andrews' wonderful skeleton thesis [5].

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction & Concept Formulation

"Poca favilla gran fiamma seconda."

[Dante Alighieri]

Across generations and disciplines, engineering has followed a certain pattern, the engineering cycle. It is commonly perceived as consisting of four steps: specification and design, prototyping, test, evaluation. These follow each other, and as the denotation of being a cycle implies, are typically fed back from the end, evaluation, to its origin, design.

On the one hand, this model expounds the rather philosophical view of technological advance; on the other hand it shows the rocky path that may lead from a problem specification towards a solution to it. The American Engineers' Council for Professional Development (ECPD) has defined 'Engineering' precisely as being [17]:

> ... the creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property.

The process of developing some marketable product of whatever nature now exactly follows the above mentioned engineering cycle. To break new ground may bear unforeseeable obstacles, and therefore massive endeavours have been undertaken to remove the commercial risk from engineering and product development. The ECDP already mentions behaviour prediction, and this is exactly what finally could be obtained some decades ago by making use of increasing computational capacities.

Computer aided design and manufacturing, virtual prototyping, simulation and problem optimization therefore have been employed in a rapidly increasing number of cases. Up to today, all engineering disciplines have lept forward through creation of ingenious tools speeding up the engineering cycle's rotational speed.

Yet, the key to efficient problem formulation by professionals still lies in the use of sophisticated simulation software capable of processing numerous diverse design and optimization tasks in a versatile way. More often than not, different tools for different workflows need to be coordinated and interdepend on each other's data in the design process chain. This is especially true, whenever the solution to a given problem can be – or has to be – partitioned in multiple domains of different physical nature, such as mechanical stress or load and thermodynamics. When such toolchains need to be run multiple times, as it is typically the case in numerical optimization, the lineup overhead tends to be tedious to both man and machine.

This is the inflection point this thesis hooks into.

1.1 What to Expect

This thesis is a recapitulation of results obtained during a research project at the Institute for Fundamentals and Theory in Electrical Engineering of Graz University of Technology [25]. The institute's team works on simulation and optimization strategies in electrical engineering. Around 1980, the institute's research and development activities resulted in the first release of **EleFAnTs**, the **Ele**ctromagnetic **F**ield **An**alysis **T**ools [24]. This software package solves numerical problems in electromagnetics by employing the finite element method (FEM). At that time, FEM was already well introduced for structural analysis in aeronautics, and just made its transition to other engineering disciplines such as thermodynamics, computational fluid dynamics and, surprisingly, also the financial sector.

Though the mentioned software package has undergone development and major revisions during its three decades of existence, its major focus has not been laid on integration into optimization strategies. This thesis, aiming to close this gap, therefore follows the undermentioned structure.

The next chapter, Chapter 2, places the work conducted into its context, as of giving detailed analysis of existing approaches towards computer aided alignment of prototyping tools and optimization toolboxes. Special emphasis is put on multiphysics and how weakly coupled problems may be modeled and solved.

Chapter 3 compiles a catalogue of postulations for an ideal system, delineating what would be a desirable set of assistance tools. It also reports essential characteristics of weak coupling of multiphysical problems and how these should be represented within such toolsets.

Subsequently, Chapter 4 describes the engineering aspects around design and implementation of such a prototypic software system as well as a selection of novel realization details.

Chapter 5 depicts problems actually researched into at IGTE in all necessary detail, and how they can be routed towards a solution making use of the afore detailed system.

As a conclusion, Chapter 6 will reflect on ideas about general trends around this issue, and will motivate possible future work to further improve the existing work and pushing it beyond its current horizons.

Chapter 2

State of the Art

"The most reliable way to anticipate the future is to understand the present."

[John Naisbitt]

When numerical optimization of multiphysical problems is to be addressed, three different issues arise. Each of these accounts for several dimensions of complexity and, therefore, different approaches towards solving have been researched. As a result, numerical simulation, optimization and coupling frameworks are in use in areas as different as natural science, engineering, medicine, and social sciences.

Common to all of these is that the numerical simulation of large-scale, complex systems requires consideration of a number of physical components, such as electromagnetics, solid mechanics, heat transfer, or fluid dynamics and the like [32]. This multidisciplinary nature of research requires simulation code development to be partitioned among different research groups, yielding independently developed software modules. Integration of these modules into a coupled system is usually done by integration frameworks [76].

2.1 Software Integration Frameworks

For describing such frameworks and for establishing clear relations to existing tools, a couple of terms - application, framework, component, and module - need to be defined properly. Table 2.1 gives such a definition.

Application	A collection of components that can be used without adaption to perform tasks for the computer user
Component	A system element offering a predefined service and able to communicate with other components
Framework	A collection of reusable components that act as a template for software applications
MCS	Multi-component simulation application
Module	A software entity that groups a set of (typically cohesive) components

Table 2.1: Definition of technical terms for software frameworks [71]

It has to be put on record that the difference between frameworks and applications is only a small one. [71] defines frameworks as being *"a reusable set of components that can be used as the basis for software applications"*, leaving some kind of freedom to the users for adapting the software at their wish. That may be some parametrization as well as user-produced code that is brought into execution. Additionally, frameworks are based on the reuse of components.

Aiming on the reuse of components and modules, the software designer is confronted with having to find a way to elaborate appropriate data and communication structures. As early as in the 1970s, information hiding was introduced and found to tremendously improve software adaptability [52]. Essentially that means to decouple design decisions which are subject to change. This enables the designer to change the affected parts of code independently. Another key idea would be to create the option to improve a whole system by experimenting with new or other implementations of its sub-parts, be they components or modules, and by seamlessly substituting any inferior for superior solutions [65]

For multiphysics code to be integrated into such a system of different abstraction and interaction layers, a number of challenges in software engineering arise, besides complexities of physical model creation, numerical methods, and computing efficiency [32]:

> First, the data exchanged between modules must be *abstracted* appropriately so that inter-module interfaces can be as simple and clean as possible.
>
> Second, the software architecture must encourage good software practice, such as *encapsulation* and *code reuse*, and provide conveniences to code developers while being as *nonintrusive* as possible.
>
> Third, the software architecture must be *heterogeneous* to provide sufficient flexibility to application scientists and engineers in choosing appropriate discretization schemes, data structures, and programming languages according to their tastes and needs.
>
> Fourth, to support cutting-edge research, the software architecture must *maximize concurrency* in code development of different subgroups and support *rapid prototyping* of various coupling schemes through well-defined service components.

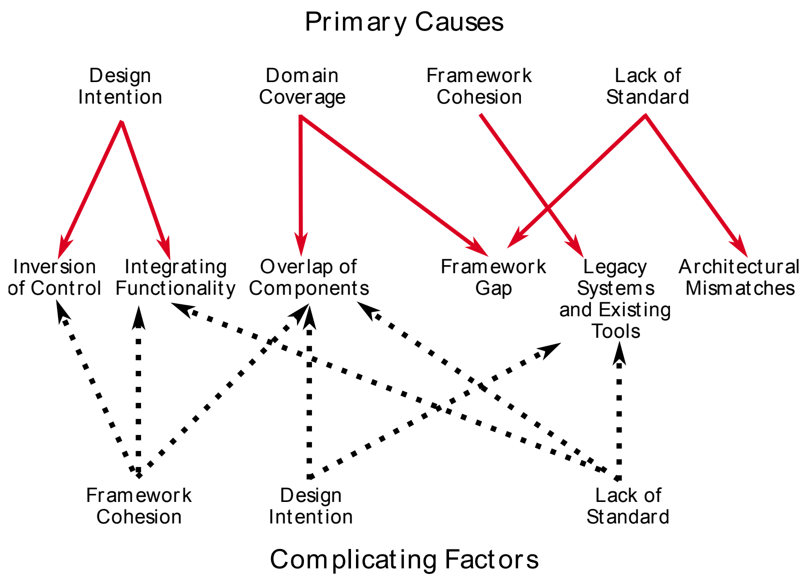

Figure 2.1: Relationships between integration problems and causes. [45]

From a software engineering point of view, [45] identifies a couple of integration problems for software frameworks and their components. The most common can be identified as being the cohesion of frameworks, domain coverage, design intention, source code inaccessibility, and a lack of design standards. Figure 2.1 shows the implications of these problems on actual software generation.

Based on the solutions proposed there, a set of approaches exists for ensuring clean integration of architectural elements of whatever the abstraction level.

Data Management for Inter-Module Communication
Independently from each other, physical components of simulation code may be developed by different sources. Especially when products originating from different development styles, eras or objectives have to be integrated, they differ in significant partitions. Spatial discretization (meshes), data structures, code base, and data interfaces may be seen as examples for that.

While for above mentioned reasons, data structure unification across modules is impossible (or, to say the least, impractical), a unified, language-independent, abstracted view of data objects is inevitable for inter-module communication and data exchange. Serving both physical and numerical requisitions, such a view has to be flexible and self-descriptive, and should encapsulate data along with its description and metadata. Mechanisms for safeguarding data integrity add to these requirements.

Concurrency in Component Development
Due to models and algorithms in numerical simulation being ever-evolving, programme code and resulting components evolve along. This requires an option for different modules' concurrent development based on reliable architecture of service utilities, data communication and control structures.

Programming-Language Interoperability

Integration of different modules into a whole often sees itself confronted with a variety of programming language preferences. As long as different methodologies shall be allowed to exist in parallel in a software framework, it has to be guaranteed that the adoption of different programming languages is enabled by design. Different naming and calling conventions pose the major obstacle in this case. Indeed, engineers and computer scientists use their favourite coding languages on purpose, since Fortran, C++, Java or Python all have their strengths in certain aspects. In order to align all of these for working along smoothly, a common practice for interoperation without sacrificing objectorientedness of interfaces needs to be created and applied.

Complexity of Coupling Schemes

Simulation frameworks as the example described in [32] enable each physics module to solve in its own domain, and to exchange jump conditions at defined interfaces between these domains. When two or more different physics' solvers interdepend on each other, some coupling scheme of implicit nature is necessary for deciding whether another iteration is necessary or the results obtained are already correct. Operations such as transferring data between different meshes or data manipulation via interfaces are rendered necessary for such coupling. Yet, they are independent from single physics' modules and have to be made available by specialised service functionality, so-called wrappers. A central controller module orchestrates the whole process.

Plug-and-Play Capability

A numerical simulation application may as well consist of different modules that supply similar functionality based on different methodologies. Such a redundancy can be seen as to be allowed by intention for various motivations. Flexible selection of different methods for different physics or for certain demands, verification through result comparison, and the integration of third-party modules would add to these.

Particular attention should be drawn to the concept of dummy modules or black boxes that offer certain computation functionality to other modules for enabling them to run standalone. Also, code integration and verification can be facilitated through these. Although the associated implications on the controller functionality need to be kept to a minimum, the framework design must support these requirements.

Debugging and Performance Tuning

Although generation of test cases and test vectors are part of preparation work prior to actual code generation, debugging and testing add up to a large portion of the time consumed for general software development. Additionally, performance tuning for high-performance applications has to be taken into consideration in this calculation. For system integration, these efforts are just as considerable and demanding – even though individual modules may have been accurately tested – as a secondary source of faulty behaviour is introduced through interaction between subsystems. Identifying errors originating from this source is often quite challenging, especially when third-party software has to be integrated and neither access to source code nor application specialists is given.

A simulation software framework, therefore, has to provide debugging features that enable users to trace bugs to individual modules effectively in order to have experts taking adequate counteractions. Performance analysis and identification of shortages or bottlenecks are, in the same manner, the key to efficiency enhancements of multiple kinds.

Checkpointing and Restarting
In spite of ever-increasing computing capacity of large scale systems and personal equipment as well, simulation of ample problems oftentimes requires days or weeks of processor time. Still, computer systems cannot be guaranteed to work stable enough for such a period. Then again, for platforms shared among multiple users, booked time may have been consumed before all computations have converged and come to some result.

Therefore, it is advisable to offer functionality for saving a snapshot of the current system state. This would enable the user to restart a simulation run later on or using different computing infrastructure. Implicitly or weakly coupled physical modules also have to store their internal states in order to restart iterations as mentioned above.

Consistent functionality for obtaining system snapshots and for restarting operation hence is a significant requirement. Providing this functionality to individual modules adds to this demand.

2.2 Data Structures

Summing up the afore mentioned challenges in software engineering, it soon becomes clear that within software integration frameworks careful design of data structures plays a key role in meeting these challenges.

This is especially true for voluminous packages of data of similar nature, such as model meshing or, even more, calculation results. Years ago, the only way to cope with these huge amounts of data was to make use of programming languages' low-level I/O functionality for storing the data as files to physical media. Due to file I/O operations being tightly coupled to the operating system, the data structures used could not easily be referred to as being open and self-descriptive [54].

Therefore, for model and result data of numerical simulations, more productive, powerful or efficient data structures have been researched into [1], [13]. Both eXtensible Markup Language (XML) and Hierarchichal Data Format (HDF) have proven most prominent for storing and exchanging data across operating system or computing infrastructure boundaries [75], [54].

As search operations within the result set of numerical simulations account for the largest part of a postprocessor's operating speed, organizing the data to be searched through in reasonable structures promises to accelerate this process eminently. In [75], an XML tree structure is proposed for this operation. XML is a self-descriptive data description language, forming documents of both logical and physical structure by the mere use of textual characters [9]. Being based on simple text, it is perfectly cross-platform capable as well. In this context, extensible would mean that it is possible for software designers to expand the data definition space at their discretion, thus forming a meta-markup language without any fixed set of descriptors.

Despite all freedoms offered by XML with regard to content and content characterization, it is bound to some quite strict grammar [74]. This grammar offers enough strictness to create XML parsers for any given code base, thus again adding to its platform indepence.

In the world of engineering, HDF has likewise gained more and more supporters. In addition to the five basic reasons demonstrated by [54] and summarized below, the Hierarchical Data Format has, in its 5th version, also advanced to present itself as both fast [22] and remotely accessible from distributed architectures [62].

The tree structure of HDF

Similar to XML data structures, HDF is organized as a data tree, the term ”hierarchichal” in its name explicitly denotes that fact. In computer science, tree-shaped data structures are well known [36], and have proven to be robust even when holding huge amounts of data [1]. Furthermore, there is a need for a formalism to reflect a simulation’s design and data space, which is motivated by [54]:

> Simulations often have many equations, scalar values, and fields of values; some are input parameters or constants, while others are results that depend on this input. Such values explicitly and implicitly define the context for your simulation. With huge amounts of interrelated data, you need a conceptual model to help organize and manage data – first in your mind, and hopefully on the computer thereafter.
> [...]
> There’s no better organization than yours, because you know your simulation and your application, as well as its algorithm and required parameters. But if your application will be used more than once, you have to write all this down – both so you can remember it and so you can share it with other scientists if the need arises.

The data tree’s root element would, accordingly, contain the simulation’s global information, while the actual tree structure is then built of grouped data, that can hold more groups and so-called data sets themselves. Metadata and attributes facilitate handling and distinguishing of groups.

Middleware for Numerical Simulation

A software layer that is inserted between the operating system and applications building on it is called middleware. In that context, HDF may be seen as some middleware that provides storage services for large sets of data, usually being numbers. Now that the organization of this data is determined to be a tree, the actual storage of the data that forms the leaves of this tree has to be taken care of. HDF offers all functionality for handling numbers in huge multidimensional arrays, along with sophisticated virtualization of the physical layer. This is needed to uncouple data representation from whatever kind of file system the data is stored in. Especially when a need for spreading data across multiple file systems arises, sound virtualization stands as an imperative.

With programming interfaces (APIs) available for all major programming languages and a couple of functionality for parallelization, the range of application has even been widened for HDF5.

Portability, Code Maintainability, Compatibility
HDF safeguards portability through a set of descriptive meta data, and thereby sheds light on type sizes, compression, byte ordering and more. With all this information being part of the actual file, not only data sets with fixed interpretation rules can be exchanged, but also a full context system including complete data. This context along with the interpretation of transported data is made available through a couple of functions within the API, thus ensuring portability of the data saved to a HDF file. HDF implements both backward and forward compatibility.

This portability is extended even more when taking the data format's compatibility with other operating system platforms, code libraries and applications into account. Such compatibility facilitates the use of inhomogeneous infrastructure, making use of vectorial computers, Linux clusters, or GPUs and the like. Similar to other open source projects, also updates of installed applications with a new version would therefore be possible without having to change API calls. The designers of HDF strictly adhere to the concept of incorporating both forward and backward compatibility as versions advance. Table 2.2 explains how these terms may be interpreted.

	Format	**Software**
Backward	Newer library versions can read a file and/or objects within a file that were created or written by older versions.	Applications written to work with a newer library version will compile, link, and run as would be expected with an older version.
Forward	Older library versions can read a file and/or objects within a file that were created or written by newer versions.	Applications written to work with an older library version will compile, link, and run as would be expected with a newer version.

Table 2.2: Backward and Forward compatibility as understood by [20]

The Open System Option
With simulation demands constantly increasing in complexity, adequate frameworks are often created by pulling together software from different sources – in-house developments, commercial or open-source software. Each of these features its own input-output system, while code-to-code data exchange remains a necessity. Aggregation of different players therefore can be implemented through direct one-to-one translation functionality, capable of interceding between two modules. This approach will, traceably, result in tremendous efforts

each time a new software module is added to the framework, as for each connection a new data translator would have to be created.

Keeping such a system open on base of a common abstracted data model would render single translators unnecessary. The introduction of an established public file format not only would reduce the number of direct code-to-code connections, but also keep single modules easily exchangeable. Though, all modules to be added would have to be able to read and write HDF files. HDF is frequently used as such a data model, as many tools handle it natively. Those which do not, can nicely be included using wrapper functionality [58]. Still, the newly aggregated framework system is kept open, and each new player can be connected to it with manageable efforts.

Trustworthiness and Support
The standard HDF is based upon is wide spread both in terms of geography and disciplines in science and engineering. A broad spectrum of tools utilizing HDF and tools and platforms of guaranteed functionality are available today, proving the software library's maturity and stability.

For a conclusive summary, it can be noted that HDF consists of three parts that make it a eligible candidate for utilization within software frameworks [20]:

- The HDF5 file format, a specification for the bit-level organization of an HDF5 file.
- The HDF5 data model, consisting of the building blocks for organizing and storing data.
- Software for managing, analyzing, acquiring, and querying HDF5 data. This includes an open source HDF5 library, a set of command line tools, and a large number of third party applications.

2.3 Programming Languages for Software Frameworks

Software reuse forms a major goal in framework creation, it does so likewise an abstraction layer lower for modules' programme code writing. Software architecture elements such as libraries or object orientation, or visual programming have improved the programmers' efficiency by magnitudes.

In order to create a programmable environment which aggregates software components on a high level, a couple of approaches have been investigated by [59]. For reasons to be discussed in the due course, interpreted programming languages turned out to be of particular advantage to that undertaking. Python has been selected for closer examination within that context:

> We settled on Python for a number of reasons, including: its concise and almost pseudocode-like syntax; its modularity; its object oriented design; its

> profiling, debugging, reflection, introspection and self documentation capabilities; and the availability of a numeric extension allowing the efficient storage and manipulation of large amounts of numerical data. Python is as good a glue as any other interpreted language but in addition it can be used to develop substantial extension components.

Python is available for all current operating system platforms, and through its interpreted nature its features and services are identical to all of them. This is true for interactive usage, for stand-alone scripts or large programmes, and as well when used as extension language for existent applications [72]. Furthermore, Python itself is open to extension through modules written in C or C++ [55].

Compared to other interpreted or compiled languages that are commonly employed within software framework designs, Python offers a couple of vantage points. It is easier to read, write and maintain than Perl code, programme code produced with Python is substantially shorter than C code, and in comparison with TCL, larger or more elaborate programmes are represented in a still comprehensible, inornate way.

In the engineering world MATLAB is, on the other hand, an omnipresent player in any profession used for a vast multiplicity of purposes. While a widespread impression may take it as yet another programming language, this has to be regarded as being only half the truth. MATLAB - the **MAT**rix **LAB**oratory - offers in fact a fully integrated development environment (IDE) for technical algorithm design. A mathematical shell for interactive calculations or debugging purposes and a source code editor for more complex scripts or even applications cater for any user interaction need. Code, file and data management and debugging is taken care of by various tools that are expected to be part of an integrated development environment. A vast collection of plug-in libraries, so-called toolboxes, have functionality for almost any engineering discipline in store [68]. The current version of MATLAB is 7.11 (called R2010b), and is available for Windows, Linux and MAC platforms, each in their 32 bit or 64 bit flavours. Its code base is formed by C and Java, and it is delivered under proprietary licensing [37].

For developing a numerical optimization framework MATLAB is able to supply a couple of functionality packages, obtainable by only little endeavours:

- data visualization (2D and 3D graphics)
- data analysis
- built-in debugging
- profiling (i.e. testing for efficiency)
- interactive algorithm development
- native HDF/HDF5 support

The programming language offered by MATLAB adhers to several different programming paradigms. As a start, MATLAB code can be produced in an *imperative* way, meaning that it is formulated in *statements* each of which changes the programme's *state* during execution. Also, it can be of *procedural* nature, a term which is often used interchangeably with being imperative, with the additional value of the *subroutines* concept being introduced. That means at any point of the programme, a fully swapped-out series of commands can be executed.

One of the major differences to other popular programming languages is MATLAB's characteristic of being an array programming language. That means an array is also carried as a *basic type* of variables in exactly the same manner as it is the case with e.g. integer or floating point variables.

Starting with 7.6 (R2008a), MATLAB now also extensively supports *object oriented programming*, which, together with its IDE functionality, makes it an interesting candidate for prototype developments. Properties, methods, and events with associated event handling are fully supported architectural components by now.

Surprisingly enough, though being an interpreted language, MATLAB in its current development status is claimed to run specific tasks faster than C or C++ [46]. This fact may well be derived from MATLAB being based on Java in large portions. Java, in return, has already been seen as being superior to C/C++ and others in certain specialized fields of application [8], [10].

Chapter 3

Objectives

“Failure comes only when we forget our ideals and objectives and principles.”

[Jawaharlal Nehru]

At the institute for Fundamentals and Theory in Electrical Engineering, a large part of current research activities revolves around different topics in the field of numerical optimization [25]. As representative examples, Evolution Strategies as in [2] or [39], and their application in industry ([14], [63], [26]) and education ([28], [31]) can be cited.

Facing the fact of inadequate support through software tools that could assist in obtaining sound conclusions for the questions and problems under investigation, a set of functionality requirements for such a tool was compiled.

Sought for is the creation of a

- coupled
- transient
- multi-conductor
- multi-objective
- optimization system
  - following a modular software architecture
  - allowing for multiple solver kernels to be triggered
  - in a client-server environment
    - with definable user access rights
    - offering output functionality suitable for web-browsers and
    - accepting input via some special client or web application

For controlling the optimization process within such a system, some separate toolbox has to be created. The functionality of such a toolbox can be summarized with the following requirements:

- problem description in near-standard language or as a geometric sketch
- able to optimize a problem's...
  - geometry
  - materials' composition using a material properties database
  - boundary conditions
  - sources
- within definable constraints and
- following a certain strategy
  - of a given definition
  - using appropriate quality functions, where applicable, and
  - discontinuing on reaching some provided stopping criterion

For the framework to be developed, the system boundary is set at taking over commands for solving a certain coupled problem, and at the delivery of the current solution to the optimization control again, thus offering full connectivity to an optimization toolbox. Fig. 3.1 shows an exemplary block diagram, containing both actors (coloured rectangles) and data to be exchanged between these (rounded rectangles). On the contrary, the optimization control functionality is denoted as an orange pentacle.

As the input for the problem setup should be any given problem that may be subject to optimization, data interoperability and data exchange are the main principles to be followed. Therefore, as a format for exchanging and storing data, eXtensible Markup Language XML was chosen [9]. XML bears a number of advantages to proprietary data formats in exactly these fields [49]:

> XML is a language for specifying semi or completely structured data. It has been explored over and over by both research and industry communities. More than ten years after its proposal, its initial expectation of "solving all the problems in the world" has not been fulfilled. However, we cannot say XML failed, since it solved some really important problems very efficiently. Some of those problems include data integration, data interoperability, and data publishing on the Web. Moreover, XML is adopted as a standard language by many industries (from retail to healthcare) for exchanging data.

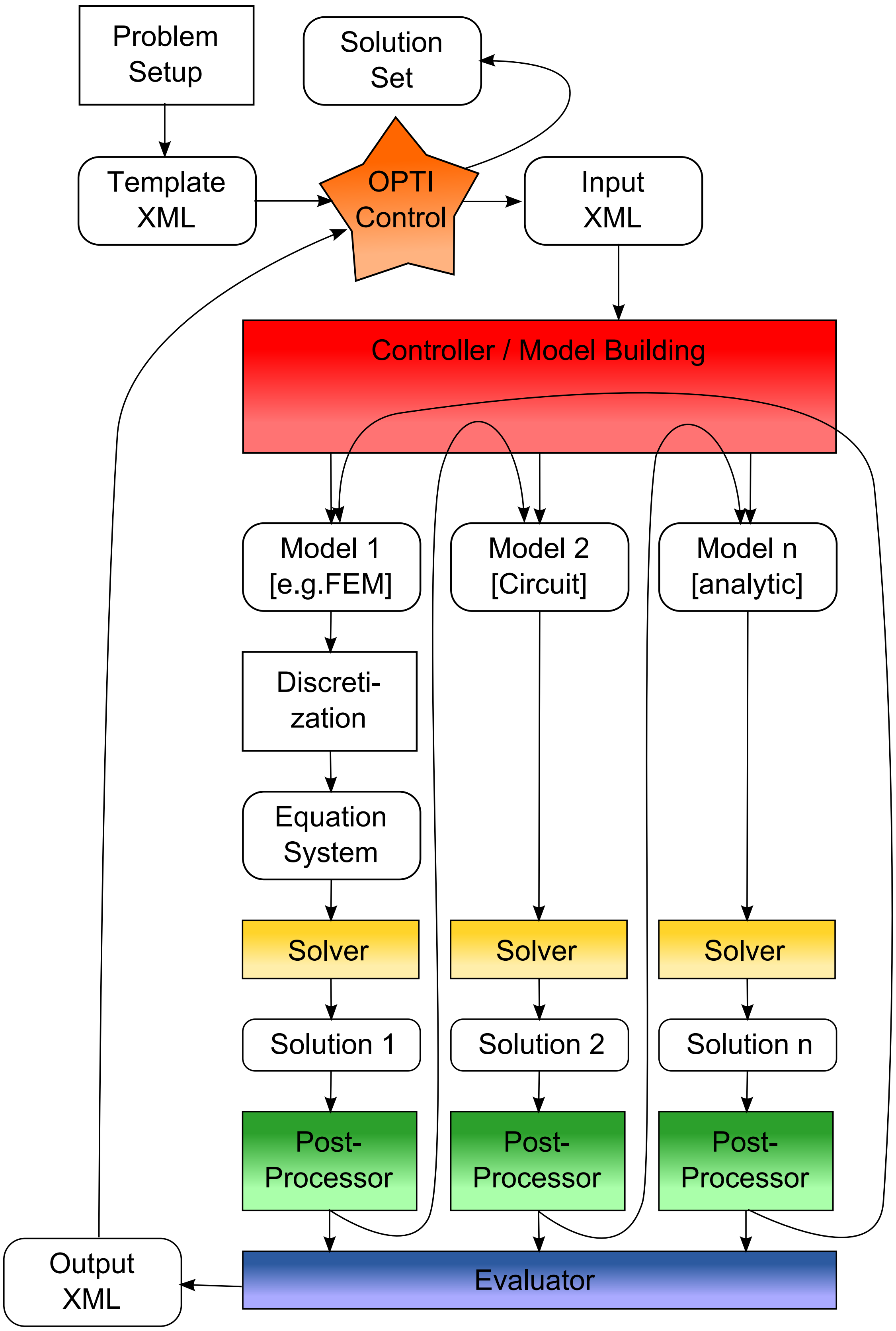

Figure 3.1: Block diagram of the ideal optimization framework

Among others, it is for these reasons that today XML is seen as to be the most successful, ubiquitous technology for the Web, at the same level as URI, HTTP, and HTML [74]. This may well be the merits of the main objective stated in [9], namely for XML being *"straight-forwardly usable over the Internet"*. Others of the objectives to be found l.c. strongly support the design decision of making use of XML as data exchange format as well. These are:

2. XML shall support a wide variety of applications.
6. XML documents should be human-legible and reasonably clear.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

The tenth design goal, on the other hand, apparently already foreshadows XML's major drawback. When conciseness is not a major concern, applications or programmes dealing with XML data do not have to implement sophisticated data compression or packing. This makes working with XML coded data quite straight forward on both sender and recipient side of the communication channel.

In addition, the little impact of verbose data markup allows for any arbitrary extension of the data to be worked on, which nicely plays to extending the framework in the future in any dimension. Unfortunately, voluminous data collections with a large quantity of data quanta may get hard to handle, apart from the mere data volume that can easily be multiplied in comparison to proprietary data formats.

This is, for example, especially true for matrix or solution data. Therefore, other data formats have to be found. For these it is necessary to consume significantly less memory space and/or computing time, play well with XML on input and output terms, and be easily accessed and handled.

As a last field of requirements, modularity and module communications shall further be expounded. The concept of software modularity, allows system designers to build complex products from smaller subsystems. These subsystems may be designed independently, but still work together to form a larger whole [6]. Additionally, modularity allows for shorter development cycles while still improving both flexibility and comprehensibility of a system [52] Albeit, the most important argument to value software modularity that much may be that it gives designers the freedom to experiment with different approaches. The obedience of introduced design rules is the only rule in that context [65]. For the system under scrutiny that would mean that single components shall be exchangeable by such ones that respect the same input and output rules - but 'do their thing' in an alternative way.

Chapter 4

Framework and Components

"Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience."

[Roger Bacon]

Based on the objectives established in chapter 3, a major part of research was focused on pulling together a working prototype offering according functionality. The result consisted of a system that allows verification of this prototype and its objectives against real-world examples.

The next sections describe the engineering aspects around design and implementation of such a prototypic software system as well as a selection of novel realization details.

In the engineering world, MATLAB is an omnipresent player in any profession and is used for a vast multiplicity of purposes. Therefore, a wide number of so-called toolboxes with standardized interfaces exist. The combination of this fact with offering a fully integrated development environment (IDE) for technical algorithm design, even allowing for object oriented design, led to the decision of selecting MATLAB for prototype development.

Depending on the application area, data representation follows one of three approaches. The finite element analysis makes use of certain parts that can be seen as objects, such as matrices, vectors, meshes and elements. Therefore, concepts of object oriented design and programming are employed [43]. That means for example that a computation's results are kept in a computer's memory represented by objects.

Persistent yet accessible storage of large data sets, on the other hand, requires employment of appropriate strategies. The Hierarchical Data Format HDF as introduced in chapter 2.2 has proven to largely comply with these demands, and therefore interfaces with HDF have been created. Together with XML officiating as the data exchange backbone between the framework and the outside world as well as between actors within the framework, expedient data representation models have been identified and implemented for each use case.

Details about potential and limits of an XML data representation are discussed in chapter 4.6, while chapter 4.3 shows how the system's functionality avails itself of employing HDF features.

4.1 Controller

The Controller offers the main interface to the outside world. It is also the main instance that governs the handling of the full process between input and output data as depicted in figure 3.1.

All other actors within that process are fed by data provided through the controller, and are also started and queried for results by this command structure accordingly. It therefore makes use of the overall input as an XML string and other models' postprocessing results as defined in chapters 4.6 and 4.7. The input is split up into single models' data which is, where necessary, supplemented by data retrieved from other models' results in order to consequently yield self-contained and fully-fledged model data that can be forwarded to model solving functionality of whatever nature.

This linkage of different models defines the weak coupling process mentionend in chapter 3. Most prominently, this will be the coupling of two finite element method's models, but the current functionality is not limited to these. For the simulation of electrical circuits, for example, active parts are typically presented as equivalent circuits [34]. As such equivalent circuits often do not provide sufficient information about the behaviour of the replaced system, the field equations are solved numerically and then coupled with the circuit equations [50]. In order to fulfill these demands, the circuit can either be modelled in external applications, or through inclusion of the respective logics using the Python interface of chapter 4.2.2.

As already foreshadowed, coupling of models can result in circular dependencies between them. This may, for example, be the case when electric or magnetic field intensities or temperatures derived by one model influence material properties of another. Similar examples arise when mechanical forces act on parts of the setup which are therefore displaced and result in a new problem topography. All these entail recalculation of the full setup. The controller automatizes this need. Figures 3.1 and 4.1 respectively delineate this fact with simulation data being fed back to the controller and possibly to the next model. It is one of the controller's main responsibilities to monitor suitable stopping criteria in order to halt the process once stability is reached. This stability can be tested on parameters that are interchanged between models both in absolute numbers and percentages.

Although weak coupling may increase the overall computation time by having to let circular dependencies converge, it offers great flexibility in model design. An interesting option that arises due to this approach may be seen as the possibility to use different meshes for differend finite element models. This is especially beneficial when thermal problems have to be simulated. Chapter 5.1 describes the coupling of such a problem, obtaining its

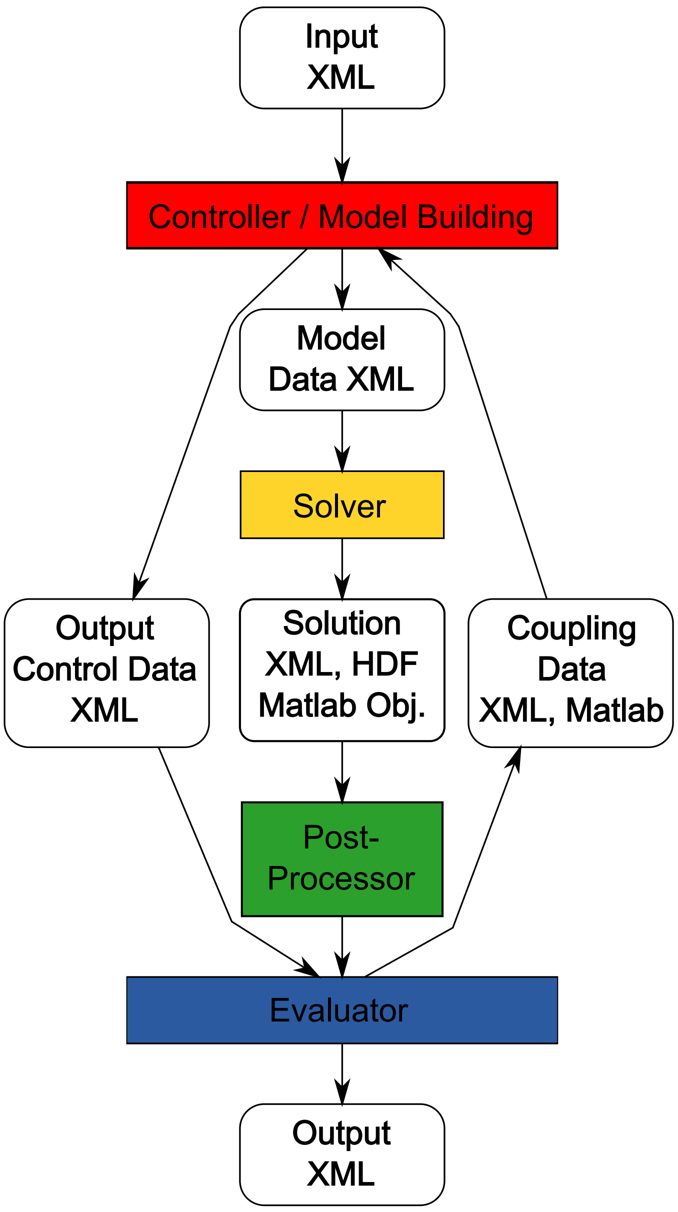

Figure 4.1: Data interfaces between framework actors

input for determining thermal sources from the results of a precedingly calculated current flow model.

Thermal finite element models can often be conveniently described by the use of boundary conditions, and their results are formed by the distribution of temperature values within the modelled domain. Therefore, the solution space can be obtained by interpolation from coarse meshing with sufficient accuracy. This provides the opportunity to save computation time through using different meshes.

As an example, the setup of the problem depicted in figure 5.1 may be seen. It describes the geometry of a electrothermal heating device along with a workpiece of steel. The simulation of this problem is done in two partitions, one calculating the current flow through the heating plate, the other using the ohmic losses within the heating plate as a source of heat for the thermal problem. Figures 4.2 and 4.3 show an according setup for both partitions. As both of these share the geometry of the horizontal heating plate, the domain occupied by it has to be part of both models, while the rest of the problem domain does not have to be modelled within both problem setups.

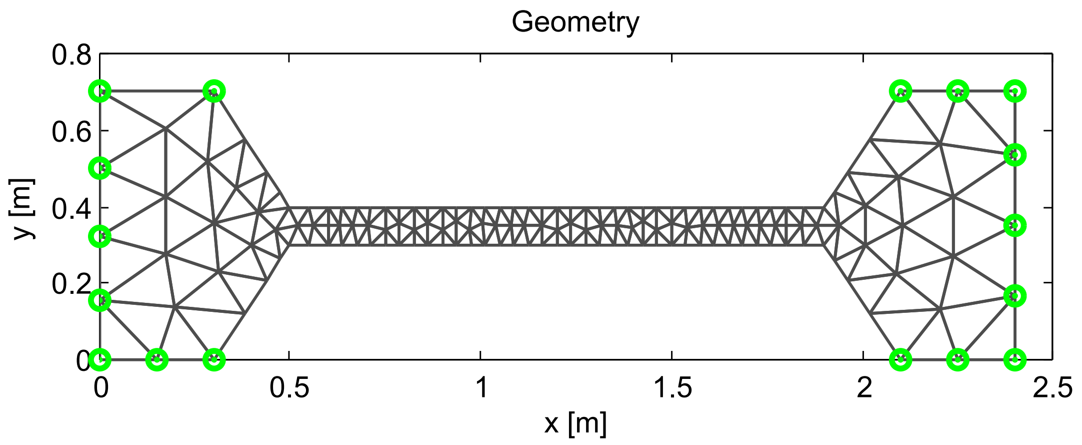

Figure 4.2: Mesh of the current flow model

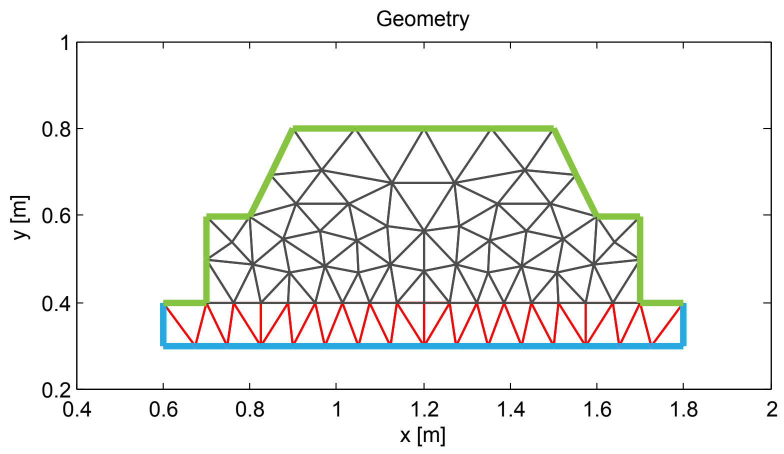

Figure 4.3: Mesh of the thermal model

The option of adaptive meshes also leaves room for being adjusted from outside by the optimization control. This is especially useful when geometric structures with strong curvature or bending, acute angles or eminent differences in adjacent materials' parameters have to be modelled. In such cases, simple static meshes with constant edge length for each element will not suffice for yielding a satisfactorily accurate discretization as singularities would probably come into existence. These in turn would lead to problems during mathematical processing and calculation. Global refinement of the meshing grid that would discretize these areas with sufficient precision would in most cases be somewhat counterproductive in terms of eminently rising memory consumption and computation efforts.

The idea of adaptive mesh refinement with respect to this is to apply finer discretization only and exactly at these places where higher resolution is needed [73]. By this, the area around the problematic realm receives new discretization with stronger weighting, while the singularity itself is accounted for with less weight.

Chapter 4.6.2 gives more details on how to prepare models in order to circumnavigate computational errors using different mesh refinement methods.

4.2 Solvers

4.2.1 FEM

4.2.1.1 The Ritz-Galerkin Finite Element Method

As a part of research conducted on the topic of creating a software framework for coupled simulation problems, a novel Finite Element Method solver was created [37]. FEMtastic, as it was called by the developers, in its current state of development allows obtaining solutions to electrostatic, thermal, static current flow and magnetostatic problems alike, offering support to problems that are either of two-dimensional nature or can be modelled by a two-dimensional setup. This can, for example, be achieved for rotational symmetric problems; cross sections through a problem geometry with an assumedly infinite third dimension would also qualify.

It implements the Ritz-Galerkin method for solving the continuous potential function $u(x, y)$ through means of discretization, owing to the fact that analytical solutions to such function are de facto nonexistent for problems that haven't undergone special conditioning. Therefore, the continuous potential function u gets approximated by the discrete potential function u_n. More precisely, this can be achieved through employing methods of variational calculus to the Laplace-Poisson equation as in (4.1). The basic idea will subsequently be explained by means of solving an electrostatic Laplace problem.

$$\text{div}\,(\,\text{grad}\,u) = \Delta u = -\frac{\rho}{\epsilon} \tag{4.1}$$

Therefore, relation (4.2) has to be solved, and it is the starting point of the following derivation.

$$\mathcal{A}u \overset{!}{=} f \tag{4.2}$$

In equation (4.2), $\mathcal{A}$ denotes a linear, symmetric and positive definite operator to a variable u, while f is an arbitrary function. According to variational calculus, assignment (4.2) can be solved by minimizing

$$W(u) = \frac{1}{2}\iint_{\Omega} u\mathcal{A}u d\Omega - \iint_{\Omega} u f d\Omega \tag{4.3}$$

Under the assumption of $\bar{u}$ being the exact solution to assignment (4.2), (4.4) follows:

$$W(\bar{u}) = \text{minimum!} \Longleftrightarrow \mathcal{A}\bar{u} = f \tag{4.4}$$

As already mentioned, for the continuous potential function $u(x, y)$ analytical solutions can be derived only for certain cases, the approximation of the continuous space through discrete space has to be accepted. The potential function u therefore is approximated by the discrete potential function u_n as in (4.5).

$$u \approx u_n = \sum_{i=1}^{n} a_i f_i + \underbrace{\sum_{k=n+1}^{n+m} u_k f_k}_{u_D} \tag{4.5}$$

For successful processing of (4.5), a number of issues arise that have to be taken care of subsequently.

- Instead of a continuous problem domain, only a limited amount of $m + n$ discrete points within a two-dimensional domain has to be considered. This paves the way for application of object oriented design patterns.
- The index k relates to those m points the potential values u_k of which are already determined by the problem definition, thus already bearing the solution value. These points are called *Dirichlet points* following *Dirichlet boundary conditions*.
- The index i relates to those n points the potential values a_i of which are unknown and therefore have to be computed.
  Stated more prominently, the determination of the coefficients a_i yields the solution to the given problem.
- Immediately following that statement is the fact that the problem consists of n unknowns. Therefore, n also denotes the degree of freedom of the problem.
- All potential values at continuous coordinates between this solution set of discrete points have to be retrieved by interpolating through using the basis functions f_k and f_i respectively. Role and usage of these and of the interpolation procedure will be discussed below.

Now, the next step would comprise inserting the discretized potential function (4.5) into functional (4.3). Two important derivatives will turn out to be useful for that:

$$\frac{\partial u_n}{\partial a_i} = \frac{\partial}{\partial a_i}\left(u_D + a_1 f_1 + a_2 f_2 + \cdots + a_i f_i + \cdots + a_n f_n\right) = f_i \tag{4.6}$$

$$\frac{\partial}{\partial a_i}(\mathcal{A}u_n) = \frac{\partial}{\partial a_i}\left[\mathcal{A}\left(u_D + a_1 f_1 + a_2 f_2 + \cdots + a_i f_i + \cdots + a_n f_n\right)\right] = \\ = \frac{\partial}{\partial a_i}\left[\mathcal{A}\left(a_i f_i\right)\right] = \mathcal{A}f_i \quad (4.7)$$

Insertion of (4.5) into (4.3) then yields

$$W(u_n) = \frac{1}{2}\iint_\Omega u_n \mathcal{A} u_n d\Omega - \iint_\Omega u_n f d\Omega \quad (4.8)$$

Through minimization of this functional, the solution to assignment (4.2) with the discretized potential follows:

$$\mathcal{A}u_n = f \text{ with } u_n \approx u$$

The coefficients a_i denote the unknowns which are to be determined. The functional has its minimum when all partial derivatives with respect to the coefficients a_i are zero, which is referred to as the *first variation* δ of the functional:

$$\delta(W) = \frac{\partial W(u_n)}{\partial a_i} = 0 \qquad i = 1, 2, \ldots, n \quad (4.9)$$

Therefore, the solution to the given problem is obtained once these n equations are solved. Further evaluation of (4.9) gives

$$\begin{aligned}\frac{\partial W(u_n)}{\partial a_i} &= \frac{1}{2}\iint_\Omega \underbrace{\frac{\partial u_n}{\partial a_i}}_{=f_i} \mathcal{A}u_n d\Omega + \frac{1}{2}\iint_\Omega u_n \underbrace{\frac{\partial}{\partial a_i}(\mathcal{A}u_n)}_{=\mathcal{A}f_i} d\Omega - \iint_\Omega \underbrace{\frac{\partial u_n}{\partial a_i}}_{=f_i} f d\Omega = \\ &= \underbrace{\frac{1}{2}\iint_\Omega f_i \mathcal{A} u_n d\Omega + \frac{1}{2}\iint_\Omega u_n \mathcal{A} f_i d\Omega}_{\text{Symmetry: } f_i\mathcal{A}u_n = u_n\mathcal{A}f_i} - \iint_\Omega f_i f d\Omega = \\ &= \iint_\Omega f_i \mathcal{A} u_n d\Omega - \iint_\Omega f_i f d\Omega = \\ &= \iint_\Omega f_i\left(\mathcal{A}u_n - f\right) d\Omega = 0 \qquad i = 1, 2, \ldots, n \quad (4.10)\end{aligned}$$

This can now be summarized as the Ritz-Galerkin equations as in 4.11:

Ritz-Galerkin equations

$$u_n = u_D + \sum_{i=1}^{n} a_i f_i \quad (4.11a)$$

$$\iint_\Omega f_i \mathcal{A} u_n d\Omega = \iint_\Omega f_i f d\Omega \qquad i = 1, 2, \ldots, n \quad (4.11b)$$

Taking into consideration that f_i are the basis functions and f is the right-hand side of equation (4.2), equation (4.11b) turns out to be equal to (4.10) with the known function f converted to the right-hand side.

After derivation of the Ritz-Galerkin equation (4.11b), the operator $\mathcal{A}$ can be substituted with the negative Laplace operator $-\Delta$ which is defined to be linear, symmetric and positive definite. The Laplace-Poisson equation can, within that respect, be denoted as

$$-\Delta u = \frac{\rho}{\epsilon} \tag{4.12}$$

On the other hand, applying the Laplace-Poisson equation to the Ritz-Galerkin equation (4.11b) yields

$$\iint_\Omega f_i(-\Delta u_n) d\Omega = \iint_\Omega f_i \frac{\rho}{\epsilon} d\Omega \qquad i = 1, 2, \ldots, n \tag{4.13}$$

With ϵ brought to the left-hand side and using $\operatorname{div}\operatorname{grad}$ as notation for Δ, the term transforms to

$$-\iint_\Omega f_i(\operatorname{div} \epsilon \operatorname{grad} u_n) d\Omega = \iint_\Omega f_i \rho d\Omega \qquad i = 1, 2, \ldots, n \tag{4.14}$$

In (4.14), ϵ denotes a tensor that can represent anisotropic materials. This relation has also been implemented into the software. Nevertheless, further considerations will treat ϵ as a scalar dependent on a position.

By the use of *partial differential integration* and *Green's identity* (as described in [33]), (4.14) can be transformed to

$$-\iint_\Omega (\operatorname{grad} f_i)^T \epsilon \operatorname{grad} u_n d\Omega = \iint_\Omega f_i \rho d\Omega + \int_{\Gamma_D} f_i u_D d\Gamma_D \qquad i = 1, 2, \ldots, n \tag{4.15}$$

and is, consequently, transformed further by substituting u_n with (4.5) to

$$-\iint_\Omega (\operatorname{grad} f_i)^T \epsilon \operatorname{grad} \sum_{j=1}^{n} a_j f_j d\Omega = \iint_\Omega f_i \rho d\Omega + \int_{\Gamma_D} f_i u_D d\Gamma_D \quad i = 1, 2, \cdots, n \tag{4.16}$$

For the reason of u_D being defined on boundary $\Gamma_D \subseteq \partial\Omega$ only and $\Omega \cap \partial\Omega = \emptyset$, it does not appear on the left-hand side of the equation. Index i is already in use within equations (4.10) and (4.11b). Therefore, (4.16) introduces j as the index for the approximated potential (4.5).

Taking the linearity of the Laplace operator into consideration, the sum can enclose the integral:

$$-\sum_{j=1}^{n} a_j \iint_\Omega (\,\mathrm{grad}\, f_i)^T \epsilon\, \mathrm{grad}\, f_j d\Omega = \iint_\Omega f_i \rho d\Omega + \int_{\Gamma_D} f_i u_D d\Gamma_D \quad i = 1, 2, \cdots, n \tag{4.17}$$

Summing up left and right-hand sides for all indexes i yields

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_j \iint_\Omega (\,\mathrm{grad}\, f_i)^T \epsilon\, \mathrm{grad}\, f_j d\Omega = -\sum_{i=1}^{n} \left(\iint_\Omega f_i \rho d\Omega + \int_{\Gamma_D} f_i u_D d\Gamma_D \right) \tag{4.18}$$

or, when presented in matrix form:

$$\mathbf{K}\mathbf{a} = \mathbf{f} \tag{4.19}$$

with

$$\mathbf{K} = \begin{bmatrix} \iint_\Omega (\,\mathrm{grad}\, f_1)^T \epsilon\, \mathrm{grad}\, f_1 d\Omega & \cdots & \iint_\Omega (\,\mathrm{grad}\, f_1)^T \epsilon\, \mathrm{grad}\, f_n d\Omega \\ \iint_\Omega (\,\mathrm{grad}\, f_2)^T \epsilon\, \mathrm{grad}\, f_1 d\Omega & \cdots & \iint_\Omega (\,\mathrm{grad}\, f_2)^T \epsilon\, \mathrm{grad}\, f_n d\Omega \\ \vdots & \ddots & \vdots \\ \iint_\Omega (\,\mathrm{grad}\, f_n)^T \epsilon\, \mathrm{grad}\, f_1 d\Omega & \cdots & \iint_\Omega (\,\mathrm{grad}\, f_n)^T \epsilon\, \mathrm{grad}\, f_n d\Omega \end{bmatrix}$$

$$\mathbf{a} = \begin{Bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{Bmatrix} \quad \text{and} \quad \mathbf{f} = \begin{Bmatrix} -\iint_\Omega f_1 \rho d\Omega - \int_{\Gamma_D} f_1 u_D d\Gamma_D \\ -\iint_\Omega f_2 \rho d\Omega - \int_{\Gamma_D} f_2 u_D d\Gamma_D \\ \vdots \\ -\iint_\Omega f_n \rho d\Omega - \int_{\Gamma_D} f_n u_D d\Gamma_D \end{Bmatrix} \tag{4.20}$$

Some conclusions can be drawn from these relations:

- Matrix $\mathbf{K}$ is an $(n \times n)$ square matrix. With respect to its representation in structural analysis, it is called the *stiffness matrix.*
- Vector $\mathbf{f}$ has n entries and is defined as being the *load vector.*
- Setting up the stiffness matrix and the load vector is known as the *assembly* of the finite element system of linear equations (*SLE*).

After definition of basis functions f_i, the system of equations is ready for being solved. The next paragraphs will further explain the necessary steps.

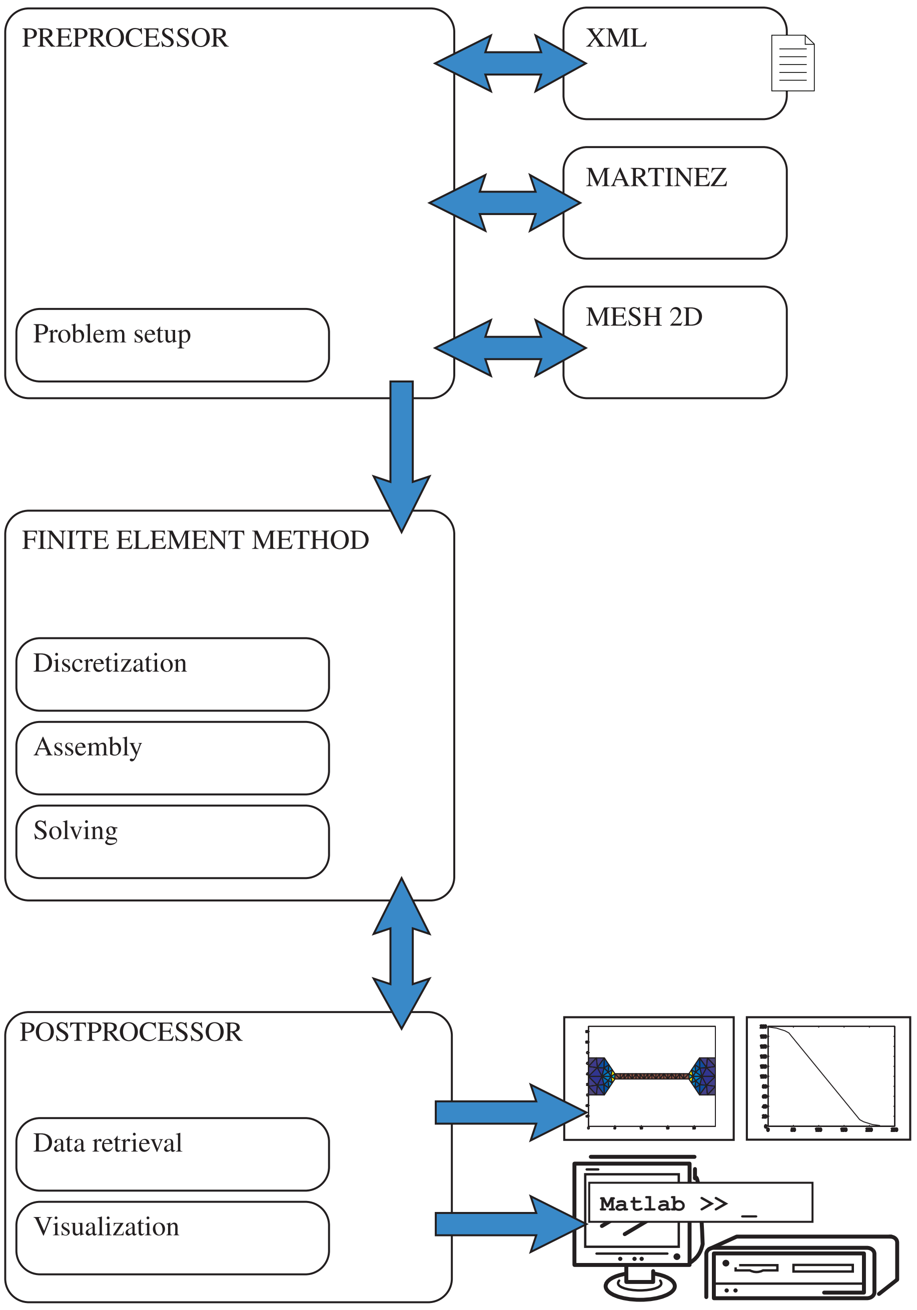

Figure 4.4: Modules and module interaction within FEMtastic

4.2.1.2 Implementation Details

4.2.1.2.1 Preprocessor In a typical FEM workflow, certain functionality is needed for reconditioning incoming data in order to retrieve data structures that can be forwarded to an equation system solver. Usually, this functionality is referred to as the preprocessor.

In the given case, the FEM solver's preprocessor is fed with a model's XML representation by the controller. Therefore, such a representation first needs to get itemized by an XML parser so as to incorporate the information deduced from the input.

A problem's geometry markup is composed of macro elements. An efficient polygon assembly algorithm takes care of any two-dimensional setup by applying boolean set operations to given polygons [44]. Engineered by F. Martínez et.al., it is referred to by the Martinez section within the system's block diagram representation in figure 4.4. Its functionality is defined as applying said operations – *union* (A or B), *intersection* (A and B) and *subtraction* (A and not B) – to the macro elements as defined within the XML file and passing back the finalized geometry to the preprocessor.

The final step performed by the preprocessor is generating a triangulation of the geometry; this yields a set of finite elements in the end. The MATLAB extension `mesh2d` [19] constitutes the groundwork for that functionality. It uses quadtree decomposition and Delaunay triangulation and produces meshes of very high quality [18], [53].

4.2.1.2.2 Finite Element Method The functionality block covering the Finite Element Method follows in the workflow's due course. While the present code of FEMtastic's FEM solver is technically prepared for any shape finite elements may assume within multiple dimensions, the focus of the following considerations is laid on two-dimensional triangular finite elements featuring linear basis functions. For static potential problems, the appendant model is defined within the domain Ω with boundary $\partial\Omega = \Gamma = \Gamma_E \cup \Gamma_D$ (the latter two within electrostatic problem descriptions).

As a consequence of analytical solutions being practically nonexistent within continuous space, the above mentioned triangulation algorithm is applied to discretize Ω into a delimited number of subdomains, so called finite elements $T^{(r)}$ with r being the unique index of each element.

This discretization procedure has to yield a result meeting the following prerequesites:

- The union of all finite elements should approximate Ω, being a continuous domain, to its best effort:

$$\Omega \approx \bigcup_{r=1}^{R} T^{(r)} \tag{4.21}$$

- Any two finite elements $T^{(r)}$ and $T^{(r')}$ must either share a common node, or edge, or nothing at all:

$$T^{(r)} \cap T^{(r')} = \begin{cases} \emptyset \\ \text{one common node} \\ \text{one common edge} \end{cases} \tag{4.22}$$

Basis Functions

The nodes derived by above mentioned discretization serve as exactly these points within Ω where a solution will be computed consequently. In order to correctly interpolate this discrete solution to any point within both continuous and discretized representations of domain Ω, the concept of form and basis functions is introduced.

Form functions are defined with in the local scope of a certain finite element, each node being assigned exactly one form function. A basis function, on the contrary, serves as the combination of all form functions a certain node in global space is assigned. It is commonly referred to as the *pyramid function.* All basis functions of nodes within a valid triangulation share the following main prerequisites:

- The function's value is 1 at the very position of the current node
- The function's value is 0 at the position of any other node
- The function's value differs from 0 only within the finite elements that share this node.

These basis functions will "span" the discrete solution over the global domain Ω_n, therefore superinducing a continuous solution again. Figure 4.5 depicts two neighbouring nodes P_{14} and P_{43} of a sample triangulation along with their (linear) basis functions. The finite elements contributing form functions to P_{14}'s basis function are accentuated in gray.

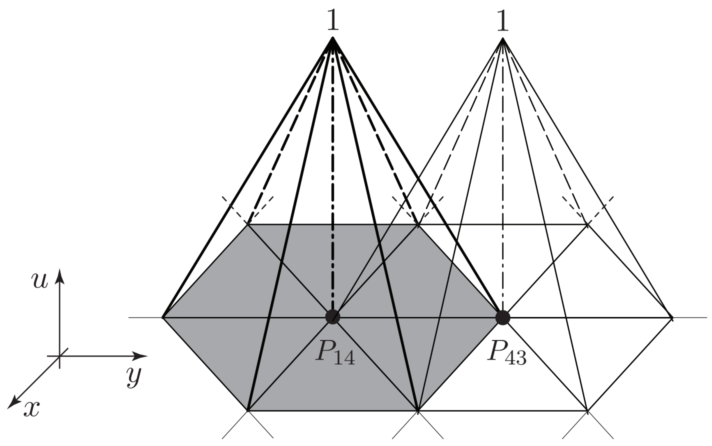

Figure 4.5: Basis functions

A finite element's shape within the global problem scope is obtained by a projection of a local reference element. Such a reference element or reference frame is, for triangular elements, defined by

$$\hat{T} = \{(\xi, \eta) : 0 \leq \xi, \eta \leq 1, \xi + \eta \leq 1\} \tag{4.23}$$

ξ and η denote the local coordinates in that respect. The left part of figure 4.9 depicts such a reference element in local space. In the case of triangular finite elements, each of the three local nodes $\alpha = \{1, 2, 3\}$ defining the reference element is assigned a form function as in (4.24).

$$\hat{f}_\alpha(\xi, \eta) = \begin{cases} \alpha = 1 : 1 - \xi - \eta \\ \alpha = 2 : \xi \\ \alpha = 3 : \eta \end{cases} \tag{4.24}$$

For obtaining the global coordinates (x, y) of the problem domain associated with a local coordinates (ξ, η) of the reference frame, a mechanism of coordinate transformation needs to be applied as delineated in equation (4.25).

$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{J}^{(r)} \begin{pmatrix} \xi \\ \eta \end{pmatrix} + \begin{pmatrix} x_1^{(r)} \\ y_1^{(r)} \end{pmatrix} \tag{4.25}$$

The Jacobian Matrix $\mathbf{J}^{(r)}$ in (4.25) is defined as in equation (4.26).

$$\mathbf{J}^{(r)} = \begin{bmatrix} x_2^{(r)} - x_1^{(r)} & x_3^{(r)} - x_1^{(r)} \\ y_2^{(r)} - y_1^{(r)} & y_3^{(r)} - y_1^{(r)} \end{bmatrix} \tag{4.26}$$

In (4.25) and (4.26), $(x_1, y_1)^{(r)}$, $(x_2, y_2)^{(r)}$ and $(x_3, y_3)^{(r)}$ depict the global coordinates of nodes 1, 2 and 3 of a finite element r.

Among other characteristics, for finite elements exhibiting linear basis functions, each element's Jacobian matrix $\mathbf{J}^{(r)}$ is computed at its initialization.

Finite Element Initialization

The FEM block's input consists of the geometry data provided by the preprocessor. In the first step the input data is being fed into congruous data structures, which means that for each node of the triangulation a *node object* is created. Each triangle forms a *finite element object*, and in terms of one finite element consisting of exactly 3 nodes, these types of objects are linked to each other appropriately. Some initial computations are also conducted immediately after the object's creation. For finite elements employing linear form functions, these are:

- The computation of the Jacobian matrix $\mathbf{J}^{(r)}$ defined by equation (4.26),

- the computation of the inverse Jacobian matrix $\mathbf{J}_{(\mathbf{r})}^{-1}$ and
- the computation of the Jacobi determinant $\det \mathbf{J}^{(\mathbf{r})}$.

These three characteristics have to be calculated only once for above mentioned type of finite elements during initalization, owing to the fact of being invariant. Further initialization of finite element objects would contain the determination of:

- the finite element's circumcenter in global space
- the radius of its circumcircle.

This is an important issue for performing the *inverse global-to-local transformation* and will be discussed within the description of the postprocessor.

Equation System Assembly

Assembly of the system of equations to be solved constitutes the next step. Stiffness matrix and load vector as stated in (4.20) are developed using *numerical quadrature* [37]. While Neumann boundary conditions are considered implicitly within the FEM system of equations, Dirichlet boundary conditions are incorporated using *homogenization*.

The left-hand side of equation (4.18) is:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_j \iint_{\Omega} (\operatorname{grad} f_i)^T \epsilon \operatorname{grad} f_j d\Omega \tag{4.27}$$

The continuous domain Ω has been discretized by a triangulation as in (4.21). Thus, the integration over Ω is approximated using all subdomains of the triangulation, these being the finite elements $T^{(r)}$:

$$\sum_{r=1}^{R}\left\{\underbrace{\sum_{i=1}^{n}\sum_{j=1}^{n} a_j \iint_{T^{(r)}} (\operatorname{grad} f_i(x,y))^T \epsilon \operatorname{grad} f_j(x,y) dxdy}_{=0 \text{ if node } P_i \text{ or node } P_j \notin T^{(r)}}\right\} \tag{4.28}$$

Considering the basis functions being different from zero only in exactly one node and within the region covered by the finite elements adjacent to it, it is sufficient to evaluate (4.28) only for nodes that belong to the current finite element r:

$$\sum_{r=1}^{R}\sum_{i^{(r)}}\sum_{j^{(r)}} a_j \iint_{T^{(r)}} (\operatorname{grad} f_i(x,y))^T \epsilon \operatorname{grad} f_j(x,y) dxdy \tag{4.29}$$

with

$$i^{(r)} = \left\{ i : P_i \in T^{(r)} \right\} \tag{4.30}$$
$$j^{(r)} = \left\{ j : P_j \in T^{(r)} \right\} \tag{4.31}$$

In equation (4.29), the basis functions f_i and f_j are formulated in global space with coordinates x and y. A formulation of these within the realm of local space would allow performing all computations by means of the local reference element $\hat{T}$. Therefore, a linkage between global and local nodes has to be introduced:

$$\alpha \leftrightarrow i = i^{(r)}(\alpha) \tag{4.32}$$
$$\beta \leftrightarrow j = j^{(r)}(\beta) \tag{4.33}$$

This means that for a certain finite element $T^{(r)}$ with three global nodes it is known which global node corresponds to which local node of the reference element.

Equation (4.29) can therefore be transformed into

$$\sum_{r=1}^{R}\sum_{\alpha=1}^{3}\sum_{\beta=1}^{3} a_\beta \iint_{\hat{T}} (\operatorname{grad} \hat{f}_\alpha(\xi,\eta))^T \hat{\epsilon} \operatorname{grad} \hat{f}_\beta(\xi,\eta) \left|\det J^{(r)}\right| d\xi d\eta = \sum_{r=1}^{R} \mathbf{K}^{(r)} \tag{4.34}$$

with $\mathbf{K}^{(r)}$ being denoted as the *element stiffness matrix* of finite element $T^{(r)}$:

$$\mathbf{K}^{(r)} = \sum_{\alpha=1}^{3}\sum_{\beta=1}^{3} a_\beta \iint_{\hat{T}} (\operatorname{grad} \hat{f}_\alpha(\xi,\eta))^T \hat{\epsilon} \operatorname{grad} \hat{f}_\beta(\xi,\eta) \left|\det J^{(r)}\right| d\xi d\eta \tag{4.35}$$

Analogously, also a *element load vector* $\mathbf{f}^{(r)}$ exists. Neglecting the boundary condition term on the right-hand side of (4.18) – $\int_{\Gamma_D} f_i u_D d\Gamma_D$ – and by employing local scope formulation, it can be transformed to:

$$\mathbf{f}^{(r)} = \sum_{\alpha=1}^{3} \iint_{\hat{T}} f(x(\xi,\eta), y(\xi,\eta)) \hat{f}_\alpha(\xi,\eta) \left|\det J^{(r)}\right| d\xi d\eta \tag{4.36}$$

where $x(\xi,\eta)$ and $y(\xi,\eta)$, respectively, denote the local-to-global transformation (4.25).

The association between local and global nodes is retained by the *element coherence matrix* $\mathbf{C}^{(r)}$. It is a $(\alpha \times i)$ matrix and is defined for every finite element r:

$$\mathbf{C}^{(r)}(\alpha, i) = \begin{cases} 1\text{, if } i \text{ is the global node number of local node } \alpha \\ 0\text{, otherwise} \end{cases} \tag{4.37}$$

with (α, i) being the row and column indices.

The stiffness matrix $\mathbf{K}$ can, therefore, be constructed as follows.

$$\mathbf{K} = \sum_{r=1}^{R} \left(\mathbf{C}^{(r)}\right)^T \mathbf{K}^{(r)} \mathbf{C}^{(r)} \tag{4.38}$$

The load vector $\mathbf{f}$, on the other hand, consists of:

$$\mathbf{f} = \sum_{r=1}^{R} \left(\mathbf{C}^{(r)}\right)^T \mathbf{f}^{(r)} \tag{4.39}$$

Having all these at hand, the system of linear equations as in (4.19) can be composed.

Integration of boundary conditions

Neumann boundary conditions Integrating the Neumann boundary condition term of (4.18) into the FEM system of equations changes the load vector $\mathbf{f}$ as depicted in (4.20) [33].

Like the whole domain Ω, also the boundary $\partial\Omega$ is partitioned into finite elements. For a two-dimensional Ω_n, $\partial\Omega_n$ consists of straight lines $E^{(e)}$, with E denoting an *edge* of the index e. Similar to two-dimensional finite elements, a reference element $\hat{E}$ exists, which is a one-dimensional straight line defined by a local coordinate ξ within the interval $[0, 1]$. It exhibits two local nodes at $\xi = 0$ and $\xi = 1$. Such a construction leads to an *edge element load vector* $\mathbf{f}^{(e)}$:

$$\mathbf{f}^{(e)} = \sum_{\alpha=1}^{2} \int_{\hat{E}} u_D(\xi) \hat{f}_\alpha(\xi) d\xi \tag{4.40}$$

In conjunction with an *edge element coherence matrix* $\mathbf{C}^{(e)}$ implicating the two local nodes of the edge to the global nodes of the problem discretization, the Neumann boundary conditions can be integrated into the FEM system of equations as in (4.41):

$$\mathbf{f} = \mathbf{f} + \mathbf{C}^{(e)T} \mathbf{f}^{(e)} \tag{4.41}$$

Dirichlet boundary conditions For the demonstration of how Dirichlet boundary conditions can be integrated, an assembled FEM system of equations with two unknown node values and one known Dirichlet node value is imagined:

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \cdot \begin{Bmatrix} a_1 \\ \bar{a}_2 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \\ f_3 \end{Bmatrix}$$

Assuming $\bar{a}_2$ being mentioned known value, it can be brought to the right-hand side:

$$\begin{bmatrix} k_{11} & 0 & k_{13} \\ k_{21} & 0 & k_{23} \\ k_{31} & 0 & k_{33} \end{bmatrix} \cdot \begin{Bmatrix} a_1 \\ \bar{a}_2 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} f_1 - k_{12}\bar{a}_2 \\ f_2 - k_{22}\bar{a}_2 \\ f_3 - k_{32}\bar{a}_2 \end{Bmatrix}$$

This step offers the opportunity to now delete the stiffness matrix' second line and column, and the load vector's second line. This yields the final system of equations, being reduced by exactly these lines for which a solution is already known:

$$\begin{bmatrix} k_{11} & k_{13} \\ k_{31} & k_{33} \end{bmatrix} \cdot \begin{Bmatrix} a_1 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} f_1 - k_{12}\bar{a}_2 \\ f_3 - k_{32}\bar{a}_2 \end{Bmatrix}$$

This process is usually referred to as *homogenization*. It follows the general approach of:

- Setting $a_j = u_D(x_j, y_j)$ for all indices j of known coefficients
- Correcting the right-hand side of the FEM system of equations:

$$f_i = f_i - \sum_j \mathbf{K}(i,j)a_j \qquad \forall i \neq j$$

- Deleting the rows j of $\mathbf{K}$ and $\mathbf{f}$ and the columns j of $\mathbf{K}$.

Problem solving

Solving the system of equations (4.19) is the last duty the Finite Element block has to fulfill. The solution set sought for, namely the coefficients a_i, is therefore retrieved by evaluation of (4.42). For reasons of efficiency, this fully relinquished to MATLAB native functions.

$$\mathbf{a} = \{a_1, a_2, \ldots, a_n\} = \mathbf{K}^{-1}\mathbf{f} \tag{4.42}$$

The solution arising from these calculations yields the so far unknown node potential values. These are offered to postprocessors and thus – via detour through the postprocessors – to other models as well.

In practice, problems easily exhibit numbers of several thousand to millions of coefficients a_i. In such cases, the solution cannot simply be obtained through inverting the stiffness matrix to $\mathbf{K}^{-1}$. While for small problems *Newton's method* might be sufficient to solve equation (4.42), larger problems or problems of higher dimensions have to employ *gradient descent methods* such as the *conjugate gradient method* or one of its varieties for solving the equation systems that arise.

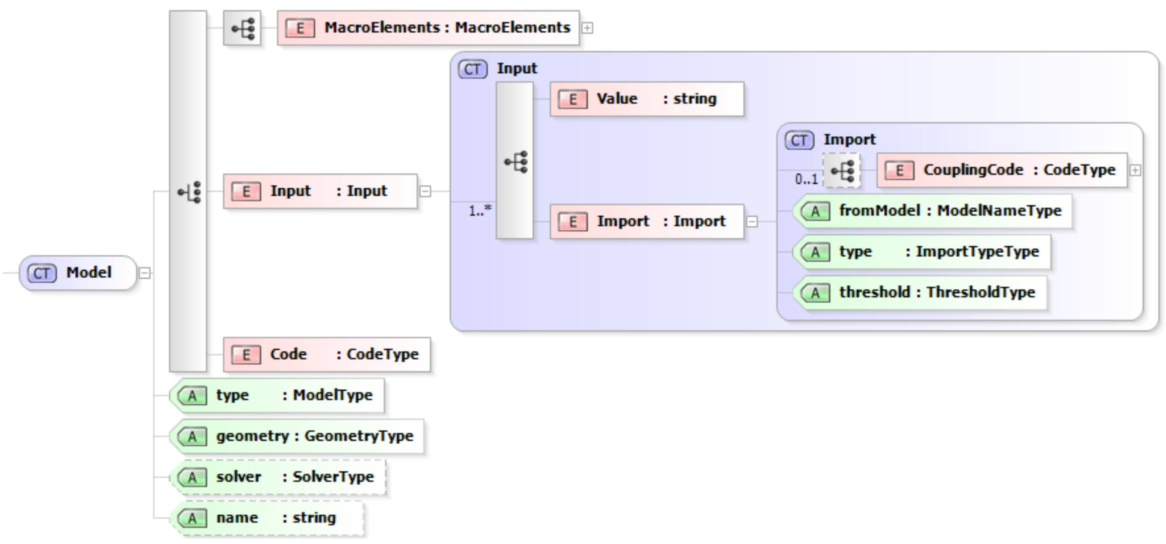

Figure 4.6: Data definition for addressing analytic solver code

4.2.1.2.3 Postprocessor The postprocessor of an application of the finite element method works as the interface between the derived solution and its further utilization by man or machine, providing data acquisition and visualization functionality. The postprocessor within FEMtastic is a self-contained module, offering direct access to methods implemented within the FEM module. Its range of functionality is discussed in detail in chapter 4.4.

4.2.2 Analytical Solvers - Python and Matlab

Usually, computer applications are distributed in some sort of self-contained way, i.e. they offer encapsulated functionality aimed at solving a given problem or fulfilling a given task. Some programmes provide extensions or even extendability with limited complexity; the potential sphere of operations is limited in both cases.

Therefore, a completely open approach was selected for the work under scrutiny, open in a sense of not binding the user to a newly created set of directives and therefore to predefined functionality. Rather, a scripting framework was added, which would allow fast prototyping of new functions or even applications, thus postponing code specialization as much as possible [59].

For achieving such a goal, interpreted languages were chosen because of their characteristics of flexibility, interactivity and extensibility. Matlab, as forming the backbone programming language of the whole project, lends itself to being employed automatically. Apart from that, Python was selected for a number of reasons. It offers concise and almost pseudocode-like syntax, modularity, object oriented design principles, and the availability of numeric extensions that allow for efficient handling of large volumes of numerical data [72].

Within the simulation framework under scrutiny, an interface for carrying out Python as well as Matlab code was created. The latter one emerges straightforward from the design decision of Matlab being the development environment for most of the project, Python on the other hand offers the above mentioned assets. Therefore, models that do not have to undergo FEM simulation can be solved using deterministic code using the interfaces to these two environments.

The XML schema in figure 4.6 shows how analytic solvers are deployed as a part of the global data structure. In case the *Model*'s attribute *solver* is set to either `Matlab` or `Python`, the respective interface is loaded. Therefore, the subsequent XML directives have to consist of both *Input* and *Code*. The values supported using the *Input* directive either as their explicit value or via the *Import* from another model are then supplied to the *Code* as a vector of variables, `input[]`. This array contains all the values forwarded in the order of their appearance. An example for this is given in listing 5.10, containing two explicit values and two values implicitly derived from model 1.

In the same manner, `returnValues[]` is reserved by the interface as means of data transport back into the framework again. It has to be seized by the solver's code and may contain any values that are worth being considered for further evaluation through any part of the framework. Thus, import into other models or creating output to the superimposed control structure are facilitated.

The *Code* section does not impose any restrictions on the actual code transported with it. For both of the programming interfaces currently supported – Python and Matlab – import mechanisms for outsourced code packages are available. Therefore it is advisable to keep the XML payload short and simple, and to only fill it with code portions that either call external routines or are bound to automatic refactoring by the mentioned control structure.

For Matlab, also a circuit solver exists. It takes over PSpice netlist files [69] from the file system and returns matrices for voltages and currents, respectively. The netlist can be altered for each simulation run by external functionality, but its inclusion into the simulation framework is granted without any constraints.

4.3 Storage of Intermediate Results

In many cases, the simulation framework under scrutiny may be used for cycling repeatedly through simulation tasks. That may either happen in horizontal direction, meaning that two or more models are coupled in a circular manner, interdepending on each other's results in certain ways. The circulation typically stops once stability is reached, and exactly one result set is obtained. Nevertheless, in some cases also provisional results may be of interest, such as for creation of chapter 5.1's convergence examination in figure 5.9.

On the other hand, also the vertical dimension may require repetition of simulation processes. This is the case when the whole framework is queried for its services multiple times by the superimposed control structure, such as an optimization toolbox. In that case, intermediary results are often used for determining the quality of the finally obtained one.

Even the combination of these two mechanisms is common, thus forming a two-dimensional

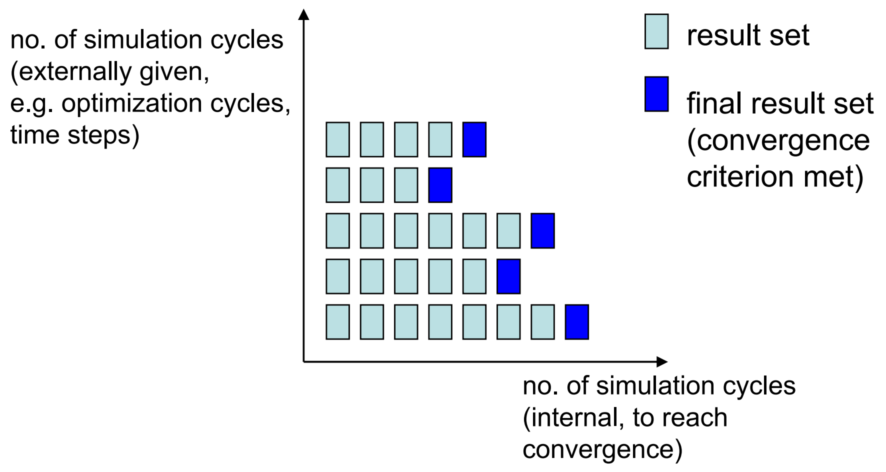
no. of simulation cycles (externally given, e.g. optimization cycles, time steps)
result set
final result set (convergence criterion met)
no. of simulation cycles (internal, to reach convergence)

Figure 4.7: Accumulation of result sets

simulation process. Chapter 5.3 discusses an example where an external control – the time-stepping algorithm – depends on results obtained through cyclic approximating and converging simulation processes, calculating a coils impedance as a function of amperage. Here, a set of solutions arises for each time step, thus resulting in a two-dimensional matrix of all retrieved solutions. Figure 4.7 explains how solution sets may accrue in the simulation's due course.

In all of these cases, access to such historical data may be of interest. The simulation framework stores all calculated data, and the evaluator offers said access to all of these using data storage mechanisms offered by HDF5 [20].

HDF stores data in hierarchically ordered data sets that make it particularly suitable for the two-dimensional superset of result sets. The object representation of finite elements, for example, stores all information that is necessary to recalculate any quantity that is asked for, without the need for any additional data compounds. All these data are inserted into a data set within a HDF5 file which bears the current simulation run as description within its meta data. The file itself then contains all solution sets that have accrued within the whole process, and is named using the unix timestamp of the moment when the last entry was stored. Therefore, for externally initiated process repetition, the vertical dimension keeps distinctly identifiable as well.

4.4 Post-Processors

Delivery of computational support in interpreting the solution is the postprocessor's main task. Therefore, it can be seen as an interface between the framework and the user, as well as it assists in interpreting the results and preparing data for further computations.

The postprocessor implemented for this framework prototype consists of a set of routines delivering access to data acquisition and visualization. Though being a self-contained module, methods offered by finite element objects are accessed by the postprocessor. Access to form functions and their gradients range among the most expedient features a finite element object exhibits. This is especially useful as externally provided FEM code may exhibit convenient access to such functionality as well, and therefore it wouldn't have to be duplicated as a part of the postprocessor module.

The most important functions offered by the postprocessor are:

- `Value()`: Calculates the solution value at any continuous coordinate of the solution space through interpolation using the finite element basis functions.
- `Gradient()`: Retrieves the solution gradient at any continuous coordinate of the solution space.
- `Data()`: Retrieves a sequence of solution data along a straight line. Line diagrams, for example, make use of such a functionality.
- `GradientData()`: Retrieves a sequence of the solution gradient along a straight line.
- Additionally, plotting functions are quite useful for data interpretation by the user. Geometry, solution values, solution gradient, and the gradient taking material properties into account can be depicted through these.

The implementation of functionality beyond these basic features is left to user-defined scripting code. Such an interface allows extensive usage of the calculated result set without being bound to prearranged methods. The availability of any physical quantity at any point of the problem domain is the basis to all such extensions. This can be reached using a mechanism of transformation between global problem space and local reference frame which closes the gap between globally defined problems and locally executed computations.

4.4.1 Transformation of Local to Global Coordinates

This mechanism of transformation retrieves the global coordinate (x, y) of the problem domain associated with a local coordinate (ξ, η) of the reference frame. This is obtained through application of equation (4.25), which is implemented as a function of each finite element object.

In (4.25) and (4.26), $(x_1, y_1)^{(r)}$, $(x_2, y_2)^{(r)}$ and $(x_3, y_3)^{(r)}$ depict the global coordinates of nodes 1, 2 and 3 of a finite element r.

Each finite element's Jacobian matrix $\mathbf{J}^{(r)}$ is computed at its initialization. For finite elements applying linear form functions the Jacobian Matrix is static, while for finite elements of higher order this is not the case. It depends on the local coordinates to be transformed, and therefore necessitates to be computed at every function call.

4.4.2 Transformation of Global to Local Coordinates

The common approach of obtaining the potential value u at any given global coordinate (x, y) from the solution space requires global-to-local coordinate transformation. This stems from the necessary step of interpolation using the finite elements' basis functions. Nevertheless, for several reasons the inverse coordinate conversion cannot be done by transforming (4.25) in order to retrieve the local coordinates (ξ, η):

- The finite element r that contains the point in space with coordinates (x, y) is unknown.
- For finite elements of higher order, the inverse transformation gets nonlinear because of the Jacobian matrix depending on the local coordinates $\mathbf{J}^{(\mathbf{r})}(\xi, \eta)$. A circular statement arises out of the inverse Jacobian matrix also depending on the local coordinates $\mathbf{J}^{-1}_{(r)}(\xi, \eta)$.

The first issue can be counteracted through employing the concept of circumcircles. For finite elements with linear basis functions, the second problem converges easily, for higher order elements a numeric method has to be employed. The gradient descent method fulfills the requirements quite well.

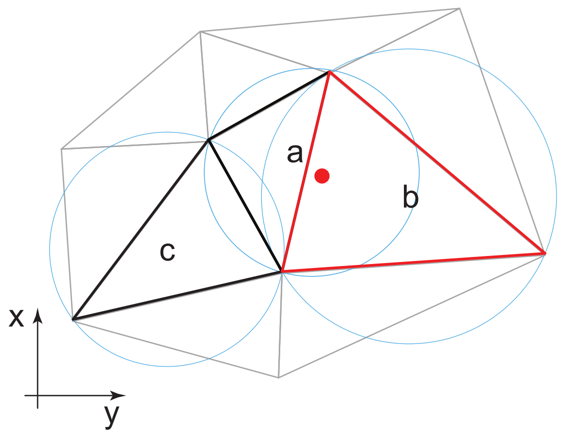

Figure 4.8: Circumcircles and nearest neighbour search within finite elements

In figure 4.8 a sample mesh triangulation is shown. Three finite elements and their according circumcircles depict the area of closer investigation. The red dot displays the point of global coordinates (x, y) that shall be transformed into the local scope.

As mentioned, retrieval of the finite element r that contains the point in space with coordinates (x, y) can be algorithmized through performing a k-nearest neighbour search using the finite elements' circumcircles:

```
k ← 1
while no local coordinate found do
  {Find the next circumcenter to $(x, y)$ and tell finite element r}
  r ← k-Nearest-Neighbor(k, $x$, $y$)

  if $(x, y)$ is within circumcircle of finite element r then
    $(\xi, \eta)$ ← localCoord(x, y) {global-to-local transformation}

    if $(\xi, \eta)$ is inside of reference element then
      {found $(\xi, \eta)$}
    else
      {continue search}
      k ← k + 1
    end if
  end if
end while
```

The circumcenter and the radius of the circumcircle is a member variable of each finite element object, and thus computed at its initialization. The k-nearest neighbour search algorithm then obtains the ”k-nearest” circumcenter to (x, y). In figure 4.8, the circumcenter of finite element a marks the first result in this respect.

Now, the actual location of (x, y) whether being contained in a or not has to be determined by executing a global-to-local transformation. This might be a quite expensive step in terms of computing time, as will be shown below. Albeit feasible, such a transformation is therefore not conducted on each single finite element, buth rather after some certain proximity has been established.

The global-to-local transformation of (x, y) pertaining to finite element a delivers local coordinates (ξ, η) residing outside of a's reference triangle, thus yielding a negative evaluation of the search criterion. The search for the correct local frame therefore has to be continued with the next nearest neighbor. Finite element b would be returned as the next search result where also a valid local coordinate can be obtained, thus breaking the search loop.

The global-to-local transformation strategy is depicted in figure 4.9. The red dot at (x, y) in global space (right) is given, while its equivalent in the local frame (left) is searched for. As this transformation is not straight forward, the correspondig local coordinates need to be approximated. Following the Newton-Raphson method, starting at local coordinates of $(0.5, 0.5)$, they are transformed to global space and tested against the given coordinates (x, y).

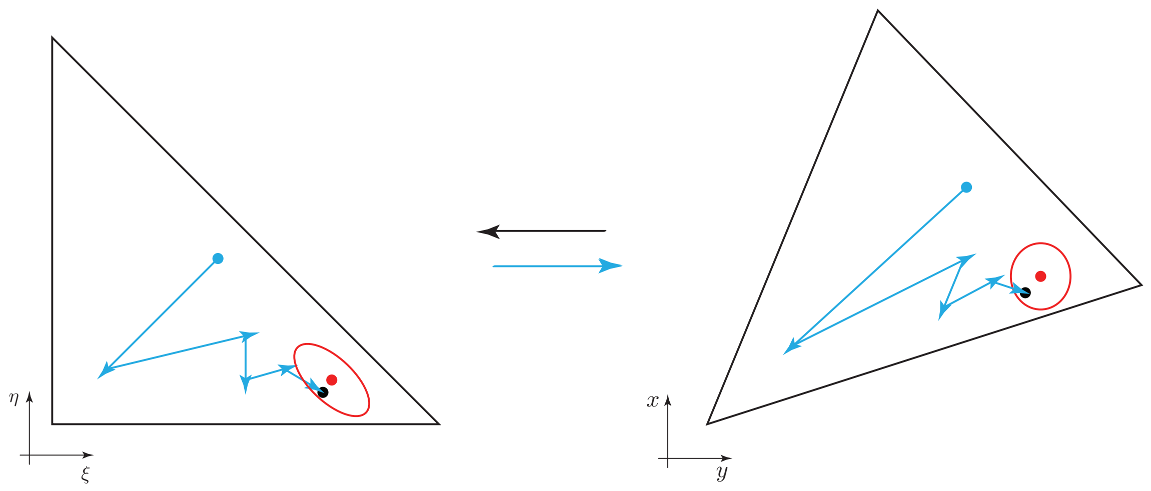

Figure 4.9: Finding local coordinates by applying the Newton-Raphson method

A stopping criterion in terms of a certain ϵ-neighborhood has to be accepted for reasons of efficiency in numerical determination of equality of two continuous figures. Depending on the test against the given coordinates, a new iteration of local coordinates is computed until finally the stopping criterion matches.

As a result, the local coordinates (ξ, η) are obtained, depicted as a black dot. Its global space equivalent resides within the ϵ-neighborhood shown as a red circle around (x, y).

The current implementation defines $\epsilon = 1 \cdot 10^{-6} m$, thus yielding sufficiently accurate results without major losses of efficiency.

Pursuing this algorithm is only necessary for finite elements of higher order. In the special case of linear finite elements, the mentioned static Jacobian matrix allows for a convenient transformation by inverting equation (4.25) into (4.43).

$$\begin{pmatrix} \xi \\ \eta \end{pmatrix} = \mathbf{J}_{(r)}^{-1} \left[\begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} x_1^{(r)} \\ y_1^{(r)} \end{pmatrix} \right] \tag{4.43}$$

4.4.3 Determination of a Solution Value

In figure 4.10, an arbitrary one-dimensional finite element in local space is shown along with a sketch of a solution interpolation query. At a global coordinate x, the value of the solution is asked for. After transforming x to the local coordinate ξ_A of the respective local reference frame, the values of the two basis functions at coordinate ξ_A are summed up, thus yielding the solution u_A as in (4.44). Figure 4.10 plots the dashed form functions of local nodes $\hat{P}_1$ and $\hat{P}_2$ with their associated, computed solution values u_1 and u_2, respectively. These two span the interpolated solution between the nodes, marked in red.

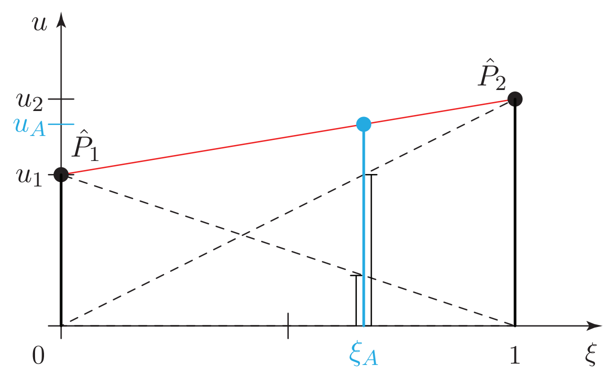

Figure 4.10: Solution retrieval by interpolation

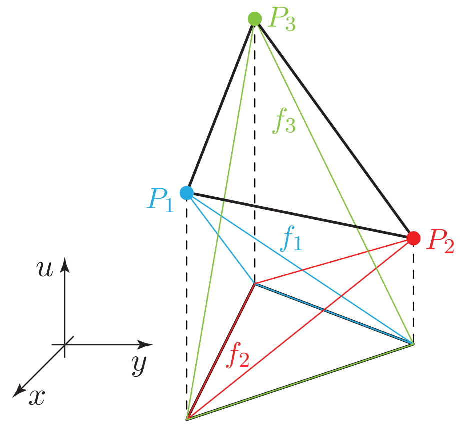

Figure 4.11: Triangular finite element in global space with linear form functions and spanned solution space

$$u_A(x) = \sum_{\alpha=1}^{2} u_\alpha \cdot \hat{f}_\alpha(\xi_A) = u_1 \cdot \hat{f}_1(\xi_A) + u_2 \cdot \hat{f}_2(\xi_A) \tag{4.44}$$

For triangular finite elements employing linear form functions, the same interpolation algorithm can be applied (4.45). The form functions for such finite elements typically consist of the set given in (4.46), and figure 4.11 accordingly shows a triangular finite element along with its linear form functions and the solution space supported by the nodal solutions.

$$u(x,y) = \sum_{\alpha=1}^{3} u_\alpha \cdot \hat{f}_\alpha(\xi(x,y),\eta(x,y)) \tag{4.45}$$

$$\hat{f}_1(\xi,\eta) = 1 - \xi - \eta \quad \hat{f}_2(\xi,\eta) = \xi \quad \hat{f}_3(\xi,\eta) = \eta \tag{4.46}$$

The output value of these interpolation operations depends on the nature of the problem solved. Its interpretations are listed in table 4.1.

problem type	interpretation	unit
electrostatic	electric potential u	$[V]$
current flow	electric potential u	$[V]$
magnetostatic	magnetic scalar potential Θ	$[A]$
thermal	temperature T	$[K]$

Table 4.1: Solution dimensions and their interpretation

4.5 Evaluator

The Evaluator's main responsibility is to generate output values on the basis of design parameters [4]. These serve, for example, as objective function values for evaluation of the obtained result by superimposed control structures.

Its functionality is more or less confined to executing the output queries that are extracted by the controller from the input XML as discussed in chapter 4.8. As it is necessary when dealing with the interface to analytic solvers – the Evaluator, in the end, offering similar functionality – these query statements need to contain a `returnValue[]` array that is filled during the course of execution. The contents of this data vector are then inserted into the output XML structure. Figure 4.12 shows how the framework's output is therefore composed.

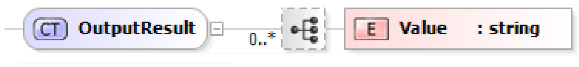

Figure 4.12: XML schema definition of returned output XML data

4.6 Data Interface Controller - Solvers

Once the controller has got hold of all data that is needed for solving a particular problem, it assembles the model's data structure and forwards it to the respective solver. While for certain problem types this may be minimal, other descriptive data collections may be somewhat voluminous in comparison. Deterministic algorithm code for analytically solvable problems such as for determination of the quench condition of chapter 5.2 would, for example, range on the facile end of said spectrum. Extensive geometries of problems demanding FEM simulation may as well produce wide ranging data structures, especially when complex boundary conditions, sources, and material properties have to be considered.

Common to both examples and all varieties in between is the fact of being representable by XML. Data structures following such demands can be defined within an *XML schema definition (XSD) file*, which for the sake of legibility can comfortably be depicted as a tree structure as in 4.13.

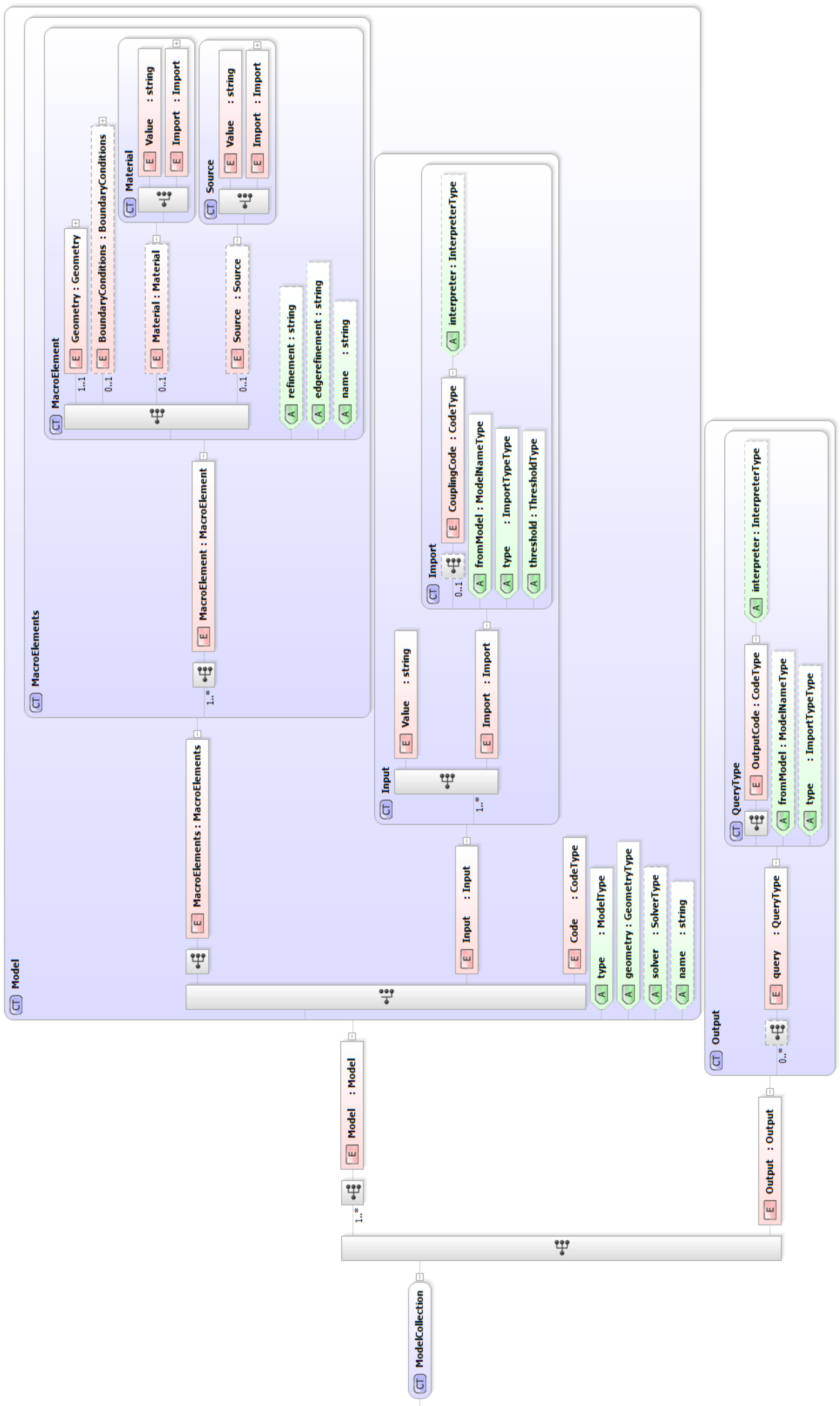

Figure 4.13: XML schema definition of the full modelling space

The root element bears the name *ModelCollection*, its children are one or more *models* of problems to be solved. These models have to be distinguished by an unambiguous name, if more than one is present. This identifier is necessary for data referentiation during the coupling process later on.

A model may hold data for different simulation options, such as FEM or analytics. The problem's nature is stated implicitly through the *solver* - attribute. *FEMtastic* would, for example, launch the finite element solver discussed in chapter 4.2.1, while *Elfe* or *EleFAnT* initiate the respective FEM solvers of [56] and [24].

Python or *Matlab* would also qualify and denote the intention to analytically solve a simulation assignment using the respective code base. This option offers an implicit data interface to and from the supported analytic computation enviroments. Both programme code input data – as payload of the *Code* - statement – and input data using the *Input* - directive can be transported as part of such a model type. The input data may well originate from another model's computation and be fed into this domain by using an *import* statement.

A FEM model, on the other hand, is composed of multiple *MacroElements*, in a cardinality of 'at least one'. This container is needed to hold the respective number of *MacroElement* statements. A particular problem is set up of at least one *MacroElements* container; typically, it is only one. The option for more of these containers has been added to unite *MacroElement* entries sharing common properties in the future.

Subsequently, a *MacroElement* is set up of four basic properties:

- A polygon defines the *geometry* of the macroelement. Figure 4.18 gives details about the according XML scheme.
- The *boundary conditions* set up properties on the border of the macroelement, thus affecting one or more edges, respectively, as in figure 4.14.
- Depending on the type of problem the *material* may define
  - the permittivity ϵ of dielectric material in electrostatic problem setups
  - the electrical conductivity σ of conducting material in static current flow problem setups
  - the permeability μ of magnetic material in magnetostatic problem setups
  - the thermal conductivity k for thermal problem setups
- The *source* is used to define volume charge densities in electrostatic problems, or a thermal source.

The definition of the geometry property is discussed within the next chapter, material, sources and boundary condition definitions are depicted by means of a real-world problem in chapter 5.1.1.

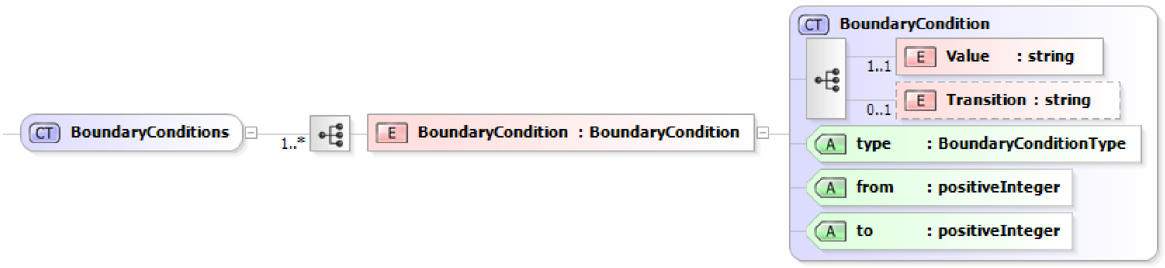

Figure 4.14: XML schema definition of a macro element's boundary conditions

4.6.1 Geometry

The toolset under scrutiny offers functionality to model any two-dimensional geometry using polygon set operations, employing a robust polygon combination algorithm [44].

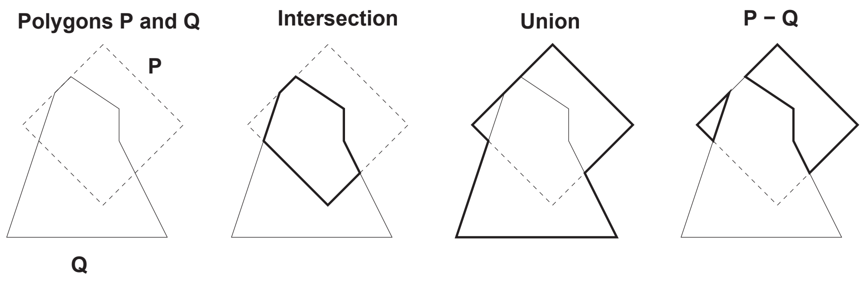

Figure 4.15: Boolean operations on polygons

This algorithm applies a boolean operation to two polygons, the *subject* polygon P and the *clipping* polygon Q. These operations are, as depicted in figure 4.15

- Union,
- Difference and
- Intersection.

The algorithm's major characteristics can be subsumed as:

- Fast computation in the order of $\mathcal{O}((n+k)\log(n))$, with n being the accumulated number of edges of both polygons and k the total amount of intersections between these edges. This makes it advantageous for usage within the scope of a graphical user interface as well, where fast response time are imperative.
- The algorithm is not limited to a certain type of polygons. Concave or self-intersecting polygons are supported.
- Polygons with ”holes” are possible.

- The output is not degenerated. Degeneration means within that respect that the output is only an approximation of the real result due to limitations implied by the algorithm itself. Figure 4.16 depicts the correct result of a boolean polygon operation on the right-hand side, while the result on the left is considered degenerated. This is due to the two areas of the real result being connected by small bands.

- Regions composed of non-intersecting polygon sets are supported and consolidated into single polygons. The result of the difference operation in figure 4.15, for example, yields two separate regions. These are, however, treated as one polygon data structure. Such a combination of polygons may also be used as an input subject or clipping polygon.

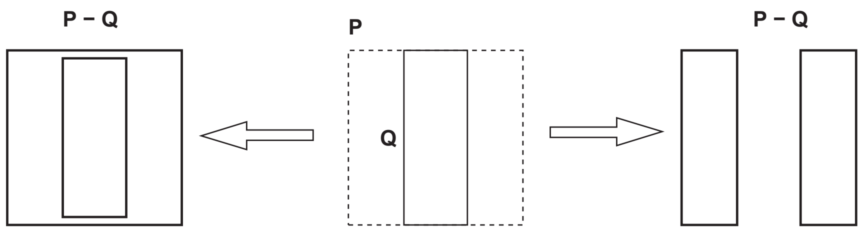

Figure 4.16: Polygon degeneration through difference calculation

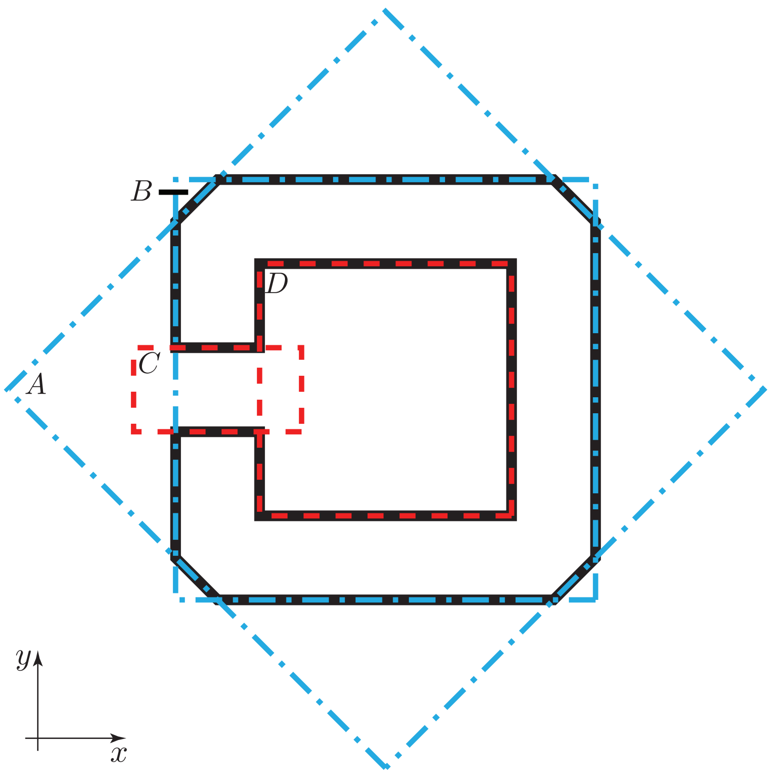

Figure 4.17: Iron core with air gap

The Martinez algorithm therefore provides adequate functionality in order to build up complex geometry topographies by defining simple polygon primitives and applying boolean operations to them. As an example, the proper combination of four rectangular polygons A, B, C and D leads to the exemplary iron core with air gap as depicted in figure 4.17. It is then the result of applying the rule $((A \cap B) \setminus C) \setminus D$, or in other words: Intersection of A and B minus C minus D. The appropriate XML statement would be

Listing 4.1: Geometry definition of an iron core

```
<Geometry>
  <Subtract>
    <Intersect>
      <Polygon> .. rectangle A .. </Polygon>
      <Polygon> .. rectangle B .. </Polygon>
    </Intersect>
    <Polygon> .. rectangle C .. </Polygon>
    <Polygon> .. rectangle D .. </Polygon>
  </Subtract>
</Geometry>
```

The geometry is set up by combining *<Polygon>* primitives. The resulting geometry is also a polygon. Possible combinations of polygons are the boolean set operations **Union** (*<Add>*), **Difference** (*<Subtract>*) and **Intersection** (*<Intersect>*). For full flexibility, the boolean operations can be nested. The only restriction is composed by the fact that the *<Geometry>* entry must have at least one child of the following types: *<Add>*, *<Subtract>*, *<Intersect>* or *<Polygon>*.

The boolean operations are evaluated from top to bottom. Therefore,

Listing 4.2: Polygon set evaluation from top to bottom

```
<Subtract>
  <Polygon> .. A .. </Polygon>
  <Polygon> .. B .. </Polygon>
  <Polygon> .. C .. </Polygon>
</Subtract>
```

evaluates to $(A \setminus B) \setminus C$. This resembles A minus B minus C. A more complex example would be

Listing 4.3: Example of a nested geometry definition

```
<Subtract>
  <Polygon> .. A .. </Polygon>
  <Add>
    <Intersect>
      <Polygon> .. D .. </Polygon>
      <Polygon> .. E .. </Polygon>
      <Polygon> .. F .. </Polygon>
    </Intersect>
    <Polygon> .. G .. </Polygon>
    <Polygon> .. H .. </Polygon>
  </Add>
  <Polygon> .. B .. </Polygon>
  <Polygon> .. C .. </Polygon>
</Subtract>
```

This satisfies

$$\underbrace{A \setminus \left\{ \left[\underbrace{\Big(\underbrace{(D \cap E \cap F)}_{\text{intersection}} \cup G \cup H \Big)}_{\text{union}} \setminus B \right] \setminus C \right\}}_{\text{subtraction}}$$

Polygons within XML

The *Polygon* directive is a set of two-dimensional *Coordinates* that define the nodes of such a polygon. By definition, polygons must have at least three nodes. The last node is defined to be automatically connected to the first one. The number of nodes per polygon is unlimited, and their succession defines the numbering of the resulting macro element's edges. This is a necessary characteristic for edge identification, as used for defining boundary conditions.

Figure 4.18 shows the XML schema definition for the <*Geometry*> entry.

4.6.2 Accuracy

With triangles being a very popular drawing primitive, there is a broad number of graphics libraries and hardware subsystems that make triangular meshes very common in computer aided modelling or drawing (CAM or CAD, respectively). Complex models reproducing substantial real-world domains can easily contain millions of faces, and current CAD/CAM tools handle these without any problem [12].

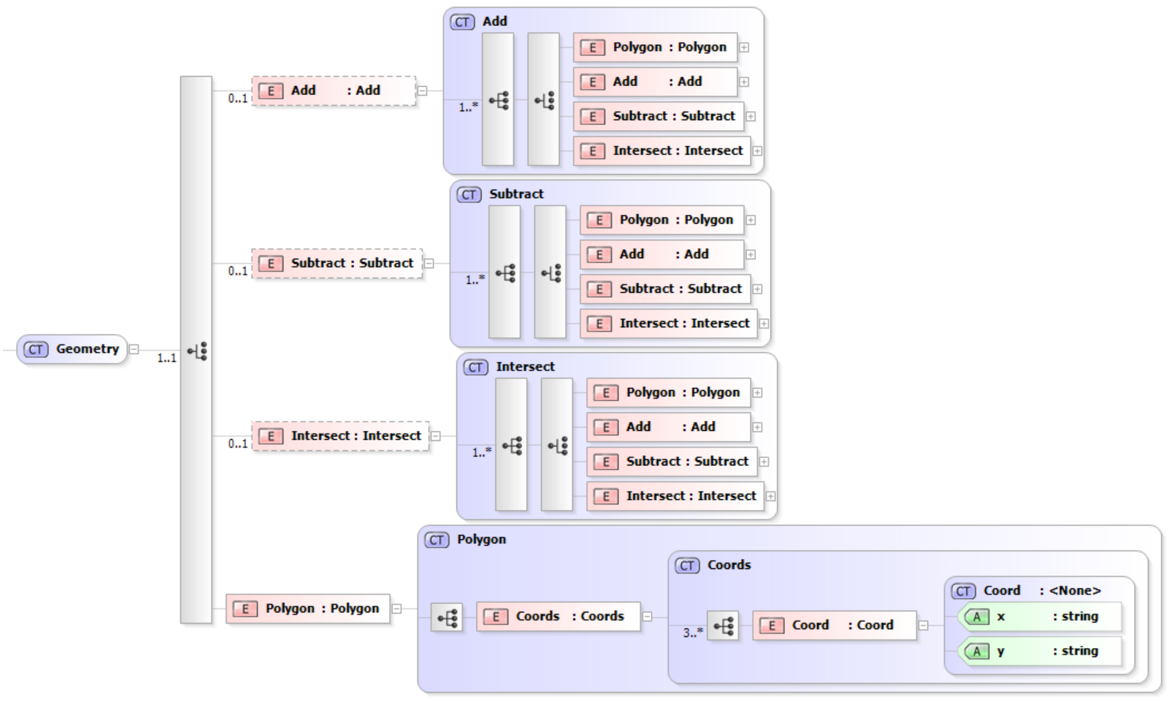

Figure 4.18: XML schema definition of a macro element's geometry

Unfortunately, a major drawback arises when such problem space discretizations not only have to be depicted, but also form the basis for numerical calculations, as an increase in data complexity may easily reach hardware performance boundaries. The algorithms used for finite element calculations within numerical simulation problems usually are of quadratic complexity, $\mathcal{O}(n^2)$, which means the consumed processing time increases in the order of the second power of the number of finite elements. Chapter 6.2.1 shows the monitoring of real-world parameters to substantiate this claim.

However, a highly complex data representation is not always required, as for many problems to be fed into a simulation tool chain only small parts of the model space are of particular interest. These parts are typically characterized by strong gradient changes of the underlying physical quantity that is to be determined. Coarse meshes in these areas would lead to inaccurate solutions, which is why often a tradeoff between overall accuracy and processing time as contemplated above has to be found.

This tradeoff can be overcome when only the mentioned regions of interest are equipped with some mesh of finer resolution. The XML problem representation offers two directives for that purpose, *refinement* and *edgerefinement*, both showing their own peculiarities. They are available in the form of attributes assigned to each *MacroElement*, where necessary (see figure 4.13 or listing 5.9).

Therefore, both are of an optional nature, are accepted as real numbers, and can be seen as to lengthen or shorten an arbitrary base length by multiplication. 1.0 is the neutral factor, it is assumed to be in force when no refinement factor is set. Accordingly, numbers larger than 1.0 yield coarser meshes, smaller ones result in finer ones.

- *refinement* refactors all of the mesh of an actual macro element. This should be used with caution when modelling the far problem boundary, as finer meshes would result in enormous numbers of finite elements and respective computation times, all out of proportion to the increase in accuracy. Therefore, regions of interest could, for example, be furnished with a small macro element, that bears the same physical quantities as the background one, but offers finer mesh refinement.
- *edgerefinement*, on the contrary, only works on the boundaries of a macro element and therefore creates a gradient of mesh resolution leading away from these boundaries. This comes in very handy when major differences of the solution's quantity are to be expected along or in the near vicinity of a macro element's edges.

Figures 4.19 and 4.20 show the difference between a problem discretization of no refinement at all and a setup with a refinement factor of 0.1. Figure 4.21, on the contrary, depicts a model setup where an edge refinement factor of 0.1 was chosen for the far boundary and for the outer (right) coil, respectively, leaving both the air space and the inner (left) coil unrefined.

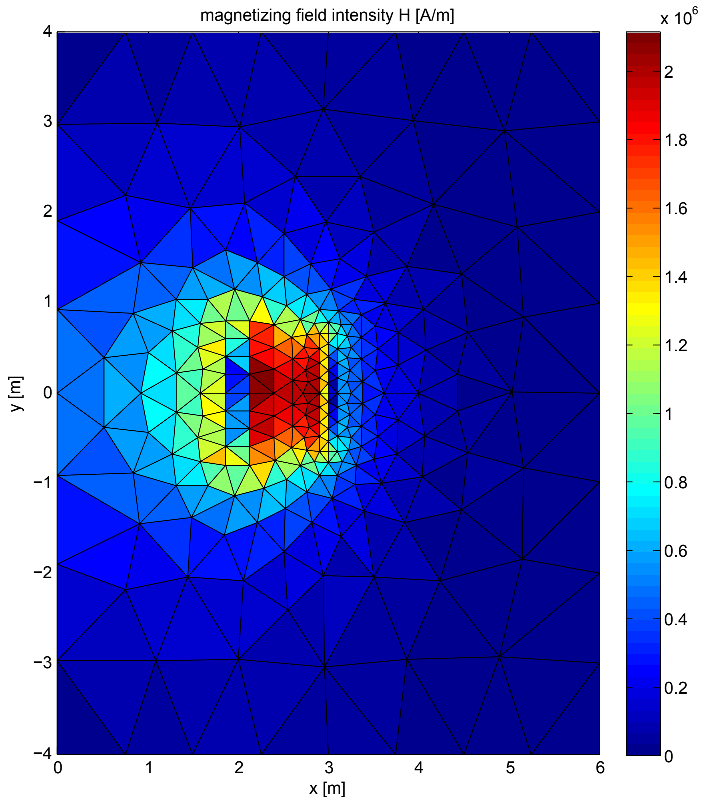

Figure 4.19: Unrefined standard meshing of the TEAM22 benchmark problem

Having the mesh refinement parameters available as real numbers, it is possible to vary the refinement process either from outside as part of the optimization process, or by instructing the controller to adapt the mesh coarseness on its own discretion. Chapter 4.1 shows the approach the controller would follow in order to exhibit this functionality.

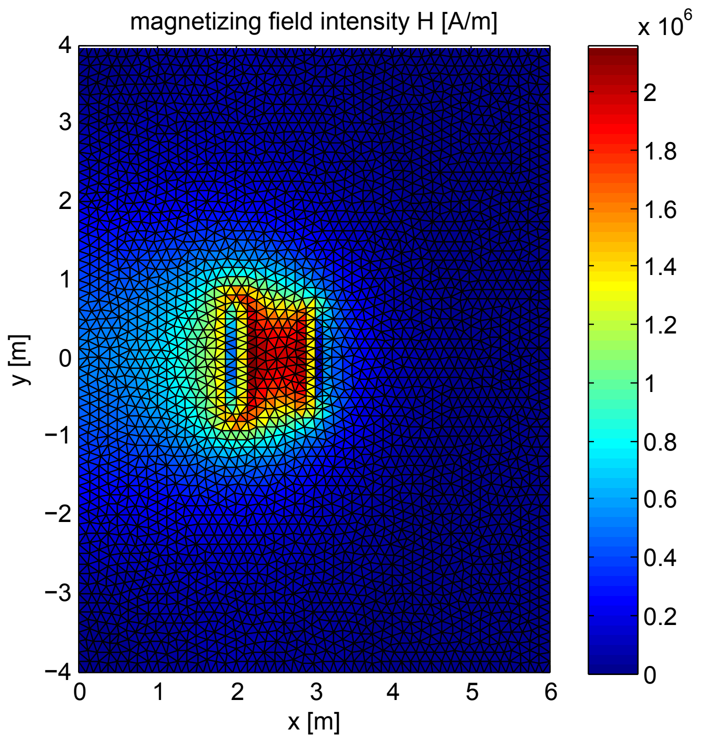

Figure 4.20: Adapted area meshing of the TEAM22 benchmark problem

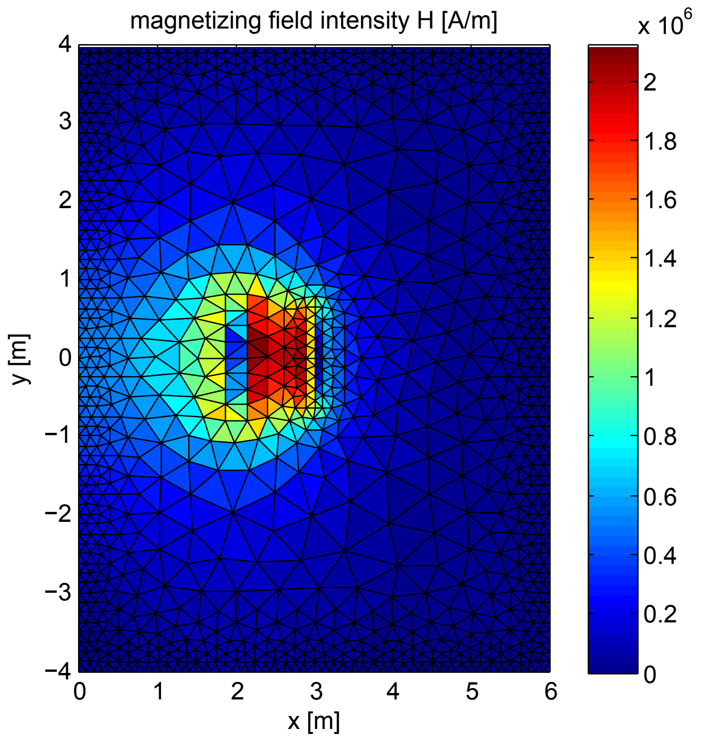

Figure 4.21: Adapted boundary meshing of the TEAM22 benchmark problem

4.7 Data Interface Solvers - Post-Processors

As depicted in figure 3.1, there is a data path between a model's solver output and another model's input. Between them, usually, some postprocessing structure is located. The framework under scrutiny offers two ways for deriving data from a problem's solution set.

4.7.1 Multiple Models & the `Import` - Directive

The import of values from one model to another is the process that is pursued with respect to weakly coupling them. Therefore, it plays an eminent role within the whole simulation workflow. Importing values is currently allowed for models that are to be solved analytically, and for certain characteristics of FEM models. Figure 4.13 depicts permissible applications within the main XML schema; the values to be applied as sources or material parameters would qualify.

The *Import* - directive within the model offers two ways of deriving values that are to be forwarded to the next model's setup. Figure 4.22 shows the according data schema diagram.

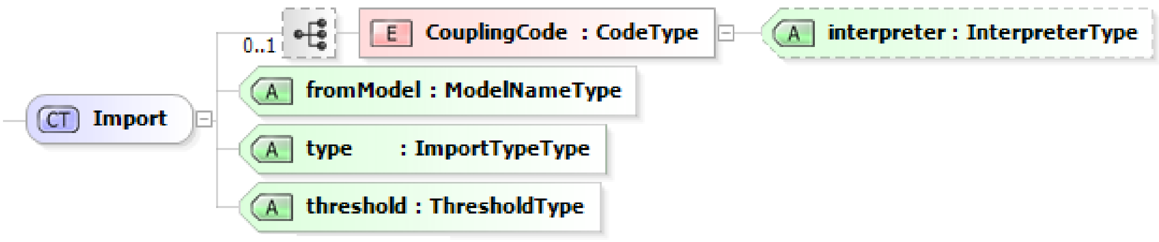
0..1
E CouplingCode : CodeType
A interpreter : InterpreterType
CT Import
A fromModel : ModelNameType
A type : ImportTypeType
A threshold : ThresholdType

Figure 4.22: XML schema for importing

On the one hand, a number of readily available functions for common coupling problems exists. For these, the *type* - attribute of the *Import* element may be employed. Currently, the following directives are available:

- `max` (ν) gathers the maximal value of a quantity ν in the domain of the current macro element (where applicable). ν depends on the nature of the problem to be solved. Apart from that, also certain derivatives from the addressed result set may be queried. For electrostatic and current flow models, for example, ν may take the following values:
  - `U` delivers the maximal value of the electric potential Φ (denoted as `U`, to simplify matters)
  - `E` delivers the maximal value of the electric field strength depending on the potential distribution within the given domain
  - `J` delivers the maximal value of the current density, depending on electric field and material properties within the given domain

- `source` retrieves integral qantities that serve as a source with respect to the given simulation model's nature. For magnetostatic problems, this would be the total current I within a given domain.
- `energy` yields the total energy stored in a magnetic or electric field within a particular domain.

Then again, more complex dependencies have to be modelled using very specific functions that are not easily mapped using said features. For example, the temperature dependency of a metal's specific conductivity follows a curve that is best implemented using formulae as discussed in chapter 4.8.1. Chapter 5.1.3 describes an example for such a coupling scheme. In such cases, special code of any necessary complexity may be implemented using *CouplingCode*. Its attribute *interpreter* denotes which code interpretation base should be deployed, e.g. Matlab directly, or Python via the interface described in chapter 4.2.2.

Common to both interface varieties is the necessity to determine where to take information from. *fromModel* denotes exactly that linkage, and refers to a model's name attribute. In that respect, the weak coupling scheme pursued within this work is of information pulling nature; a single model and its solving architecture does not have to be aware of being coupled with others, it is only queried for certain results when necessary.

When two or more models are not coupled in one direction, but information is also gathered from the last model in the chain to feed the first one again, the resulting workflow may easily end in infinite round trips. This may be desirable when particular results have to converge on account of changing external conditions determined by another simulation.

Nevertheless, such a process needs to be brought to a stop once stability is reached. This is usually done by comparing results of two succeeding simulation runs for their difference, and halt the round-trip accordingly as soon as a certain *threshold* is under-run. Both absolute and relative values are accepted for the respective attribute, with 1% being the default value that is employed automatically in order to avoid infinite loops.

4.8 Data Interface Evaluator - Solvers

4.8.1 Postprocessing Functionality

Typical numerical simulation frameworks suffer from a major drawback concerning the definition of postprocessing functionality. While standard functionality such as drawing of geometry, contour or field plots and computing a set of predefined quantities can be expected from virtually any FEM software, adding new instructions to evaluate initially disregarded quantities may as well turn out to be impossible.

The main functionality of some FEM problem solver's postprocessing functionality typically is to calculate, visualize or integrate quantities which are not a direct result of the solving procedure. These can be split into two categories [47]:

- *Local quantities* depend on a single point within the modelled domain. Current flow density or the magnetic flux density would, for example, be characterized by such quantities.
- *Global quantities* are derived by evaluating an integral along a surface or volume (or it's two-dimensional simplification) of a local quantity. Total current, tension, force or flux can be counted within this category.

Quantities of both categories can be expressed using an arithmetic formula. The advance of efficient interpreted programming languages almost inflicts creating an interface to express these formulae directly and interactively for postprocessing instead of predefining new subroutines for each quantity to evaluate. Therefore, both MATLAB and Python have been integrated for supplying postprocessing features.

The mentioned formulae, of course, need to be granted access to some basic values such as problem topology and geometry or its physical properties [38]. Above that, the introduction of parameters or placeholders is worth being considered. Using these, a formula can receive its input arguments as descendants of other formulae, thus even introducing the convenient concept of recursivity, by the way.

The most basic value that is to be obtained by postprocessing is also the most crucial one. For almost any further calculation, of course the problem solution is of interest in each point of the problem geometry. This is straightforward for the mesh nodes, especially when the solver has returned a full solution set to the framework that has been stored as objects (or their persistent HDF representation). The approximation of values at points within the area covered by a finite element may be somewhat trickier when the software module does not allow for access to according functionality. In such cases, it is in the user's discretion to resort to either the solver's wrapper or the framework, both of which offer reasonable functionality for retrieving the field value for a given computation point solely based on its coordinates.

4.8.2 Data Structures and Commands

In order to close the feedback loop back to the superordinate optimization structure signified in figure 3.1, structured output has to be produced by the evaluator. This is done by following the instructions handed over to the controller with the input XML data. The controller then splits this input into models to be simulated, and the part concerning output generation. Its structure is depicted in figure 4.23.

Output generation is based on the concept of the so-called *query*, meaning that the full space of intermediate (stored) and final solutions have to be interrogated or queried for values to be output. The permissible number of queries is unlimited for each simulation set.

Very much following the process for importing data from one physic's model into another, and therefore allowing for the above mentioned formula concept, each query requires *OutputCode* to be passed. The programming code contained in the directive's payload has to

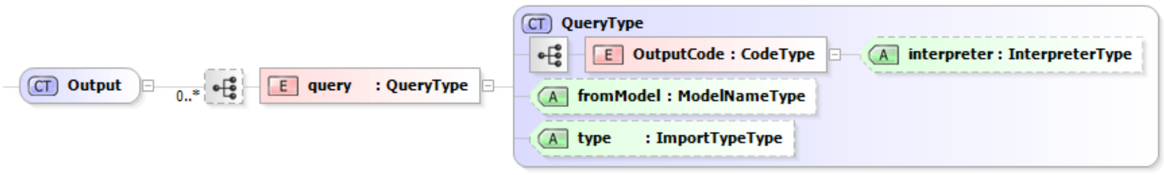

Figure 4.23: XML schema for output generation

be qualified by *interpreter* in order to hand it on to the appropriate processing functionality; again, a choice between Matlab and Python is available.

Using the *importFrom* attribute, the model to obtain data from has to be identified. Other than in the case of coupling two models, in this case also historical data may be accessed. This is fulfilled by appending the historical cycle that is to be evaluated to the model identifier. Numbering is carried out in reverse chronologic order, `-1` therefore denoting the last run before a stopping criterion has been met and the solutions have therefore been marked as definite.

The output is then composed using the functionality the evaluator provides for that purpose as discussed in chapter 4.5. With returning the full range of retrieved data the service the simulation framework offers is completed.

Chapter 5

Applications

"We have no idea about the 'real' nature of things ... The function of modeling is to arrive at descriptions which are useful."

[Richard Bandler and John Grinder]

During the course of research on the topics covered by this thesis, a number of real-world optimization problems have been investigated. This chapter summarizes the activities several publications originated from. The respective references can be found in the corresponding chapters.

5.1 Electrothermal Hardening

5.1.1 Problem Setup

Hardening of steel components is commonly done using the method of annealing. This typically requires controlled heating of the workpieces to exact temperatures following a just as exact time schedule in order to effectuate the desirable material properties.

Therefore, the design of an appropriate heating device would include the determination of the dimensions a heating plate should have, given the fact that the device is supplied with a fixed voltage. The optimization strategy and the determination of fitness with respect to the optimization objectives is left to an optimization toolbox not within the scope of this thesis. Nevertheless, as the XML file to be filled with data is the input interface to the outside world, it is of quite some interest to show how an actual input file would be made up. As mentioned in chapter 4.6, XML equally allows man and machine to read and write within its structures, and hence it is a good starting point for each optimization task to start with a man-made model and proceed in altering it automatically.

The two-dimensional cross-section model of such an appliance made from alloy is shown in Fig. 5.1 together with its cupreous feed connections shown on either side of the heating plate. An arbitrary workpiece of steel is placed on top of it. All measurements are given in meters, the third dimension is assumed to be infinite in length, which eases the model to being of two-dimensional nature only.

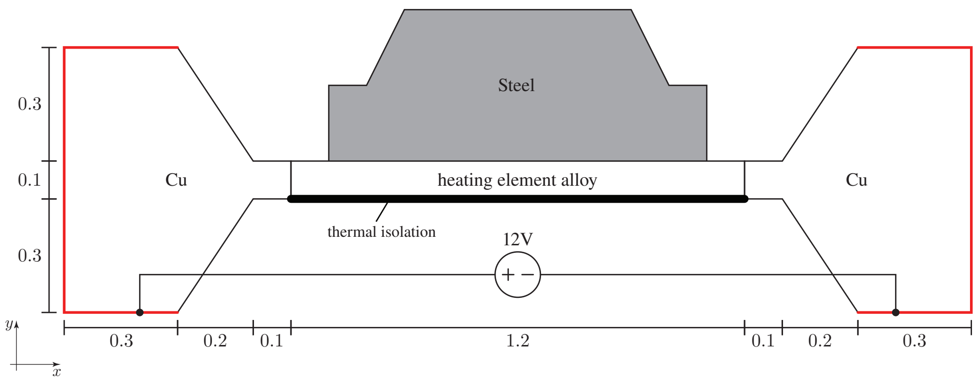

Figure 5.1: Heating device - problem geometry

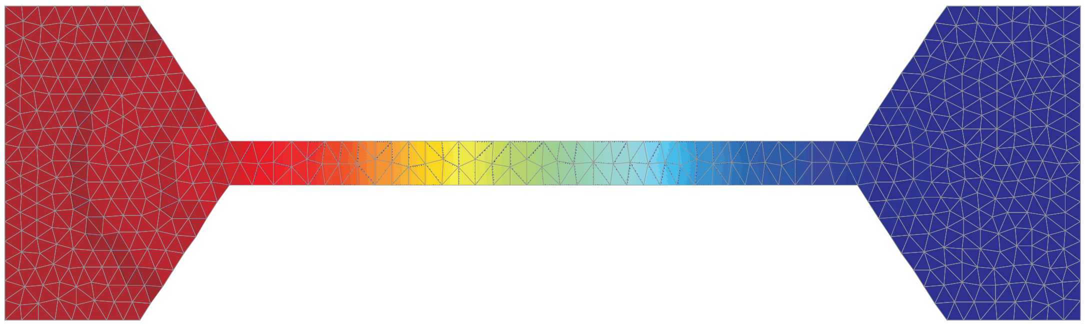

Figure 5.2: Electrical potential distribution

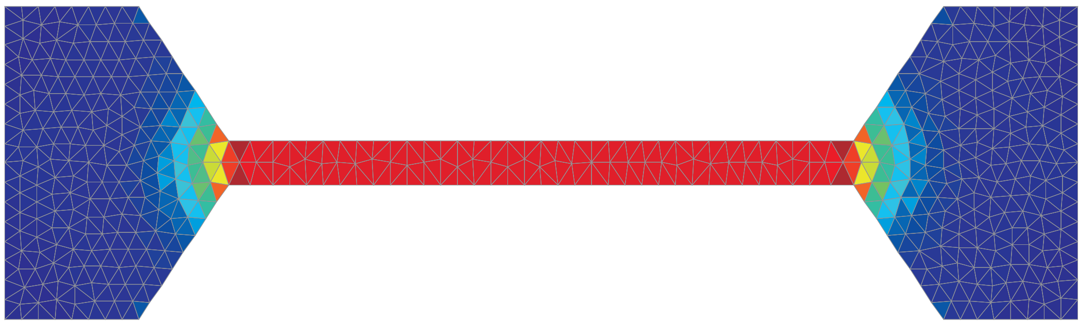

Figure 5.3: Current density causing the losses that serve as thermal source

The electric potential distribution that arises as shown in figure 5.2 drives current through the heating plate (as depicted in figure 5.3). As a result of the power dissipation due to ohmic losses therein, it will heat up and thus serve as a source for the thermal model simulation. Within this model, the feed connections can be neglected, but as, of course, the workpiece to be hardened has to be taken into account, a problem space discretization entirely different to the current flow model's is the result. Figures 5.4 and 5.5 show the according model space and its discretization as well as the resulting simulated thermal distribution.

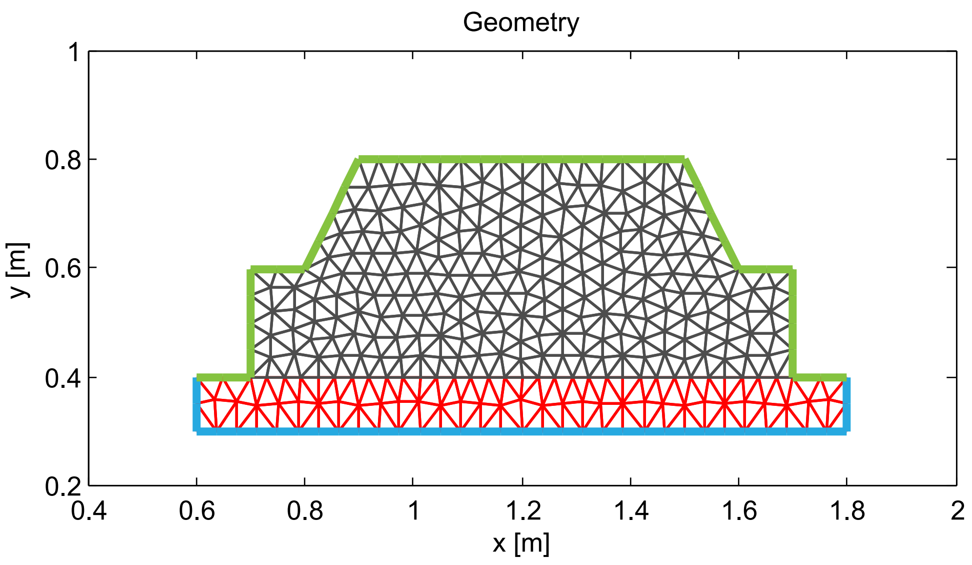

Figure 5.4: Thermal problem setup and triangulation

Now, taking advantage of the finite elements being realized as software objects, the input values for source elements of the thermal model can easily be determined for each finite element's integration points. As each FE object of the current flow problem offers its own postprocessing functionality, exact input values - the power dissipation in each integration point as a result of the current density $\vec{J}$ - can be retrieved and fed into the new thermal source elements.

Also, the temperature that arises within the heating plate can influence its material properties, e.g. specific conductivity. Therefore it may be advisable to re-feed this information into the electric model and simulate it again, before starting the thermal simulation using the now freshly calculated current flow. This round-trip can be repeated until the difference between two iterations falls below a certain threshold.

[30] holds details to early stages of reasearch on this topic, while [29] summarizes and publishes the findings expounded within the next chapters.

5.1.2 Model Creation

In order to create a fully functional model within the optimization framework under scrutiny, a few details of the data structure holding said model shall be explained. Above mentioned

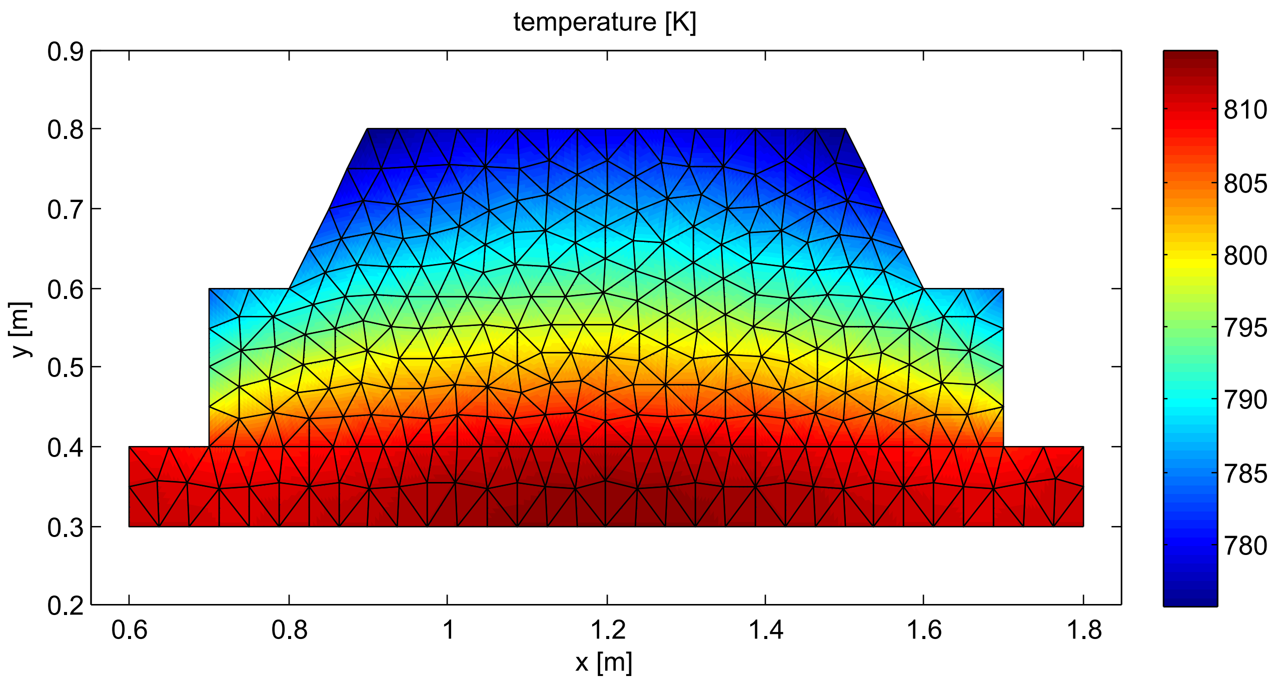

Figure 5.5: Thermal distribution within the modelled domain

problem is perfectly apt for dividing it into two partitions, each of which can be solved using FEM principles. Hence, the model is explained in a bottom-up manner, until the whole structure is filled and can be forwarded to the solving framework. The full XML structure is given in listing A.2.

5.1.2.1 Geometry

The relatively simple model geometry allows for the use of polygons only, sophisticated union or intersection operations do not have to be conducted. This is equally true for all 4 macro elements the geometry consists of. Fig. 5.1 already foreshadows a meaningful composition of these elements - both feeders and the heating plate for the current flow partition, the heating plate and the workpiece for the thermal distribution model.

For each of these macro elements, a polygonic layout as given in listing 5.1 is sufficient to fit into the XML scheme developed and described in chapters 4.2.1 and 4.6. Special attention has to be paid that even if no operation on polygon sets is necessary, at least one polygon has to be 'added' to the geometry structure.

Listing 5.1: Left feeder geometry definition

```
<Geometry>
  <Add>
    <Polygon>
      <Coords>
        <Coord x="0" y="0.7" />
        <Coord x="0" y="0" />
        <Coord x="0.3" y="0" />
        <Coord x="0.5" y="0.3" />
        <Coord x="0.6" y="0.3" />
        <Coord x="0.6" y="0.4" />
        <Coord x="0.5" y="0.4" />
        <Coord x="0.3" y="0.7" />
      </Coords>
    </Polygon>
  </Add>
</Geometry>
```

The result of such a geometric layout would be the left feeder's macro element, also depicted in figure 5.6. All other macro elements have to be set up similarly; the full listing in A.2 shows these as well.

5.1.2.2 Boundary Conditions

Applications of the Finite Element Method, in general, make use of *Dirichlet*, *Neumann* and *Cauchy* boundary conditions. How these are utilized for the hardening device example shall be delineated subsequently.

Listing 5.2: Defining Dirichlet and Neumann boundary conditions

```
<BoundaryConditions>
  <BoundaryCondition type="Dirichlet" from="8" to="2">
    <Value>
      12
    </Value>
  </BoundaryCondition>
  <BoundaryCondition type="Neumann" from="5" to="5">
    <Value>
      0
    </Value>
  </BoundaryCondition>
</BoundaryConditions>
```

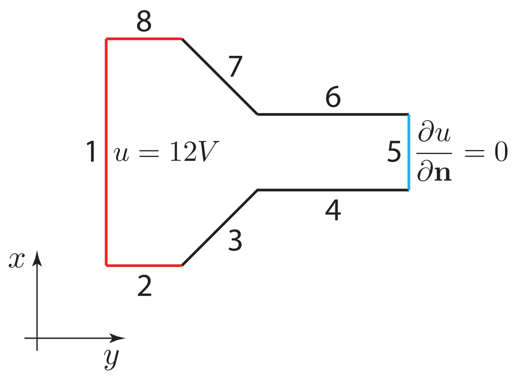

Figure 5.6: Edges and boundary conditions of the left feeder

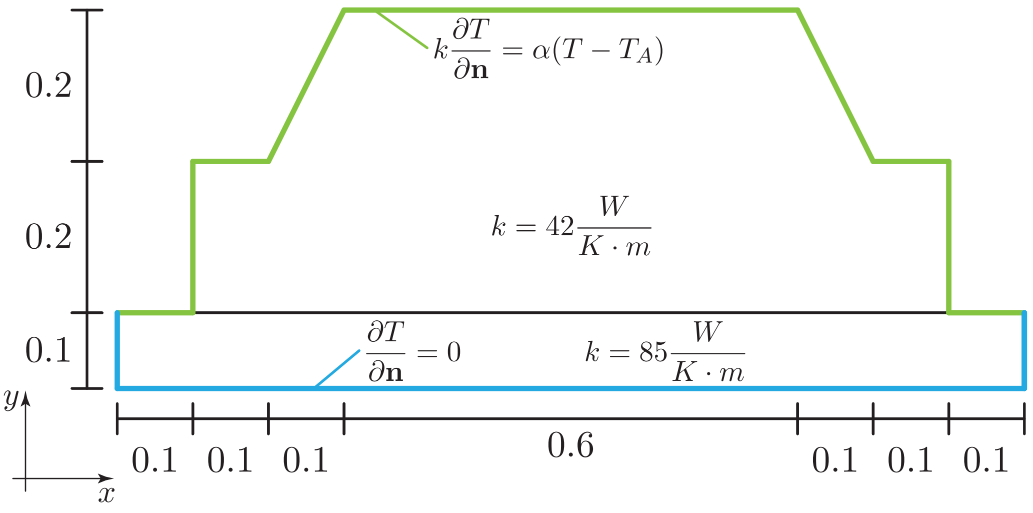
$k\frac{\partial T}{\partial \mathbf{n}} = \alpha(T - T_A)$
$k = 42\frac{W}{K \cdot m}$
$\frac{\partial T}{\partial \mathbf{n}} = 0$
$k = 85\frac{W}{K \cdot m}$
0.2
0.2
0.1
y
x
0.1
0.1
0.1
0.6
0.1
0.1
0.1

Figure 5.7: Material and boundary conditions within the thermal model

Listing 5.2 defines the boundary conditions for the heating device shown in figure 5.6. As, by design, the edges of the macroelement are numbered counterclockwise, edges 1, 2 and 8 are to be seen as being the electrodes of the device. Their electric potential is defined with $12V$ by means of defining Dirichlet boundary conditions for each of them.

A range of edges can be defined using the attributes *from* and *to*. A polygonal shape is defined as being enclosed by edges, it is both allowed and useful to have a *to*-value that is larger than the *from*-value. In such a case, all edges with a numbering equal or higher than *from* or lower than or equal to *to* are taken into account. The Dirichlet boundary condition definition in listing 5.2 for the feeder figure 5.6 shows exactly this situation.

Both attributes for the edges' range, *from* and *to*, have to be set as well in case there is only one edge to be allocated with certain boundary conditions. The Neumann boundary condition set for edge 5 would be such a case, and represents a symmetry surface.

For thermal problems, Cauchy boundary conditions are of great importance. Using these, heat transfer can be modelled quite easily. Listing 5.3 shows the definition for the steel workpiece to be hardened as given in the current problem. As depicted in figure 5.7, its seven edges that are in direct contact with surrounding air of temperature T_A emit heat. This emission is expressed through an additional value entry, *<Transition>*. Using this, the transition coefficient α of a Cauchy boundary condition is defined. The transition coefficient for the workpiece is to be assumed as $\alpha = 5.6 W K^{-1} m^{-2}$ at an ambient environment temperature of $T_A = 300K$.

<Transition> is only expected when the boundary condition's *type* property is set to *"Cauchy"*, in all other cases it will simply be ignored.

The bottom of the heating plate is thermally insulated and also the connections to the electrodes are assumed to have no temperature flow. This is mathematically expressed by Neumann boundary conditions as described above.

Listing 5.3: Defining a Cauchy boundary condition

```
<BoundaryConditions>
  <BoundaryCondition type="Cauchy" to="7" from="1">
    <Value>300</Value>
    <Transition>5.6</Transition>
  </BoundaryCondition>
</BoundaryConditions>
```

5.1.2.3 Material

By definition, a macro element is a region within a model setup that consists of material with equal properties over exactly this region. For the problem under scrutiny, two material properties are of special interest - the feeders' and heating plate's electrical conductivity, and the heating plate's and workpiece's thermal conductivity respectively. For each model (electrical and thermal) and the macro elements assigned, the material properties have to be entered. Listing 5.4 shows how this is done for the feeder macro elements which should resemble copper. As the feeders are of interest while computing the static current flow problem setup, the electrical conductivity of copper, $\sigma = 59.1 \cdot 10^6 S/m$ is given.

Listing 5.4: Material definition of a macro element representing copper

```
<Material>
  <Value>
     59.1E6
  </Value>
</Material>
```

Table 5.1 summarizes the material properties used in the electrothermal hardening device setup corresponding to the model type.

problem type	macro element	interpretation	value	unit
static current flow	left feeder	electrical conductivity	$\sigma = 59.1 \cdot 10^6$	S/m
static current flow	right feeder	electrical conductivity	$\sigma = 59.1 \cdot 10^6$	S/m
static current flow	heating plate	electrical conductivity	$\sigma = 1.4 \cdot 10^6$	S/m
thermal	heating plate	thermal conductivity	$k = 85$	$W \cdot K^{-1} \cdot m^{-1}$
thermal	workpiece	thermal conductivity	$k = 42$	$W \cdot K^{-1} \cdot m^{-1}$

Table 5.1: Material properties and their interpretation

5.1.2.4 Source

In electrostatic or thermal problem setups, a macro element can also represent a source of certain nature. Therefore, a facultative fourth macro element property has to be introduced. In case a *<Source>* entry exists and differs from 0, the macroelement represents a volume charge density for electrostatic problems or a thermal source. The value defined in such a manner bears a unit of $\frac{C}{m^3}$ or $\frac{W}{m^3}$, respectively. Listing 5.5 would show such a source's definition.

Listing 5.5: Source definition

```
<Source>
  <Value>
    5.9
  </Value>
</Source>
```

Ensuring weak coupling between two models via a way of exchanging data, the statement *<Import>* is of great help. The exact syntax for using the data import from another model as a source is given in listing 5.6. The value contained in the *<FromModel>* expression denotes the model that shall serve as the source in this context, while *<Type>* and *<Threshold>* supply the controller with additional information on how the import procedure should be handled. More information about the actual model coupling for the hardening device is to be found in chapter 5.1.3.

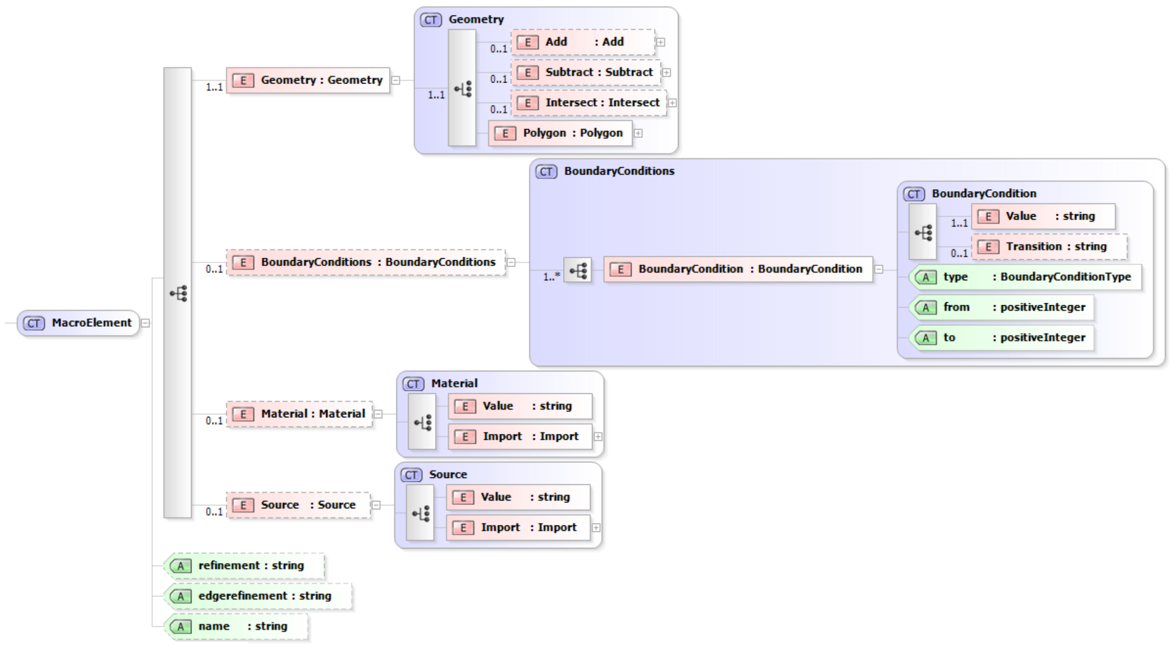

Figure 5.8: XML schema definition of a MacroElement

Listing 5.6: Import of another model's data

```
<Source>
  <Import>
    <FromModel>1</FromModel>
    <Type>current-thermal</Type>
    <Threshold>1%</Threshold>
  </Import>
</Source>
```

With all four properties of macro elements being defined by now, the XML schema definition for the <*MacroElement*> statement can be developed. Figure 5.8 summarizes such a definition.

5.1.3 Multiphysical Coupling of Current Flow and Thermal Models

The computation of a full optimization cycle for the given problem setup requires, as outlined in chapter 5.1.1, an intermediate result to be computed. In the case of the first model – the static current flow model – that would be the distribution of the current density $\mathbf{J}$.
In order to gain the final result, the thermal model has to be computed subsequently in order to receive the temperature distribution or the scalar temperature field T of the steel workpiece.

For the reason of having two coupled models originating from different physical domains this approach is called *multiphysics*. After computing the current flow within the heating plate, the respective power losses can be calculated. In the course of calculating the second thermal model, these power losses will act as energy sources for heat.

Figures 5.4 and 5.7 define the thermal subtask. Only the heating plate and the steel workpiece have to be modelled; thermal effects within the electrodes can safely be neglected in just the same manner as the influences on the current flow in the workpiece are disregarded. The heating plate is then constructed of an alloy with a thermal conductivity of $k = 85WK^{-1}m^{-1}$, the steel workpiece on the other hand shows a thermal conductivity of $k = 42WK^{-1}m^{-1}$.

The framework supports weakly coupled problems through the <*Import*> expression in the input XML file. It advises the controller to access the model that is identified by the therein contained number. It is then queried for its postprocessing features to return certain computed values. The controller, in the end, forwards these values to the actual model for assembling it into its calculations.

The current implementation necessitates parts analysed in both models to be located at the same positions in their respective geometry. For the given example this would mean that the heating plate features the same coordinates in both the current flow and the thermal model. Different setups of macroelements or a different meshing and triangulation would, on the contrary, be of no concern.

$$p = \frac{1}{\sigma}\mathbf{J}^T\mathbf{J} \tag{5.1}$$

Listing 5.6 shows how the import functionality for a given model is activated, in this actual case for a current flow - thermal coupling. Equation 5.1 states the relation between the current flow $\mathbf{J}$ and the power loss p in $\frac{W}{m^3}$, with the latter being the source of the thermal problem. This relation is specified through the value of the <*Type*> directive being set to *current-thermal*. The value for <*Threshold*> can be set to any arbitrary value, when no roundtrip coupling is applied.

In case the influence of the heating plate's temperature should also be fed back to its material properties, an import of the respective value for this would have to be put into execution before another calculation cycle is started as this may influence the current flow and the temperature distribution accordingly. In order to find a stopping criterion, either value being imported can be monitored by the controller. Should the difference of imported values between two cycles fall below the given threshold in the <*Import*> directive, the calculation cycle is halted. <*Threshold*> accepts both percentages and absolute values, and is – as mentioned above – ignored when the models to be calculated are not coupled in both ways.

The given setup was evaluated including such a feedback loop. That means that the temperature that arises within the heating plate has to be obtained and forwarded as input to calculate the heating plate's temperature dependent specific conductivity $\sigma_{(T)}$. Engineering reference compendia such as [67] list a temperature coefficient α for the electrical resistivity ρ. Therefore, the respective value of $\alpha = 0,4\%K^{-1}$ for the alloy material chosen for the heating plate has to be applied to obtain $\sigma_{(T)}$ as in (5.3).

$$\rho_{(T)} = \rho_{(T_0)} \cdot (1 + \alpha\Delta T) \tag{5.2}$$

$$\sigma_{(T)} = \sigma_{(T_0)} \cdot \frac{1}{1 + \alpha\Delta T} \tag{5.3}$$

Listing 5.7 implements exactly this relation with *temps* being an array for storing the sequence of temperatures that may arise during the simulation process. T_0 – and $\sigma_{(T_0)}$ along with it – has been fixed at 300K.

Listing 5.7: Determination of $\sigma_{(T)}$

```
sigmaT = sigma / (1+ alpha*(temps(numel(temps))-300));
```

The first simulation run per default uses T_0 for determination of $\sigma_{(T)}$. After that, the maximum temperature within the heating plate is obtained and stored in *temps*. As soon as the difference between the latest two values determined by this drops below 1%, the stopping criterion flag is set which brings the evaluation to an end.

Listing 5.8: Determination of simulation convergence

```
if abs(temps(numel(temps))/temps(numel(temps)-1)-1) < 0.01
   stop_criterion = 1;
end
```

For reasons of simplicity, α has been chosen to be a linear coefficient with respect to ρ, which in fact is a raw, yet sufficient approximation. As metals increase in specific resistivity with temperature, a temperature curve as shown in figure 5.9 arises. The temperature has also been chosen as the stopping criterion – once the recently obtained maximum temperature within the heating plate differs by less than 1% from last iteration's, the process is halted. 30 iterations proved to deliver the stability sought for. Both figure 5.9 and 5.11 show that either of the values determined for T, σ, p, or $\mathbf{J}$ would qualify for feeding the stopping criterion due to their similar convergence patterns.

Figure 5.10 depicts the position of a cross-sectional determination of the current flow density $\mathbf{J}$. It is calculated every 0.01 m along that line. The sum of 30 iterations' $s/\mathbf{J}$-diagrams is depicted in figure 5.11.

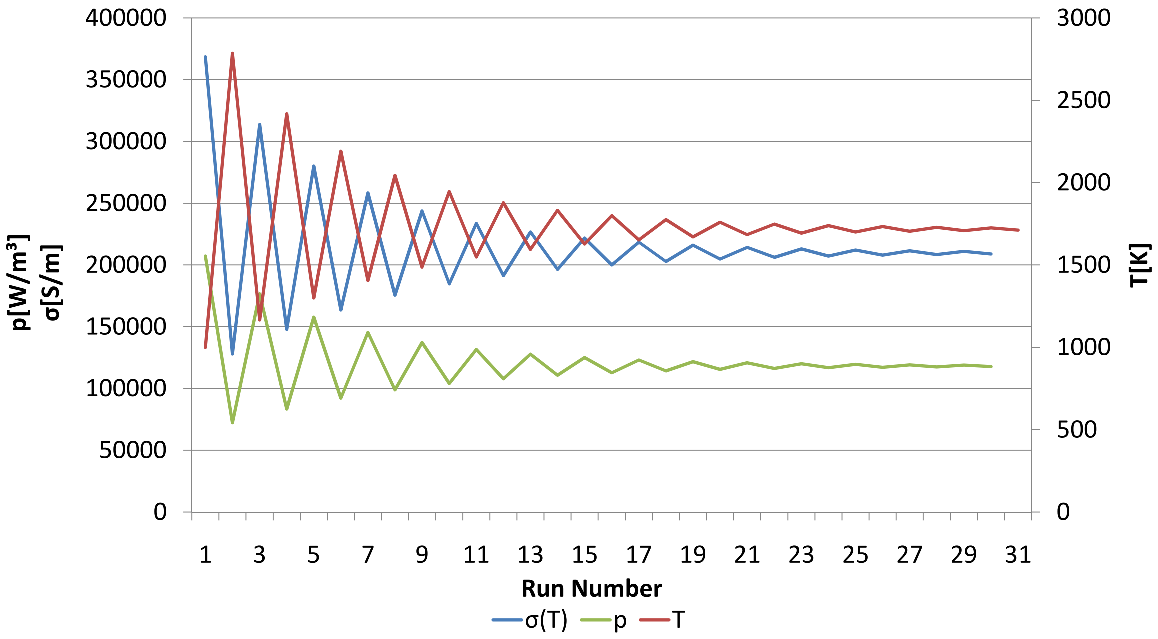

Figure 5.9: Convergence of temperature, power density, and specific conductivity over 30 coupling feedback runs

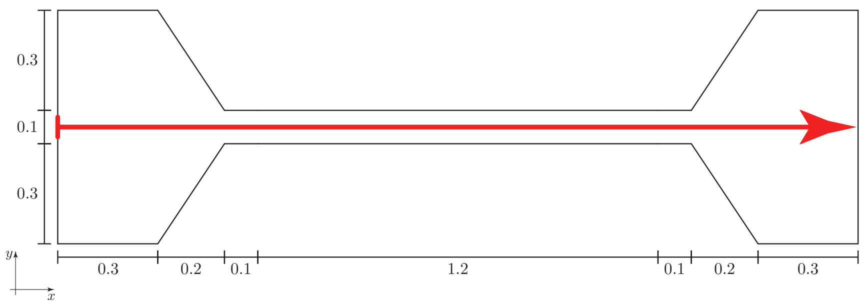

Figure 5.10: Cross-section through feeders and heating plate for evaluation of the results

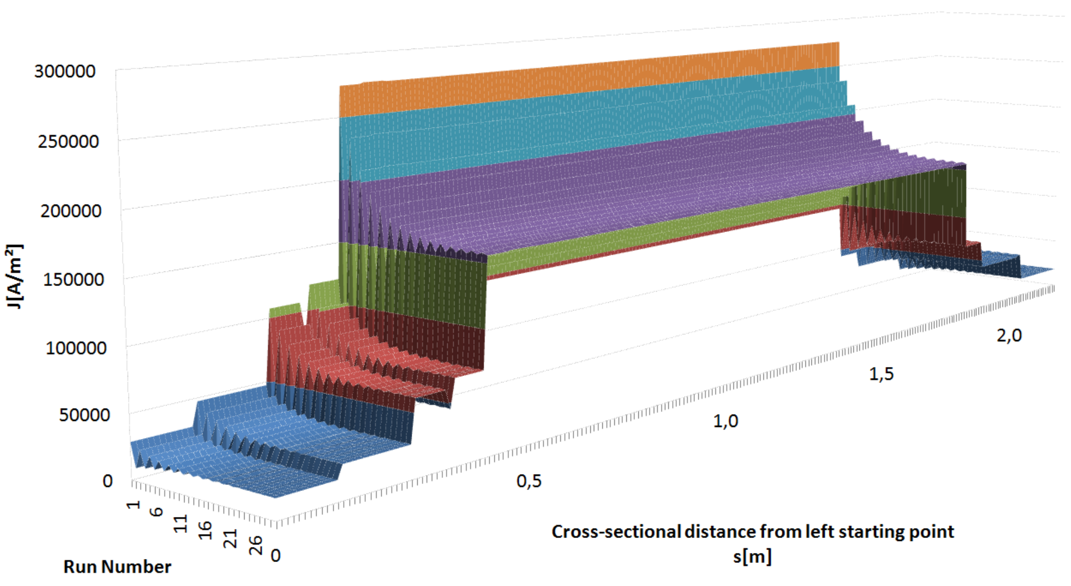

Figure 5.11: Convergence of the cross-sectional current flow over 30 coupling feedback runs

5.2 TEAM 22 Benchmark Problem

5.2.1 Problem Setup

In 1996 a Superconducting Magnetic Energy Storage (SMES) arrangement was selected to become a benchmark problem for testing different optimization algorithms, both deterministic and stochastic ones [61]. Since the forward problem can be solved semi analytically by Biot-Savart's law, this benchmark became quite popular. Nevertheless, the demands on optimization software have increased dramatically since then. To give an example, methods looking for Pareto-optimal points rather than for a single solution only have been introduced by several groups [57].

SMES are devices for energy storage within magnetic fields. These fields are produced by coils the windings of which consist of superconducting wires. The coils are connected to the power grid via a switch that facilitates feeding and discharching the apparatus. During persistent-mode operation both terminals of the coil are connected. Superconductors are the only material suitable for SMES as they lose their electrical restistance completely once they are cooled to appropriate, typically very low temperatures.

Two different approaches for SMES setups have been proposed, making use of solenoidal or toroidal coils. The advantage of solenoids is to offer easily manageable winding mechanisms, while manufacturing a toroid coil is much more intricate. Additionally, such a winding consumes almost double the amount of superconductiong material; albeit, a toroidal coil offers the benefit of the magnetic flux generated through its operation to be almost fully enclosed in the coil itself.

The setup under scrutiny within this research project consists of a configuration of two solenoidal coils, as depicted by figure 5.12. Such a configuration offers a significantly smaller magnetic stray field than a single solenoid, provided the coils are fed with currents oriented in the opposite direction. While such a setup requires more material than a setup using toroidal coils, at least the requisitions towards construction remain relatively low. Above that, a double solenoidal coil setup forms a marvellous example for optimization procedures due to its number of different, yet often conflicting, demands [2], [40].

An optimal design of the system was therefore agreed upon to couple the desired value of energy to be stored (first objective) with a minimal magnetic flux density's stray field (second objective) [3]. This problem has been accepted as TEAM benchmark problem number 22 [41].

The TEAM 22 benchmark problem offers two different approaches for performance assessment of optimization algorithms. The first one consists of a discrete problem, asking for three geometric parameters of the problem setup to be manipulated. These are radius, height and width of the outer solenoidal coil. The second approach features 8 continuous parameters to be orchestrated, being radius, height, width and current density of both inner and outer coil. Common to both problems is the fact that the objective function is defined as

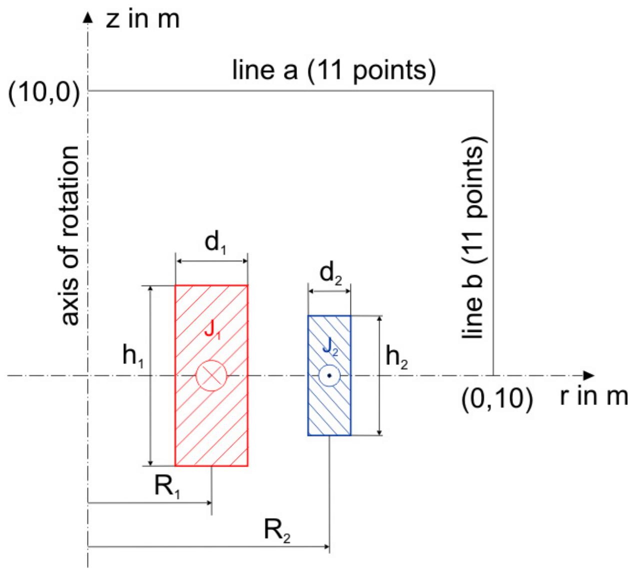

Figure 5.12: Configuration of SMES Benchmark Problem

a weighted sum of the two objectives (5.4).

Irrespective of the number of parameters to be optimized, for a SMES as described above the following objectives are in force and have to be considered:

1. The stored energy in the device should be as close as possible to 180 MJ.
2. The stray field B^2_{stray}(measured at a distance of 10 meters from the device) should be as small as possible.
3. The magnetic field must not violate a certain physical condition which guarantees superconductivity, the so-called quench condition as shown in (5.6) and figure 5.13.

However, the first two objectives are mapped into a single objective function given in (5.4).

$$OF = \frac{B^2_{stray}}{B^2_{norm}} + \frac{|E - E_{ref}|}{E_{ref}}, \tag{5.4}$$

where $E_{ref} = 180$ MJ, $B_{norm} = 200\ \mu$T and B^2_{stray} is defined by (5.5).

$$B^2_{stray} = \frac{\sum\limits_{i=1}^{22} \left|\vec{B}_{stray,\,i}\right|^2}{22}. \tag{5.5}$$

5.2.1.1 Quench Condition

An operating point that is to be avoided by any means is the return of parts of the superconduction coil to the normal resisitve operating state. This transition, called quench, immediately terminates the magnet's operation and may happen due to a number of different reasons. The most typical ones are either the magnetic field inside the coil being too large, or the frequency of current alternation is too high. The latter one would cause eddy currents in the supporting structure not made from superconductors, and thus heating of the whole apparatus.

Once a spot suffers from quenching and thus from a sudden return to its resistive state, it is subject to impetuous heating due to ohmic losses driven by the enormous current. This would have immediate effect to the neighbouring regions, as these would be heated up consequently and return to resistive state as well. The resulting chain reaction is a threat to safe operation of the device, and therefore has to be avoided, if possible, already during design. For superconductors of industrial use, the conditions under which safe operation can be guaranteed are well documented. (5.6) and figure 5.13 show the quench condition for a niobium-titanium alloy (Nb-Ti) as an example.

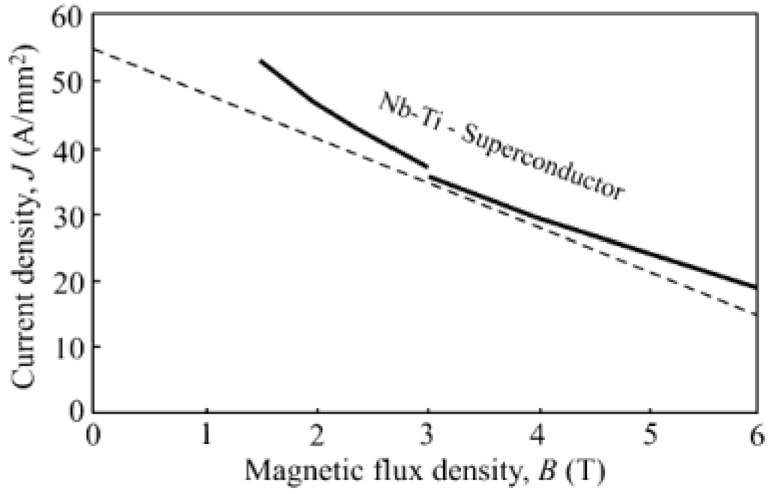

Figure 5.13: Critical quenching curve for a Nb-Ti industrial superconductor

$$|J_{\text{lim}}| = -6.4|B| + 54.0A/mm^2 \tag{5.6}$$

5.2.2 Multiphysical Coupling of Magnetostatic and Analytic Models

Other than coupling two models to be worked on using finite elements, in this case an analytic solution is of interest. That means, that calculations whether the quench condition is fulfilled or not can be conducted by the model calculation framework using the program code input interface for Python [72]. While chapter 4.2.2 describes the technical background, this section details the composition of a data structure to actually fulfill the task that is asked for.

The question to be answered can be subsumed as to calculate both total energy in the apparatus and the stray field, thus feeding the objective function as in (5.4). Then, identify and return a flag declaring the stated setup as being valid against the quench condition (5.6) as well.

This can easily be done by creation of two models within the input XML as in listing A.3. This offers the opportunity to retrieve the values of E and B^2_{stray} through FEM calculations and to immediately obtain declarative information whether the setup is valid or not through analytic calculations on base of the results delivered by the FEM solver. Though the above mentioned semi-analytical approach using Biot-Savart's law to solve this problem would suffice, calculations using a FEM model were employed for testing reasons.

The setup of the first model is shown in listing 5.9. Special attention is drawn to the model's *type* property. Infinite elongation in the third dimension is implicitly assumed for two-dimensional models such as the thermal hardening appliance of chapter 5.1. By contrast, the SMES model is of rotationally symmetric nature. Therefore, the solver has to be instructed to use the ordinate as rotational symmetry axis. For FEMtastic, this can be achieved by determining the *geometry* property as being *"rotational"* [37].

Solvers that do not handle symmetry as a directive would have do be advised to simulate a model in some symmetric way via the small detour of defining the modelling space as a macro element of its own with appropriate boundary conditions along the symmetry axes, for either rotational or axial symmetry. The decision of which procedure to use is left to the wrapper catering to the solver functionality and does not necessarily have to be part of the setup.

Listing 5.9: Magnetostatic calculation of SMES solenoidal coil field

```
<Model type="magnetostatics" geometry="rotational"
 solver="FEMtastic">
   <MacroElements>
      <MacroElement>
         <Geometry>
            <Add>
               <Polygon>
                  <Coords>
                     <Coord x="1.865" y="-0.8" />
                     <Coord x="2.135" y="-0.8" />
                     <Coord x="2.135" y="0.8" />
                     <Coord x="1.865" y="0.8" />
                  </Coords>
               </Polygon>
            </Add>
         </Geometry>
            <Source>
               <Value>22.5E6</Value>
            </Source>
      </MacroElement>
      <MacroElement>
         <Geometry>
            <Add>
               <Polygon>
                  <Coords>
                     <Coord x="2.9" y="-0.65" />
                     <Coord x="3.1" y="-0.65" />
                     <Coord x="3.1" y="0.65" />
                     <Coord x="2.9" y="0.65" />
                  </Coords>
               </Polygon>
            </Add>
         </Geometry>
                  <Source>
                     <Value>-22.5E6</Value>
                  </Source>
      </MacroElement>
   </MacroElements>
</Model>
```

The according model space and its discretization as well as the resulting magnetic field are shown in figures 5.14 and 5.15 as obtained using FEMtastic [37]. Figure 5.16, on the other hand, shows the same setup calculated using EleFAnT for verification purposes [24].

The Python code for checking the calculation for violation of the quench condition is comparably simple. It takes over four values – J_1, B_1, J_2 and B_2 – and calculates a decision whether this violation is reached or not. If safe operation can be ensured in both solenoids, 1 is returned, and 0 as return value for the erroneous case. The part of the input XML that carries the Python code for that model is given in listing 5.10.

Listing 5.10: Calculation of quench condition violation flag by means of analytic code

```
<Model type="analytic" refinement="" solver="Python">
  <Input>
    <Value>22.5E6</Value>
    <Value>-22.5E6</Value>
    <Import>
      <FromModel>1</FromModel>
      <Type>maxFromRegion</Type>
      <Threshold></Threshold>
    </Import>
    <Import>
      <FromModel>1</FromModel>
      <Type>maxFromRegion</Type>
      <Threshold></Threshold>
    </Import>
  </Input>
  <Code><![CDATA[
# input = [B1, B2, J1, J2]

J1check=(-6.4*input[0]+54)
J2check=(-6.4*input[1]+54)

# validates whether quench condition is fulfilled (1)
# or not (0, error case)
if (J1check > input[2]) or (J2check > input[3]):
     retVal = 1
else:
     retVal = 0

returnValues = []
returnValues.append(retVal)]]>
  </Code>
</Model>
```

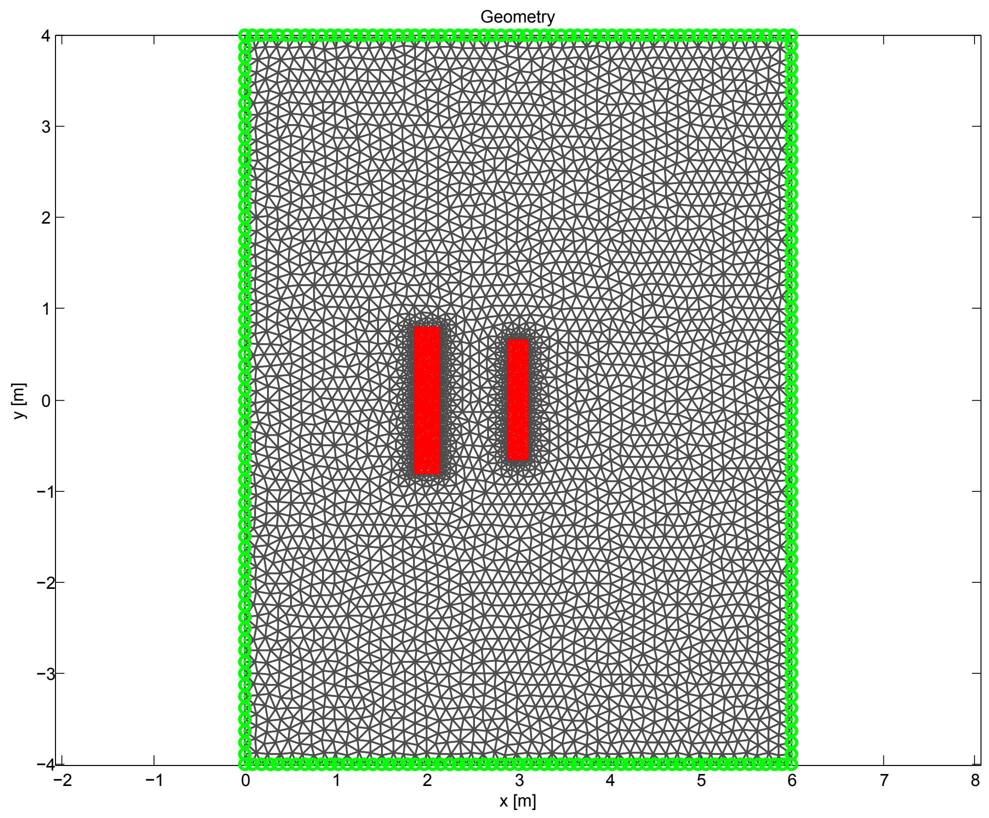

Figure 5.14: SMES problem space discretization

In- and output are handled by the according framework functionality blocks, and serve just the way they would do for FEM coupling: a postprocessor reconditions the solution of the first model and forwards it to the second one, where needed. Analogously, the return values from the second model to be solved by analytic means are available in just the same manner. That means for creation of the full computation output data the (for this example small) solution space stretched by the analytic code can be accessed by the evaluator.

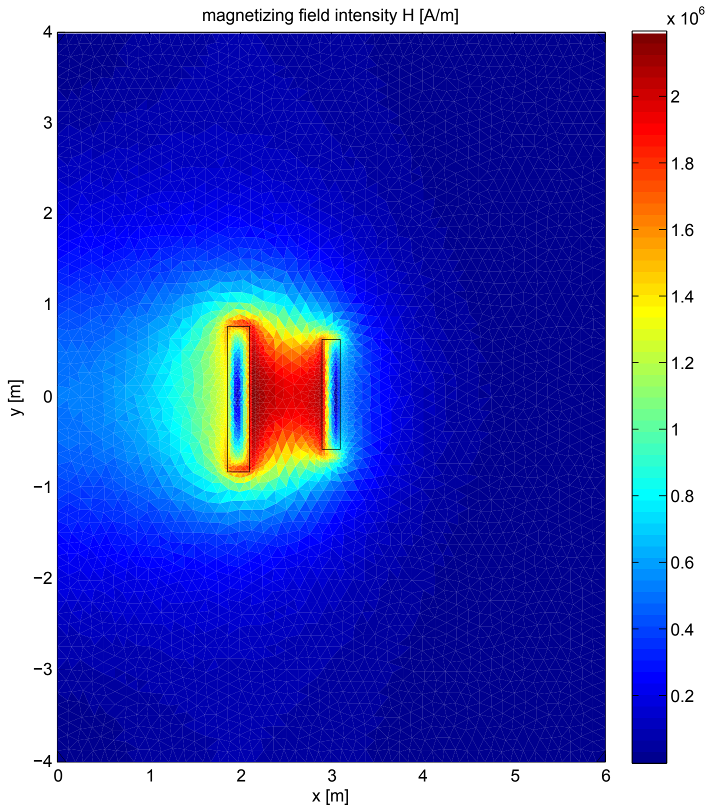

Figure 5.15: SMES solenoid coils' magnetic field intensity as derived from FEMtastic

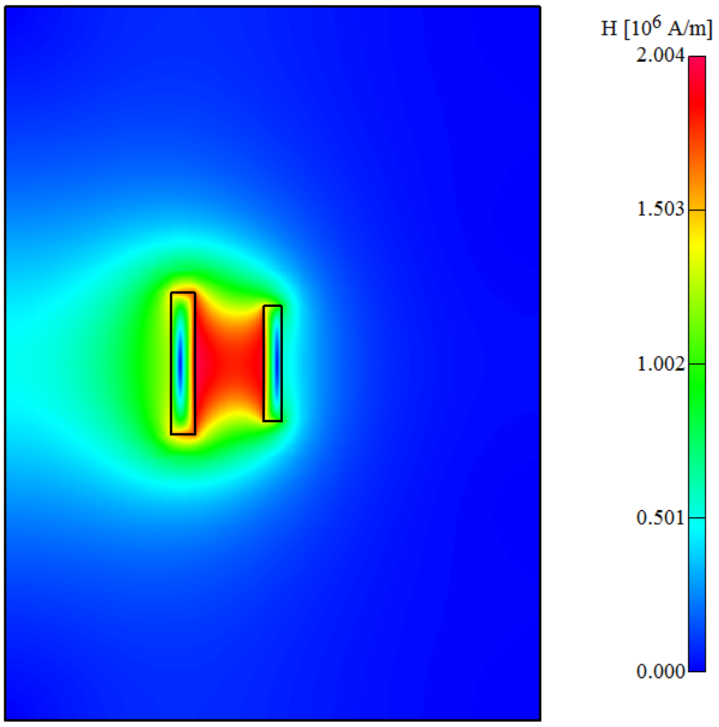

Figure 5.16: SMES solenoid coils' magnetic field intensity as derived from EleFAnT

5.3 Switching Operation within an R - L - Circuit

5.3.1 Problem Setup and Coupling Scheme

A quaint method of measuring currents can be modelled using a linear amperemeter. It features a cylindrical coil and a ferromagnetic core that works as an actuator. Figure 5.17 shows a sample setup. With current flowing through the coil's windings, a magnetic force draws the iron core towards the coil's center. Some sophisticated retention mechanism ensures the coil takes a linear movement depending on the excitation current. With the maximum amperage being reached, the iron core is fully inserted. Figure 5.18 shows the corresponding equivalent network.

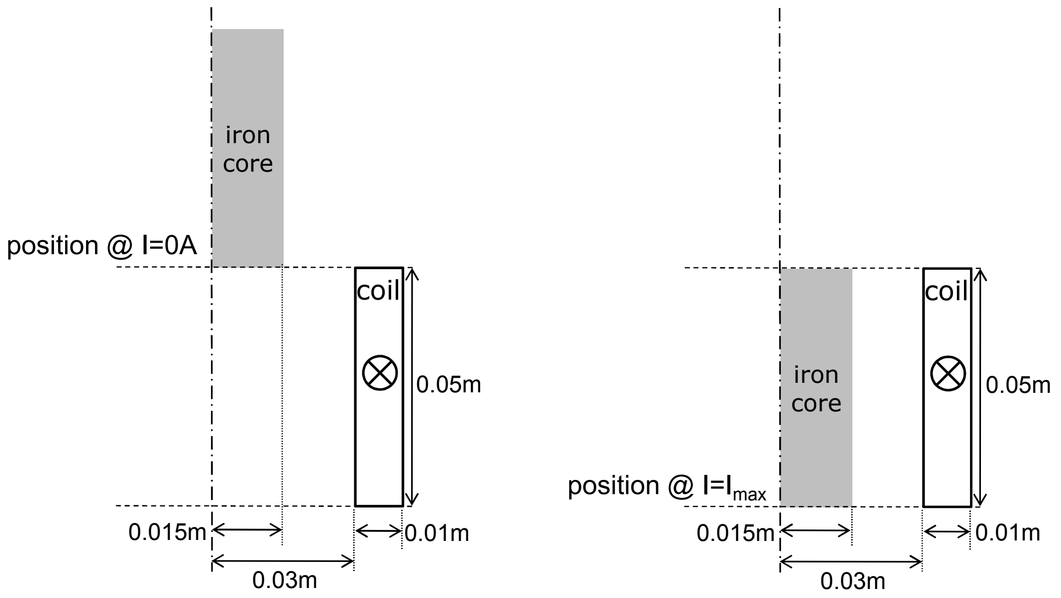

Figure 5.17: Cross-sectional axisymmetric model of an electromagnetic actuator

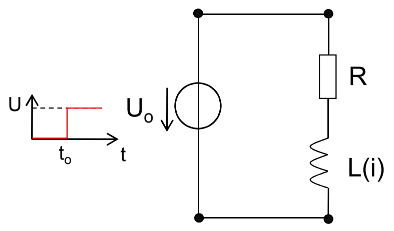

Figure 5.18: Diagram of the actuator's R-L circuit

Due to the iron core being inserted into the coil depending on the current, the coil's inductance is increased in a manner that does not allow any predictions. Simulating the behaviour of the device while switching on the voltage source therefore bears some imponderabilities, namely the amperage increase right after the switching event not only being dependent on

R and L, but L also depending on i in turn. The connection scheme can be transformed straightforward to the network equation (5.7). For the simulation, a Matlab network solver takes over the part of computing the retroactivity of the inductance on the thus arising current. FEMtastic takes care of simulations of the coil and iron core, thus paving the way to calculating the inductance for certain hardware configurations again.

$$L\frac{di(t)}{dt} + Ri = U_0 \Rightarrow \frac{di(t)}{dt} + \frac{R}{L}i = \frac{U_0}{L} \tag{5.7}$$

For setting up the simulation, the parameters outlined in table 5.2 and the geometry depicted in figure 5.19 have to be taken into consideration.

Voltage source	U_0	$12V$
Resistance	R	120Ω
Coil	number of windings, N	500
Iron core	μ_r	10^4

Table 5.2: Simulation parameters

Figure 5.19 shows the FEMtastic model with the iron core being in position $i = 0A$. The resulting XML input file is presented in full scope in appendix A.4.

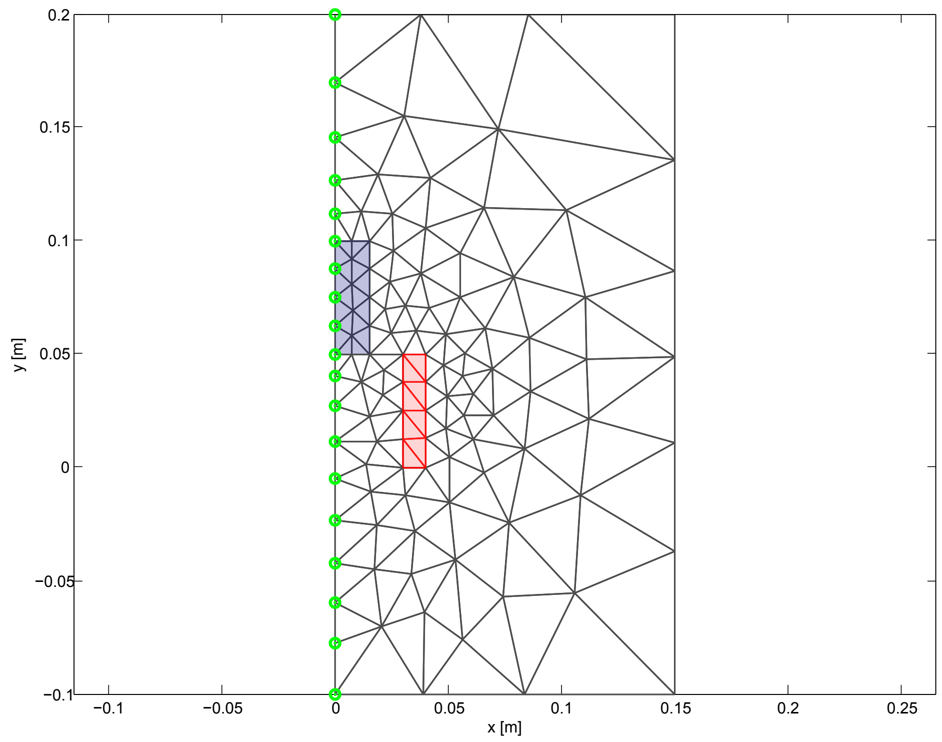

Figure 5.19: Geometry of the actuator's FEM model in position $i = 0A$

After obtaining a solution to the magnetostatic FEM problem, the postprocessor is queried for the full energy stored within the magnetic field. Based on the correlation of (5.8), the inductance follows directly:

$$W = \frac{L \cdot i^2}{2} \Rightarrow L = \frac{2 \cdot W}{i^2} \tag{5.8}$$

A graph depicting both W and thus the resulting L as a function of i is shown in figure 5.20. The nonlinear relation of the exciting current and thus the insertion state of the iron coil on the one hand and the inductance on the other is clearly visible, therefore vindicating the computational intensive use of a FEM solver for obtaining the correct value for the inductance at any given excitation amperage.

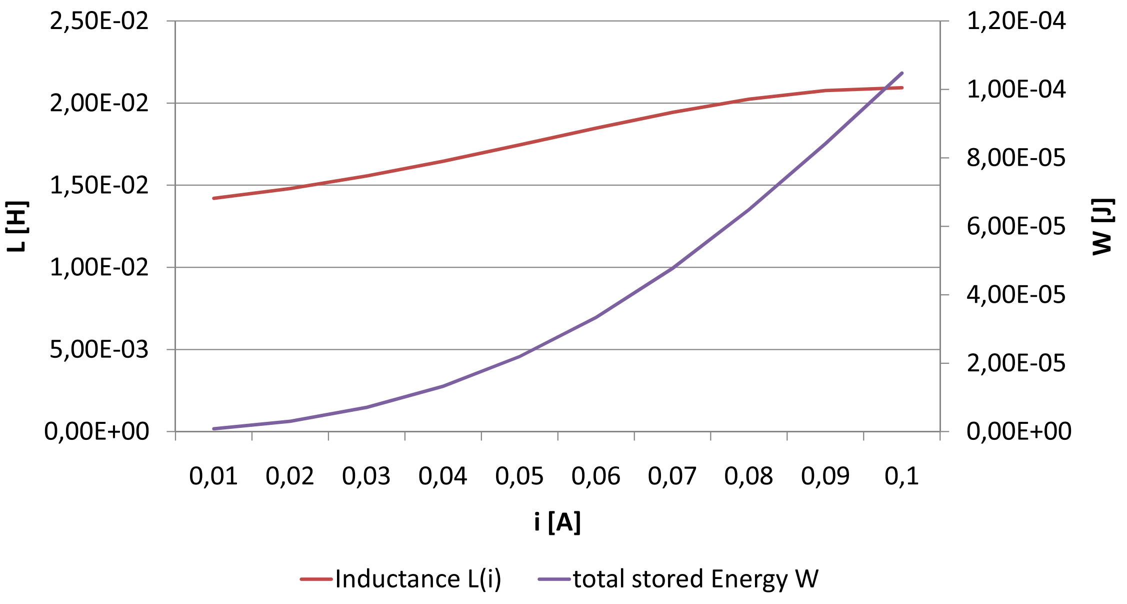

Figure 5.20: Total stored energy and inductance as functions of i

5.3.2 Application of Numerical Time-Stepping Methods

For the determination of the amperage over time as a consequence of switching on the voltage source, application of the self-contained solution (5.9) would not suffice because of the unpredictable value of L.

$$i_L(t) = \frac{U_0}{R}\left(1 - e^{-\frac{R}{L}t}\right) \tag{5.9}$$

Therefore, different varieties of the Euler method have been investigated for their convergence behaviour. Both the explicit and backward Euler methods share the characteristic of being a first-order numerical procedure for solving differential equations with a given initial value, while the Midpoint or Predictor-Corrector method is of second order, yet targeting the same purpose.

First, some preparatory work has to be carried out. As speaking of time-stepping methods, first time will have to be discretized. Given an initial point in time t_0, subsequent points t_k, and a time step of Δt, (5.10) follows.

$$t_k = t_0 + k \cdot \Delta t \qquad \forall k = 0, 1, 2, \ldots \tag{5.10}$$

With i being a function of time, the current at each point in time k can be stated as in (5.11).

$$i_k = i\left(t_k\right) = i\left(t_0 + k \cdot \Delta t\right) \tag{5.11}$$

Furthermore, developing $i(t)$ into a Taylor series would result in (5.12).

$$i_{k+1} = i\left(t_k + \Delta t\right) = i\left(t_k\right) + \left.\frac{di}{dt}\right|_{t_k} \cdot \Delta t + \left.\frac{1}{2}\frac{d^2 i}{dt^2}\right|_{t_k} \cdot \left(\Delta t\right)^2 + \ldots \tag{5.12}$$

Given the fact that Δt should already be selected very small, $(\Delta t)^2$ and all subsequent terms of the Taylor series grow insignificantly small. Therefore, the series is reduced to (5.13).

$$i_{k+1} \approx i_k + i'_k \cdot \Delta t \tag{5.13}$$

5.3.2.1 The Explicit Euler Method

Also known as the forward method, it develops i_{k+1} solely on base of data already known at t_k. Generally speaking, Euler methods develop the shape of a curve yet unknown, but of which a starting point and a differential equation the curve is satisfying is given. The differential equation allows calculating the slope of a tangent line to the curve for any point part of the curve, as soon as that point's position has been identified.

As mentioned above, starting from a given point – $i(t_0)$ – within the current example, the slope to $i(t)$ at $t = t_0$ can be obtained, and thus its tangent. For a step size Δt small enough, the tangent line resembles $i(t)$ sufficiently. Therefore, the same reasoning is applied to $i(t_0 + \Delta t)$ in order to obtain the next value. The curve resulting from several time steps stays reasonable close to the exact curve, given the time step is small enough, as already mentioned.

Combining (5.13) with i' from (5.7) and the inductance's dependency on i_k, (5.14) follows.

$$i_{k+1} = i_k + \left(-\frac{R}{L_{(i_k)}} i_k + \frac{U_0}{L_{(i_k)}}\right) \cdot \Delta t = i_k \cdot \left(1 - \frac{R}{L_{(i_k)}} \cdot \Delta t\right) + \frac{U_0}{L_{(i_k)}} \cdot \Delta t \tag{5.14}$$

In figure 5.21, the values derived for i_k are depicted using the reddish curve. It converges to an error of less than 1% within 12 simulation runs. Compared with the other examined methods, this one therefore is the fastest which is due to the fact of the determined values for i_k being inherently situated *above* the true curve.

5.3.2.2 The Backward or Implicit Euler Method

The backward Euler method is of an implicit nature, denoting that in order to obtain i_{k+1}, an equation or even an equation system has to be solved. This is due to i_{k+1} not being given explicitly, but depending on its own, yet unknown value.

$$i_{k+1} = i_k + i'_{k+1} \cdot \Delta t \tag{5.15}$$

In other words, the tangent line that is employed at i_k is assigned the slope that is to be expected at i_{k+1} as in (5.15), which unfortunately is still unknown. With the explication of i'_{k+1}, the dependency develops to (5.16).

$$i'_{k+1} = -\frac{R}{L} i_{k+1} + \frac{U_0}{L}\bigg|_{k+1} \tag{5.16}$$

For this example, the dependency can be solved into (5.18). In case such a solution cannot be obtained, other numeric methods have to be employed. Their cost of computation, of course, has to be taken into account for any decisions on employing this method to approximating differential equations' solutions.

$$i_{k+1} = i_k - \frac{R}{L_{(i_{k+1})}} \cdot i_{k+1} \cdot \Delta t + \frac{U_0}{L_{(i_{k+1})}} \cdot \Delta t \rightarrow i_{k+1}\left(1 + \frac{R}{L_{(i_{k+1})}} \cdot \Delta t\right) = i_k + \frac{U_0}{L_{(i_{k+1})}} \tag{5.17}$$

$$i_{k+1} = \frac{i_k + \frac{U_0}{L_{(i_{k+1})}}}{\left(1 + \frac{R}{L_{(i_{k+1})}} \cdot \Delta t\right)} \tag{5.18}$$

Figure 5.21 shows the collection of calculated values for i_k in green. Due to the nature of i_{k+1} offering a slope that is less than i_k's, the curve resides steadily *below* the true values. Therefore, it converges rather late. The 15th run has yielded an error less than 1%, again. Nevertheless, the errors are generally smaller than those for values obtained by the explicit Euler method.

5.3.2.3 The Predictor-Corrector or Midpoint Method

This method is part of a collection of higher-order methods, usually referred to as Runge-Kutta methods. Its name is derived from the fact that the slope to apply at i_k in order to find i_{k+1} is obtained at the point in the very middle of these, namely at $t = t_k + \frac{\Delta t}{2}$.

Obtaining i_{k+1} requires two steps to be executed, therefore this method is among a class of so-called linear multistep methods. The first retrieves an approximate value $\tilde{i}_{k+1}$, therefore (vaguely) predicting its value. The explicit Euler method as in (5.14) works nicely for this purpose, as it only relies on already known values at t_k.

Then, a correcting step is taken as shown in (5.19). It fulfills above mentioned midpoint claim inherently.

$$i_{k+1} = i_k + \frac{\Delta t}{2} \cdot \left(\tilde{i}'_{k+1} + i'_k\right) \tag{5.19}$$

While being computationally more extensive than the Euler methods mentioned above, the results are regarded as more accurate. In figure 5.21, the results obtained using the predictor-corrector method are drawn in blue.

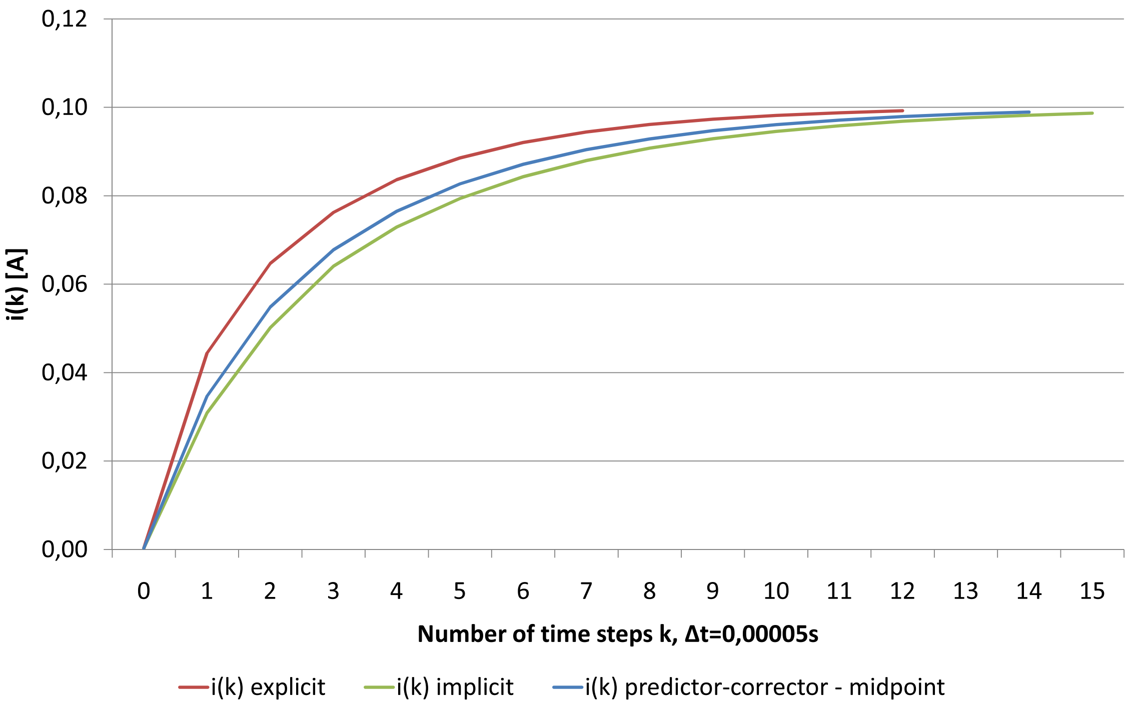

Figure 5.21: Run of the excitation current curve for different numerical methods

Figure 5.22 finally compares the run of the current curve obtained by the midpoint method with two analytically calculated solutions, both offering static inductances as retrieved for $i = 0A$ and $i = i_{max}$. The midpoint method's curve applying a nonlinear, varying L_i is expectedly oriented along the curve obtained with $L_{i=0}$ in its first parts, while later increasingly approximating the curve resulting from calculations with $L_{i=i_{max}}$ and eventually converging with it.

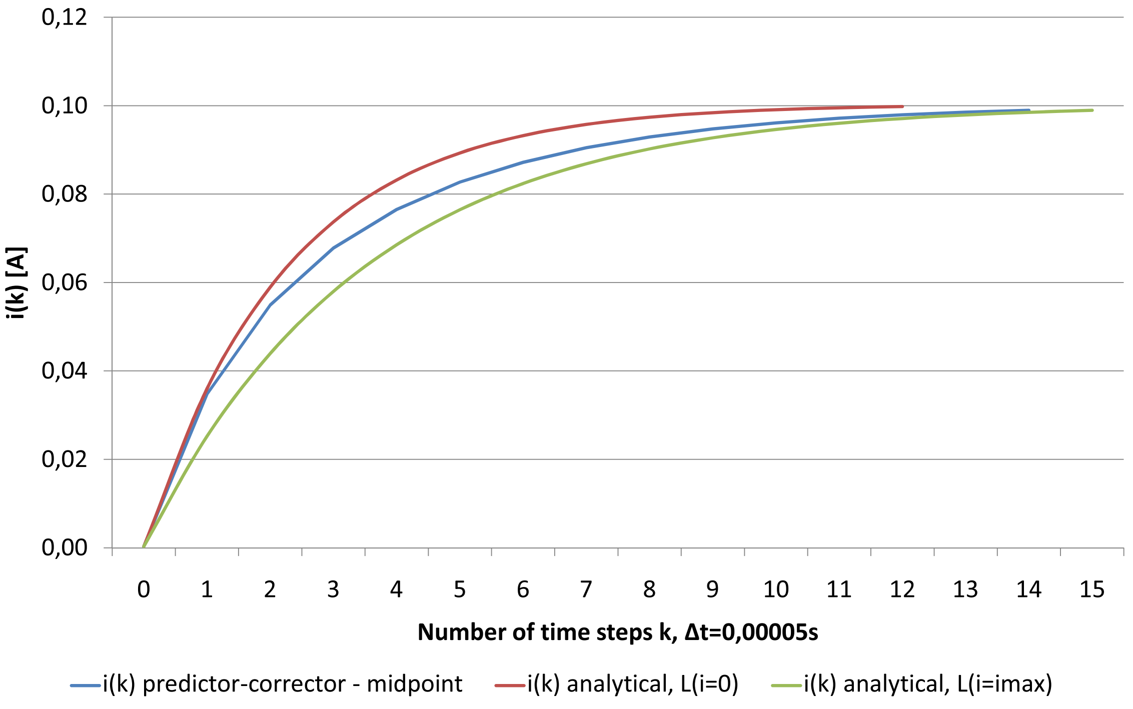

Figure 5.22: Run of the excitation current curve for a nonlinear $L_(i)$ in comparison to analytically obtained solutions

Chapter 6

Retrospect and Outlook

"The outcome of any serious research can only be to make two questions grow where only one grew before."

[Thorstein Bunde Veblen]

The fact that two or more models of different nature and simulation requirements are calculated one after another and linked losely through exchanging certain simulation results determines the coupling state as being weak. In comparison to strong coupling, where load vectors and stiffness matrices of two models are combined into one equation system, this approach may consume more computing time. On the other hand, it offers unmatched easier operation to the user, and keeps both number and characteristics of the models to be coupled unlimited.

For this reason weak coupling was incorporated as an objective towards the software framework detailed in the previous chapters, hazarding the consequences of software performance issues. Therefore, computational performance of single modules play an eminent role in such surroundings. For FEMtastic, the FEM solver developed as part of this research project, investigations concerning conceptual correctness and operational capacity have been conducted.

6.1 Proof of Concept

As mentioned in preceding parts of this work, EleFAnTs is a well established player in the field of FEM software applications. It was therefore used to verify results obtained by FEMtastic and the surrounding software framework.

Figure 6.1 shows the setup and mesh of the hardening device problem's current flow setup, as detailed in chapter 5.1, modelled using tools provided for EleFAnTs. Geometry and material properties as well as boundary conditions have been chosen to coincide. The

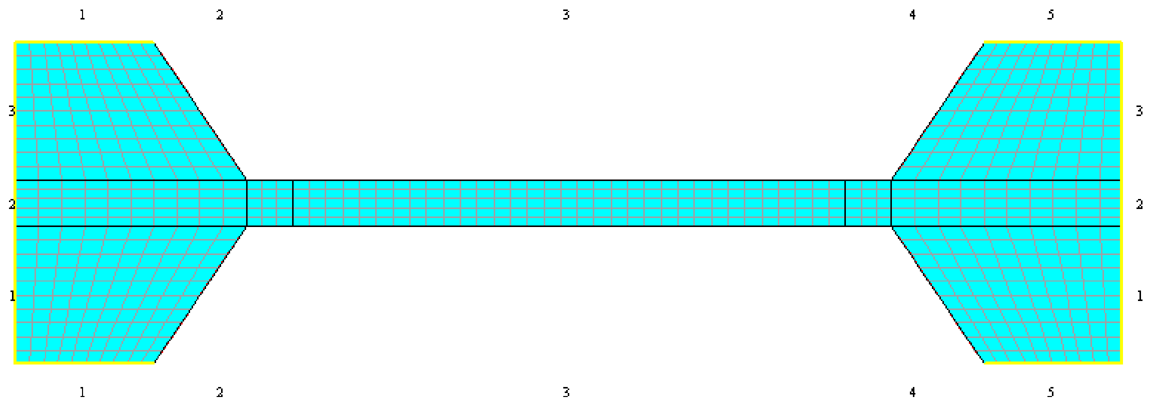

Figure 6.1: EleFAnTs' current flow model

left electrode is assigned an electric potential of $u = 12V$, the right one exhibits $u = 0V$, the material conductivity of the electrodes amounts to $\sigma = 60 \cdot 10^6 S/m$ while the heating plate's conductivity is $\sigma = 420S/m$.

Figure 6.2 illustrates the conformity of results obtained by obtained by EleFAnTs (top) and FEMtastic (bottom). This is demonstrated in more detail in figure 6.3, showing the electric potential along a straight horizontal line through the center of the modelled area. Figures 6.4 and 6.5 present the same conformity for the resulting electric field, figures 6.6 and 6.7 for the current flow.

For the thermal problem's solution domain, results comparison yields a similar outcome [37], thus confirming the correctness verification.

6.2 Performance Evaluation

MATLAB has been chosen as development environment for a couple of reasons, all of them justified in chapter 4. While for assembling software modules of different origin and functionality MATLAB has proven well-suited, for substantial numerical calculations this predication cannot be maintained.

Although by comparison of different interpreted languages only small deviations can be identified [70], this is not necessarily true for comparing MATLAB code, especially object oriented one, with other OOP languages. The differences in execution time may well accumulate to factors of 50 to 100 between slow interpreted and fast compiled code [7].

The reasons for such a significant difference shall be illuminated in the following paragraphs.

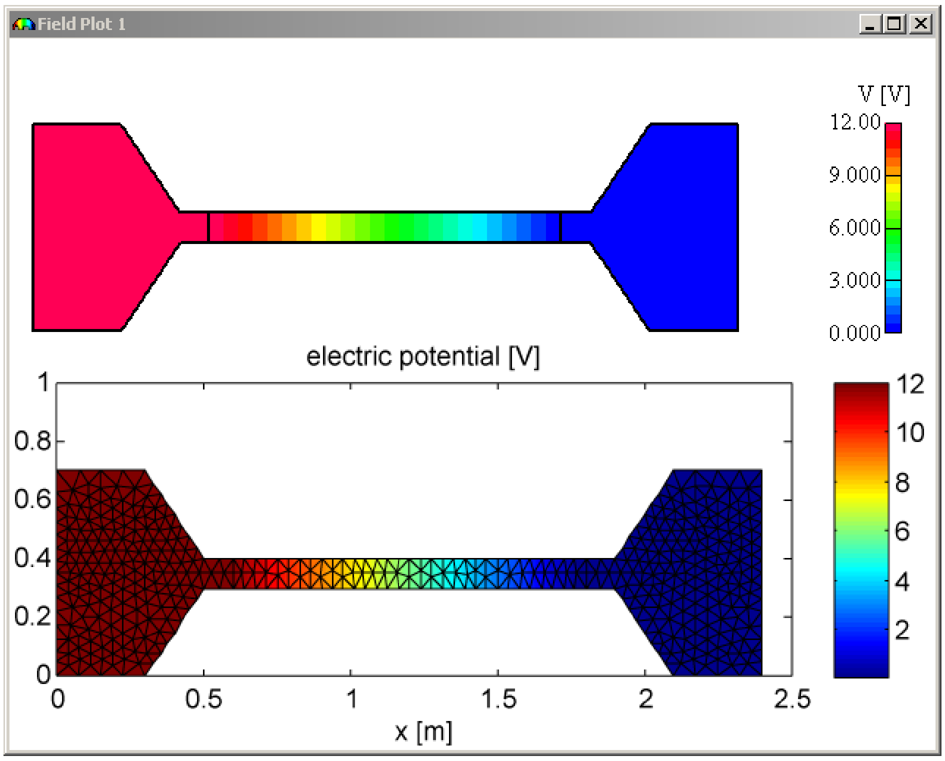

Figure 6.2: Comparison of electric potential u modelled by EleFAnTs (top) and FEMtastic (bottom)

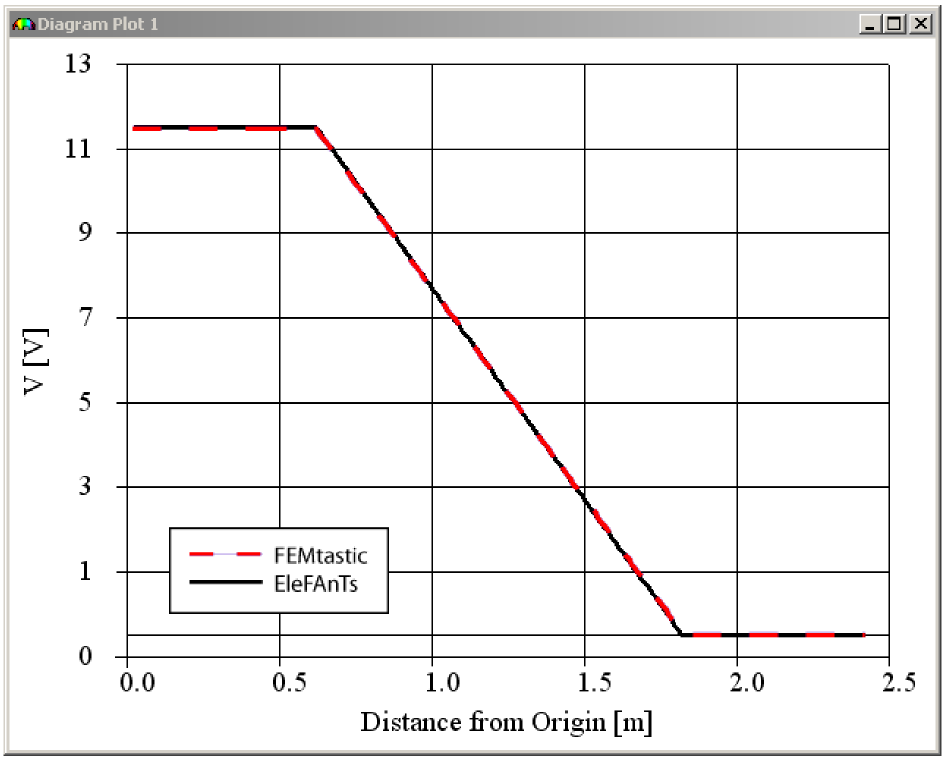

Figure 6.3: Comparison of the electric potential u along a straight line

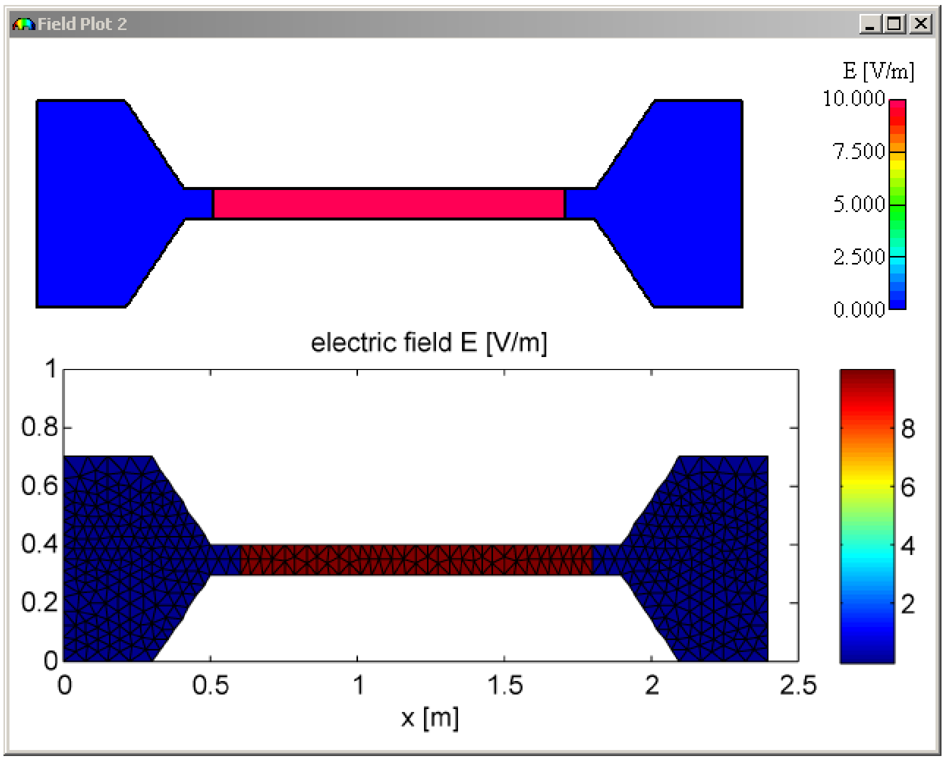

Figure 6.4: Comparison of electric field $\mathbf{E}_x$ modelled by EleFAnTs (top) and FEMtastic (bottom)

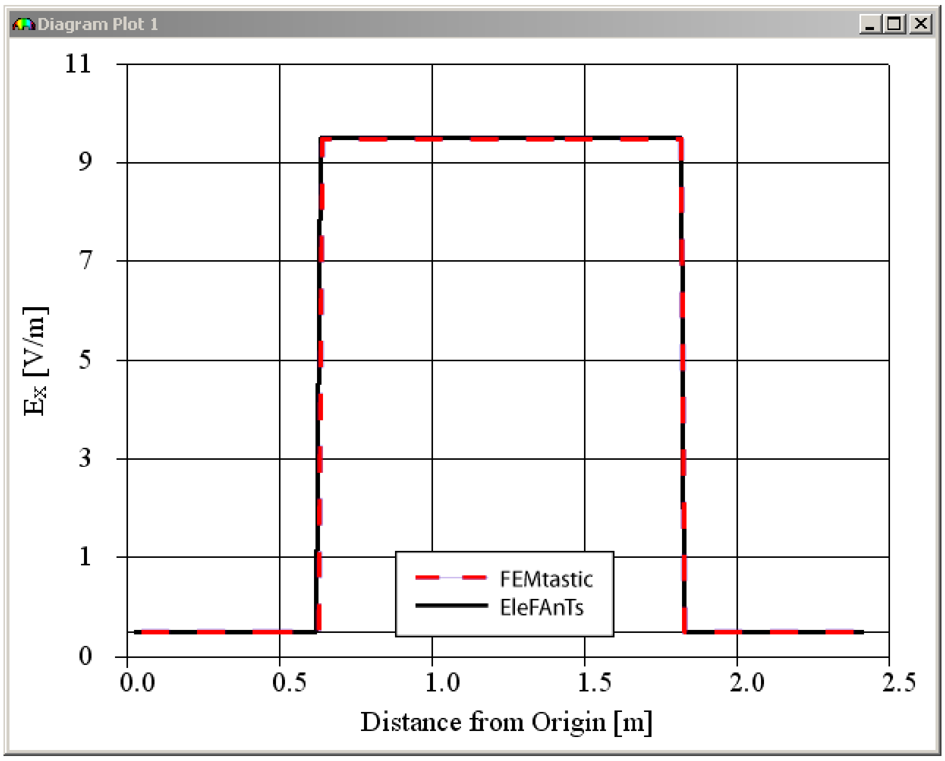

Figure 6.5: Comparison of the electric field $\mathbf{E}_x$ along a straight line

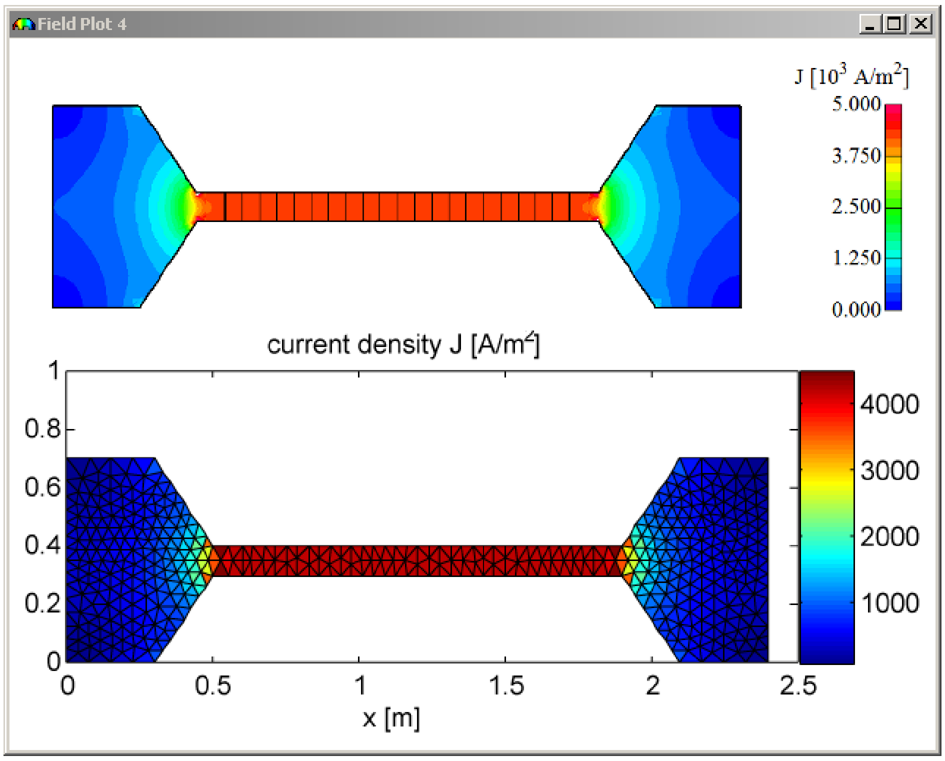

Figure 6.6: Comparison of current flow $\mathbf{J}_x$ modelled by EleFAnTs (top) and FEMtastic (bottom)

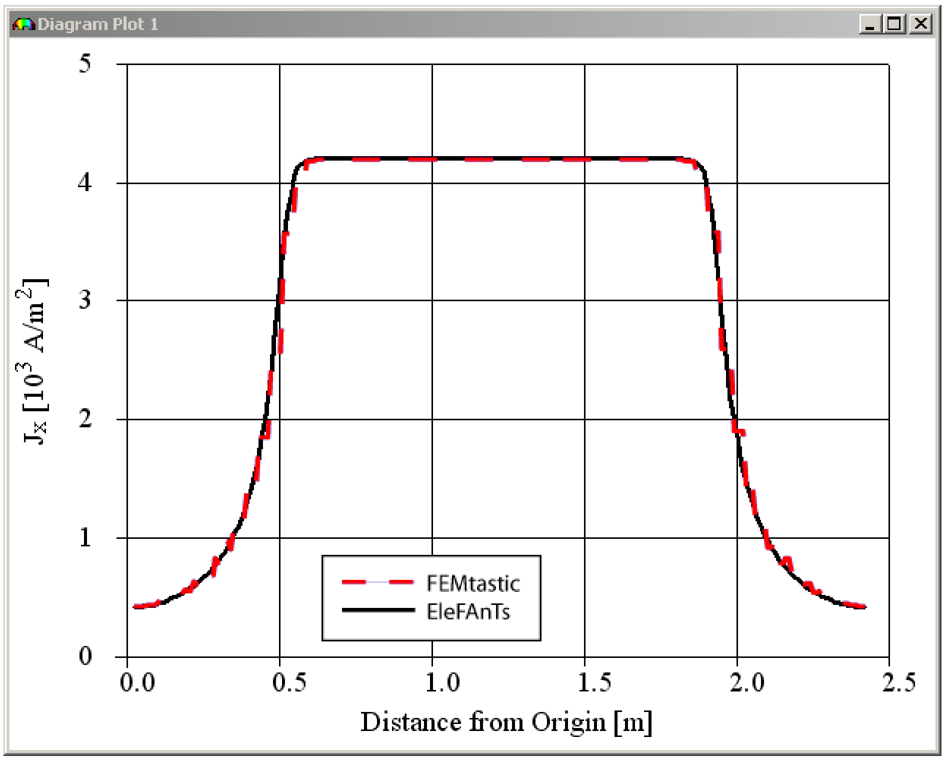

Figure 6.7: Comparison of the current flow $\mathbf{J}_x$ along a straight line

6.2.1 CPU time

For clarification of the behaviour of the developed FEM tool, a number of calculations on the current flow model described in chapter 5.1 were conducted. Different mesh refinements of the same problem topology yielded finite elements numbers ranging from 91 to 64188. In diagram 6.8, the computation times of each step in the programme flow are summed up to the total time, depicted by the red line at the top. The number of finite elements in each test calculation is delineated along the x-axis. Interpolation between actual measurements (flagged with markers) using second order polygons yields the actual curves, which implies an algorithm complexity of $\mathcal{O}(n^2)$. This is plausible inasmuch as the problem's stiffness matrix grows quadratically with an increasing number of finite elements.

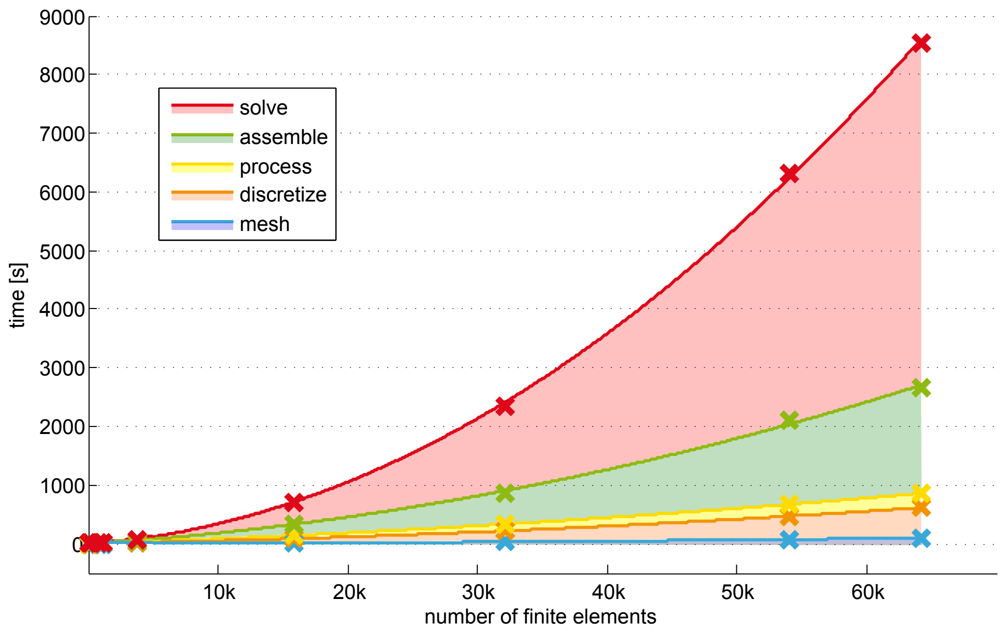

Figure 6.8: Cumulated computation time for different numbers of finite elements

Hovewer, a lack of efficiency in executing programme code is also raised to the second power, even though algorithms may prove to treat certain problems efficiently. This is why further development of the given software towards specialized, compiled, object oriented software such as treated in [56] is highly recommended.

Figure 6.9 shows the percentage share of the single computation steps. With increasing numbers of finite elements and consequentially increasing numbers of unknowns in the equation system to be solved, the MATLAB native solver accounts for ever increasing shares of the execution time. For a stiffness matrix greater than $20k \times 20k$ elements it already expends more than 50% of total CPU time. Equation system solvers specialized in handling FEM matrices and their peculiarities may help in sparing time there.

A great amount of time during system assembly is used by Dirichlet boundary conditions'

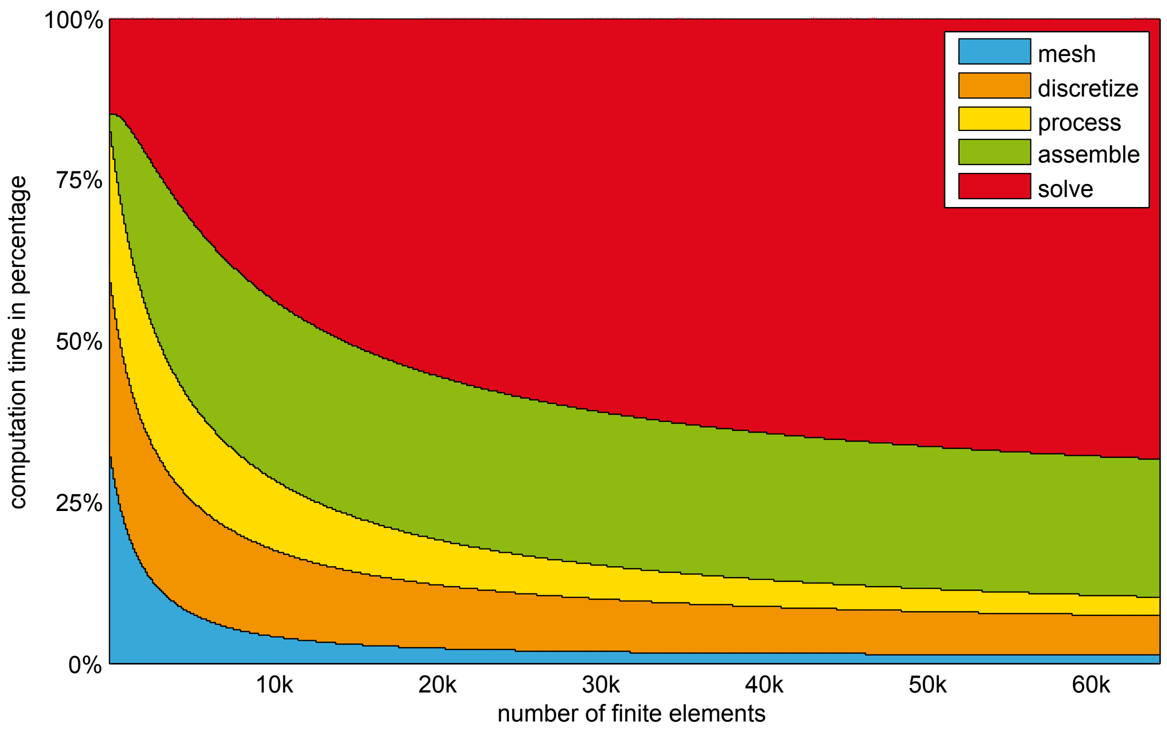

Figure 6.9: Percentage share of the full CPU time of different programme steps

homogenization. Removing 514 rows and columns from a stiffness matrix of 64188×64188 entries takes almost three times as long as assembling the actual system.

The combination of the blue, orange and yellow areas in figure 6.9 can be interpreted as the preprocessing steps. The green and red areas together form the actual FEM computation.

The yellow band in figure 6.9 represents the the search for correct edges for Neumann and Cauchy boundary conditions, having to process through all finite elements within a mesh in order to find them. Though not having that much of a stake in the time balance sheet, this step could as well be integrated directly into the meshing algorithm, saving processor time again.

Table 6.1 shows the detailed results of the measurements, conducted on an Intel T7250 CPU with 2 GHz clock speed, 2 MB L2 cache and 2 GB of working memory.

mesh refinement	1	0.1	0.05	0.03	0.02	0.01	0.0063	0.00625	0.005
finite elements	91	179	803	1152	3768	15906	32074	53987	64188
mesh	1.64	1.75	2.22	2.71	5.57	19.00	32.86	69.53	104.08
discretize	0.58	0.94	3.71	5.16	16.75	76.60	183.66	399.77	514.73
process	0.30	0.62	2.67	3.62	12.64	54.58	114.32	201.74	243.47
assemble	0.41	0.73	3.10	4.48	14.96	183.68	529.20	1441.48	1793.15
solve	0.05	0.12	0.29	2.61	22.95	371.44	1496.78	4201.45	5886.14
total	2.98	4.16	11.99	18.58	72.87	705.30	2356.82	6313.97	8541.57

Table 6.1: Absolute figures of CPU time usage (times in seconds)

6.2.2 Memory Usage

Apart from some status and helper variables, the main share of memory space is consumed by finite element objects, node objects, the stiffness matrix and the load vector. The stiffness matrix is a sparse matrix, which is inherently optimized by MATLAB to only store values to memory that actually differ from zero. In this respect, memory consumption by the algorithm depends in a linear manner from the number of finite elements to be processed. Figure 6.10 substantiates that interrelation.

Table 6.2 summarizes the peak values of memory consumption of the sequence of computation tasks. Each task's memory is cleared after finishing execution. It's worth to notice that unlike for CPU time, finite element processing does not consume any noticeable memory at all. This is due to the finite element objects already are created in memory, and results of the calculations performed on each of them (e.g. Jacobian matrix) only occupy memory space reserved for member variables already allocated before.

Above a certain number of finite elements that are processed, the solver accounts for the maximum amount of memory consumption. Nevertheless, its memory usage still remains almost linear along the monitored range.

This approximate linearity also gets clear from figure 6.11, depicting the percentage shares of each computational task's memory consumption. The nonlinearity of measured values below around 10'000 finite elements can be explained by each algorithm's helper and service data structures still being of greater influence than the actual data containers.

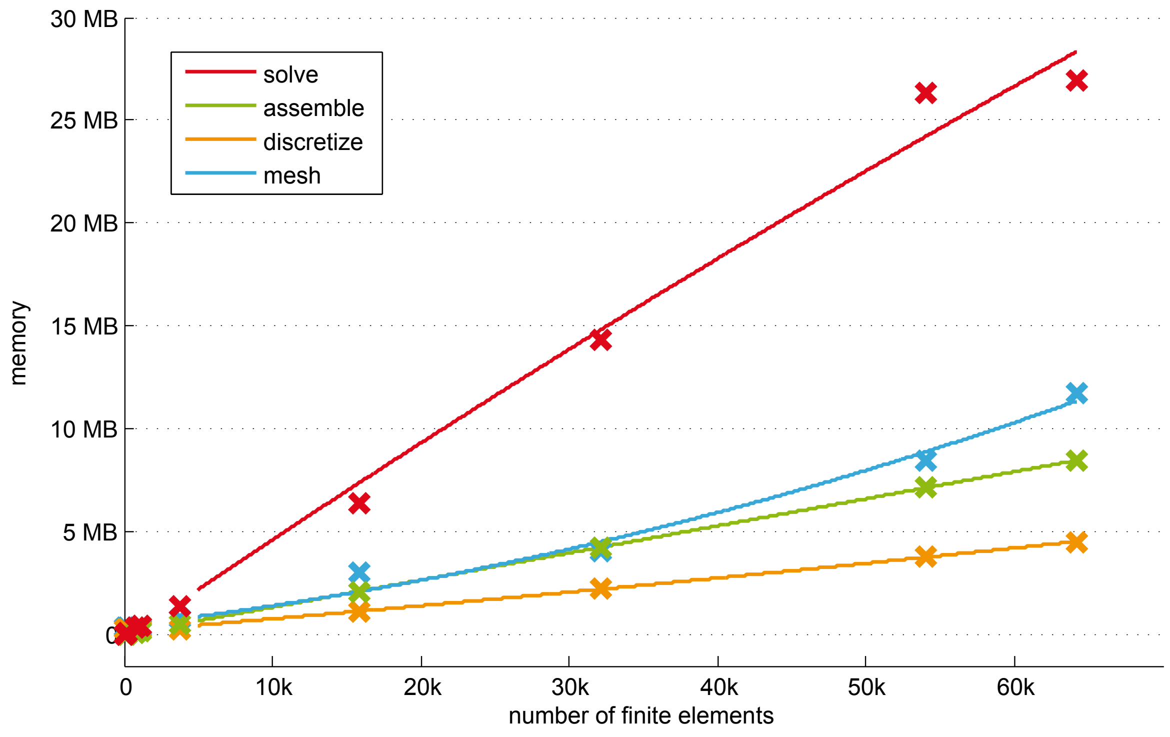

Figure 6.10: Memory usage for different numbers of finite elements

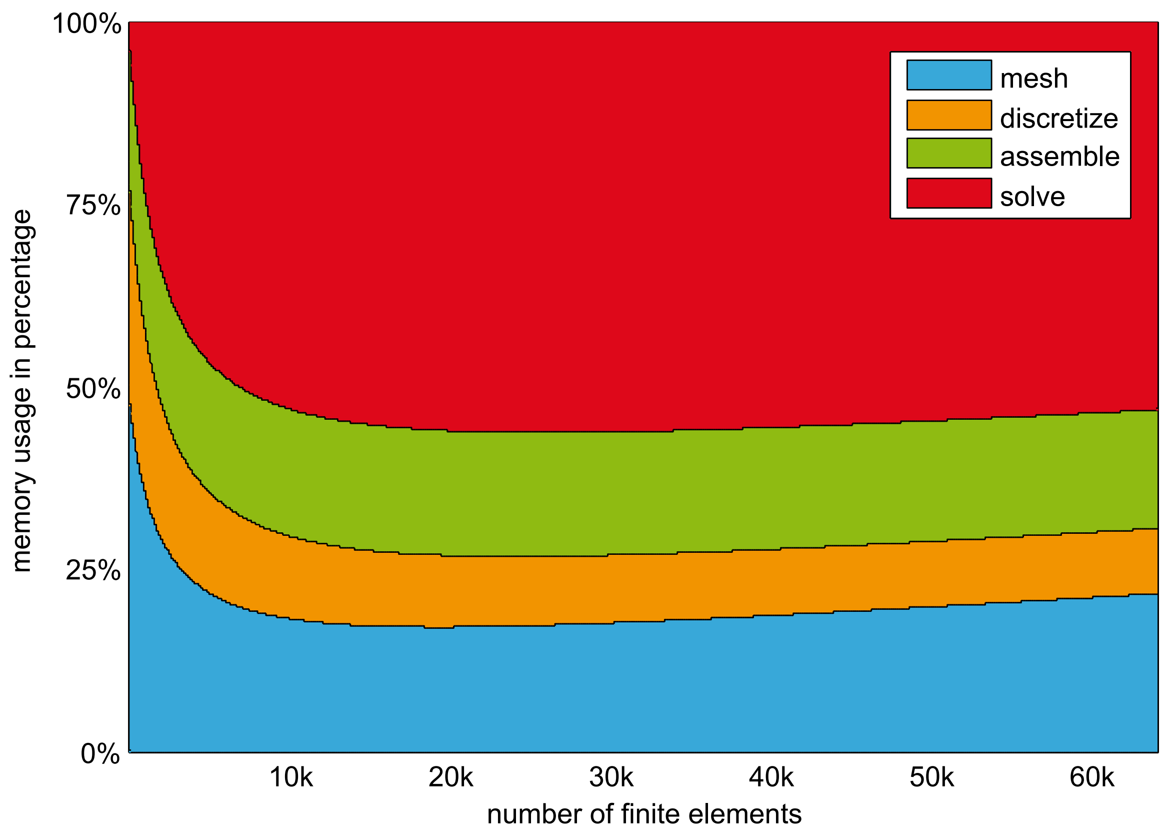

Figure 6.11: Percentage share of memory consumption of different programme steps

6.3 Framework Design

Implementation and verification of the simulation framework and FEMtastic against current research problems left a couple of issues for further investigation. Computational efficiency is one of the larger ones, and has been reviewed in the previous chapters. Porting the FEM solver to more efficient data structures offered by other programming languages now obviously is the next step to promote developments in this application area.

In contrast, for the overall software framework design, MATLAB has proven to be accurate and flexible in general. It allows a clear differentiation between software modules and

mesh refinement	1	0.1	0.05	0.03	0.02	0.01	0.0063	0.00625	0.005
finite elements	91	179	803	1152	3768	15906	32074	53987	64188
mesh	376	376	376	376	826	3013	4077	8438	11718
discretize	241	241	241	241	265	1118	2255	3796	4513
assemble	46	46	115	163	513	2119	4248	7137	8475
solve	49	67	337	434	1366	6422	14355	26375	26933
peak	376	376	376	434	1366	6422	14355	26375	26933

Table 6.2: Absolute figures of memory consumption (values in KB)

components, uniting them into common data structures, and thus making them accessible for the computation of coupled problems.

Also, the use of XML as common data interchange format has proven to cater to a number of different problem setups. Nevertheless, when pushing forward the software functionality's capabilities, the notation of problem geometry in XML may turn out to be inefficient, maybe even not sufficient to cover all needs.

IGES, the *Initial Graphics Exchange Specification*, should be mentioned as one of the resources worth investigating. It is a free, well established, manufacturer independent data format that is widely used for data interchange among computer-aided-design products [64], [35].

The incorporation of a data format like this would also overcome two drawbacks the current framework exhibits. Firstly, it would be possible to also work with complex three-dimensional topographic layouts. Secondly, through incorporating a prevalent data format also for internal purposes, interfaces to external programme modules would be more consistent, and the complexity of and efforts to create wrappers could therefore be reduced.

6.4 Parallelization

Modern computing equipment with cheap multi-core or multi-processor infrastructure allows to ventilate parallelization of the computational load. The implementation of parallelized algorithms usually contributes significantly to speeding up a numerical simulation and optimization process [51].

Parallelization for a full application framework can be achieved on two levels:

- *Module parallelization* means to run two independent software modules simultaneously. This is, from a data management point of view, an easy task; yet, for the framework under scrutiny, model coupling typically inhibits such independency. Even with entirely different model setups, at some point data needs to be interchanged. An idea would be to at least build both models until an import of results from another model becomes necessary, then halt the execution of the affected model until the missing data becomes available. Depending on model type and nature of coupling, this may at least save parts of the model building process from being performed consecutively.

  Alternatively, for couplings exchanging data back and forth until meeting some convergence criterion (see chapter 4.6.2), also some predictor-corrector functionality could be contemplated [32]. Such an approach would enable both solvers to work on their solutions, even though no definite solution to be taken over can be offered, but only a prediction. For the next iteration, this prediction can be rectified so that convergence is consequently met in the further course of calculation.

- Intra-module or algorithm parallelization usually has to be left to module suppliers. Simple parallelization of tasks consuming lots of processor time is inherently offered by any modern operating system inasmuch as programme state changes and calculations are performed on one processor or core, and all memory operations on the other. This already speeds up computation significantly. More drastic attempts to parallelize computation at least require arrangements within the programme code, or even considerate algorithm design.

Both of these approaches can be supported via the use of MPI, the Message Passing Interface [23]. MPI is used in a vast spectrum of cases for solving voluminous scientific or engineering problems using parallelized infrastructure. It can be characterized as an API specification, allowing processes to communicate with one another through sending and receiving messages.

MPI is implemented for all established operating system platforms as well as for all current programming languages. Python and Matlab are no exemption to this.

Although MPI is neither defined nor supervised by any major standardization body, it has developed into a de facto standard for communication among parallel processes. MPI remains the dominant model used in high-performance computing today [66].

Apart from MPI implementations, MATLAB offers extensive parallelization support as well. Though in 1995, Mathworks officials asserted that there was no market for parallel MATLAB functionality [48], this focus has shifted enormously. Some ten years later, 27 independent parallelization projects were under development [11]. Today, with MATLAB's Parallel Computing Toolbox$^{\text{TM}}$having been introduced recently, solutions to computationally and data intensive problems can be obtained by multicore processors, computer clusters, or through the use of a GPU. MATLAB offers a number of high-level constructs for that. Parallel for-loops, invoked by the so-called `parfor` statement, specialized array types, and numerical algorithms prepared as being parallelized by design add to these. MPI calls or invocations are not necessary for using elements of this toolbox.

Said tool collection supports the use of up to eight MATLAB computational cores, the so-called workers. These workers execute parts of applications locally on a multi-CPU or multicore machine. For an exemplary dual quad-core computer, a performance increase of more than 500% can be expected [21]. In case even more massive parallelization becomes necessary, the same code can also be run on a computer cluster or grid computing service.

Appendix A

Listings

A.1 Input XML Schema Definition

Listing A.1: Full XML schema definition of the input data compound

```
ï»¿<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema elementFormDefault="qualified"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="ModelCollection" type="ModelCollection" />
    <xsd:complexType name="ModelCollection">
        <xsd:sequence>
            <xsd:sequence minOccurs="1" maxOccurs="unbounded">
                <xsd:element name="Model">
                    <xsd:complexType>
                        <xsd:complexContent>
                            <xsd:extension base="Model" />
                        </xsd:complexContent>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
            <xsd:element name="Output" type="Output" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Model">
        <xsd:choice>
            <xsd:sequence>
                <xsd:element name="MacroElements"
                    type="MacroElements" />
            </xsd:sequence>
            <xsd:element name="Input" type="Input" />
            <xsd:element name="Code" type="CodeType" />
        </xsd:choice>
        <xsd:attribute name="type" type="ModelType" use="required"
            />
        <xsd:attribute name="geometry" type="GeometryType"
            use="required" />
        <xsd:attribute name="solver" type="SolverType" />
        <xsd:attribute name="name" type="xsd:string" />
    </xsd:complexType>
    <xsd:complexType name="MacroElements">
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="MacroElement" type="MacroElement" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="MacroElement">
        <xsd:sequence>
            <xsd:element name="Geometry" type="Geometry"
                minOccurs="1" maxOccurs="1" />
            <xsd:element name="BoundaryConditions"
                type="BoundaryConditions" minOccurs="0"
                maxOccurs="1" />
            <xsd:element name="Material" type="Material"
                minOccurs="0" maxOccurs="1" />
            <xsd:element name="Source" type="Source" minOccurs="0"
                maxOccurs="1" />
        </xsd:sequence>
        <xsd:attribute name="refinement" type="xsd:string"
            use="optional" />
        <xsd:attribute name="edgerefinement" type="xsd:string"
            use="optional" />
        <xsd:attribute name="name" type="xsd:string"
            use="optional" />
    </xsd:complexType>
    <xsd:complexType name="Geometry">
        <xsd:choice minOccurs="1" maxOccurs="1">
            <xsd:element name="Add" type="Add" minOccurs="0"
                maxOccurs="1" />
            <xsd:element name="Subtract" type="Subtract"
                minOccurs="0" maxOccurs="1" />
            <xsd:element name="Intersect" type="Intersect"
                minOccurs="0" maxOccurs="1" />
            <xsd:element name="Polygon" type="Polygon" />
        </xsd:choice>
    </xsd:complexType>
    <xsd:complexType name="Add">
```

```
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
            <xsd:choice>
                <xsd:element name="Polygon" type="Polygon" />
                <xsd:element name="Add" type="Add" />
                <xsd:element name="Subtract" type="Subtract" />
                <xsd:element name="Intersect" type="Intersect" />
            </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Subtract">
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
            <xsd:choice>
                <xsd:element name="Polygon" type="Polygon" />
                <xsd:element name="Add" type="Add" />
                <xsd:element name="Subtract" type="Subtract" />
                <xsd:element name="Intersect" type="Intersect" />
            </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Polygon">
        <xsd:sequence>
            <xsd:element name="Coords" type="Coords" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Coords">
        <xsd:sequence minOccurs="3" maxOccurs="unbounded">
            <xsd:element name="Coord" type="Coord" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Coord">
        <xsd:attribute name="x" type="xsd:string" use="required" />
        <xsd:attribute name="y" type="xsd:string" use="required" />
    </xsd:complexType>
    <xsd:complexType name="BoundaryConditions">
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="BoundaryCondition"
                type="BoundaryCondition" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="BoundaryCondition">
        <xsd:sequence>
            <xsd:element name="Value" type="xsd:string"
                minOccurs="1" maxOccurs="1" />
            <xsd:element name="Transition" type="xsd:string"
                minOccurs="0" maxOccurs="1" />
        </xsd:sequence>
        <xsd:attribute name="type" type="BoundaryConditionType"
           use="required" />
        <xsd:attribute name="from" type="xsd:positiveInteger"
           use="required" />
        <xsd:attribute name="to" type="xsd:positiveInteger"
           use="required" />
    </xsd:complexType>
    <xsd:simpleType name="BoundaryConditionType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="Dirichlet" />
            <xsd:enumeration value="Neumann" />
            <xsd:enumeration value="Cauchy" />
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:complexType name="Material">
        <xsd:choice>
            <xsd:element name="Value" type="xsd:string" />
            <xsd:element name="Import" type="Import" />
        </xsd:choice>
    </xsd:complexType>
    <xsd:complexType name="Source">
        <xsd:choice>
            <xsd:element name="Value" type="xsd:string" />
            <xsd:element name="Import" type="Import" />
        </xsd:choice>
    </xsd:complexType>
    <xsd:complexType name="Intersect">
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
            <xsd:choice>
                <xsd:element name="Polygon" type="Polygon" />
                <xsd:element name="Add" type="Add" />
                <xsd:element name="Subtract" type="Subtract" />
                <xsd:element name="Intersect" type="Intersect" />
            </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:simpleType name="ModelType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="electrostatic" />
            <xsd:enumeration value="currentflow" />
            <xsd:enumeration value="magnetostatic" />
            <xsd:enumeration value="thermal" />
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="GeometryType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="plane" />
            <xsd:enumeration value="axisymmetric" />
            <xsd:enumeration value="magnetostatic" />
            <xsd:enumeration value="thermal" />
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="SolverType">
        <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    <xsd:simpleType name="CodeType">
        <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    <xsd:simpleType name="ModelNameType">
        <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    <xsd:simpleType name="ImportTypeType">
        <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    <xsd:simpleType name="ThresholdType">
        <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    <xsd:simpleType name="InterpreterType">
        <xsd:restriction base="xsd:string" />
    </xsd:simpleType>
    <xsd:complexType name="Input">
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
            <xsd:element name="Value" type="xsd:string" />
            <xsd:element name="Import" type="Import" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Import">
        <xsd:sequence minOccurs="0" maxOccurs="1">
            <xsd:element name="CouplingCode">
                <xsd:complexType>
                    <xsd:simpleContent>
                        <xsd:extension base="CodeType">
                            <xsd:attribute name="interpreter"
                                type="InterpreterType" />
                        </xsd:extension>
                    </xsd:simpleContent>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="fromModel" type="ModelNameType"
            use="required" />
        <xsd:attribute name="type" type="ImportTypeType"
            use="required" />
        <xsd:attribute name="threshold" type="ThresholdType"
            use="required" />
    </xsd:complexType>
    <xsd:complexType name="Output">
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="query" type="QueryType" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="QueryType">
        <xsd:sequence>
            <xsd:element name="OutputCode">
                <xsd:complexType>
                    <xsd:simpleContent>
                        <xsd:extension base="CodeType">
                            <xsd:attribute name="interpreter"
                                type="InterpreterType" />
                        </xsd:extension>
                    </xsd:simpleContent>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="fromModel" type="ModelNameType" />
        <xsd:attribute name="type" type="ImportTypeType" />
    </xsd:complexType>
</xsd:schema>
```

A.2 Full Problem Setup: Electrothermal Hardening

Listing A.2: Full XML definition of the hardening device example

```
ï»¿<?xml version="1.0" encoding="utf-8"?>
<ModelCollection
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Model type="currentflow" geometry="plane" solver="FEMtastic">
    <MacroElements>
      <MacroElement>
        <Geometry>
          <Add>
            <Polygon>
              <Coords>
                <Coord x="0" y="0" />
                <Coord x="0.3" y="0" />
                <Coord x="0.5" y="0.3" />
                <Coord x="0.6" y="0.3" />
                <Coord x="0.6" y="0.4" />
                <Coord x="0.5" y="0.4" />
                <Coord x="0.3" y="0.7" />
                <Coord x="0" y="0.7" />
              </Coords>
            </Polygon>
          </Add>
        </Geometry>
        <BoundaryConditions>
          <BoundaryCondition type="Dirichlet" from="1" to="1">
            <Value>12</Value>
          </BoundaryCondition>
          <BoundaryCondition type="Dirichlet" from="7" to="8">
            <Value>12</Value>
          </BoundaryCondition>
        </BoundaryConditions>
        <Material>
          <Value>59.1E6</Value>
        </Material>
      </MacroElement>
      <MacroElement>
        <Geometry>
          <Add>
            <Polygon>
              <Coords>
                <Coord x="0.6" y="0.3" />
                <Coord x="1.8" y="0.3" />
                <Coord x="1.8" y="0.4" />
                <Coord x="0.6" y="0.4" />
              </Coords>
            </Polygon>
          </Add>
        </Geometry>
        <Material>
          <Value>420</Value>
        </Material>
      </MacroElement>
      <MacroElement>
        <Geometry>
          <Add>
            <Polygon>
              <Coords>
                <Coord x="2.1" y="0" />
                <Coord x="2.4" y="0" />
                <Coord x="2.4" y="0.7" />
                <Coord x="2.1" y="0.7" />
                <Coord x="1.9" y="0.4" />
                <Coord x="1.8" y="0.4" />
                <Coord x="1.8" y="0.3" />
                <Coord x="1.9" y="0.3" />
              </Coords>
            </Polygon>
          </Add>
        </Geometry>
        <BoundaryConditions>
          <BoundaryCondition type="Dirichlet" from="1" to="3">
            <Value>0.1</Value>
          </BoundaryCondition>
        </BoundaryConditions>
        <Material>
          <Value>59.1E6</Value>
        </Material>
      </MacroElement>
    </MacroElements>
  </Model>
  <Model type="thermal" refinement="1.5"  solver="FEMtastic">
    <MacroElements>
      <MacroElement>
        <Geometry>
          <Polygon>
```

```
          <Coords>
            <Coord x="0.6" y="0.3" />
            <Coord x="1.8" y="0.3" />
            <Coord x="1.8" y="0.4" />
            <Coord x="1.7" y="0.4" />
            <Coord x="0.7" y="0.4" />
            <Coord x="0.6" y="0.4" />
          </Coords>
        </Polygon>
      </Geometry>
      <BoundaryConditions>
        <BoundaryCondition type="Cauchy" to="3" from="3">
          <Value>300</Value>
          <Transition>5.6</Transition>
        </BoundaryCondition>
        <BoundaryCondition type="Cauchy" to="5" from="5">
          <Value>300</Value>
          <Transition>5.6</Transition>
        </BoundaryCondition>
        <BoundaryCondition type="Neumann" to="2" from="1">
          <Value>0</Value>
        </BoundaryCondition>
        <BoundaryCondition type="Neumann" to="6" from="6">
          <Value>0</Value>
        </BoundaryCondition>
      </BoundaryConditions>
      <Material>
        <Value>85</Value>
      </Material>
      <Source>
        <Import>
                        <FromModel>1</FromModel>
                        <Type>current-thermal</Type>
                        <Threshold>1%</Threshold>
                </Import>
      </Source>
    </MacroElement>
    <MacroElement>
      <Geometry>
        <Polygon>
          <Coords>
            <Coord x="0.7" y="0.4" />
            <Coord x="0.7" y="0.6"></Coord>
            <Coord x="0.8" y="0.6"></Coord>
            <Coord x="0.9" y="0.8" />
            <Coord x="1.5" y="0.8" />
            <Coord x="1.6" y="0.6" />
            <Coord x="1.7" y="0.6" />
            <Coord x="1.7" y="0.4"></Coord>
          </Coords>
        </Polygon>
      </Geometry>
      <BoundaryConditions>
        <BoundaryCondition type="Cauchy" to="7" from="1">
          <Value>300</Value>
          <Transition>5.6</Transition>
        </BoundaryCondition>
      </BoundaryConditions>
      <Material>
        <Value>42</Value>
      </Material>
    </MacroElement>
   </MacroElements>
  </Model>
 </ModelCollection>
```

A.3 Full Problem Setup: TEAM22 Benchmark Problem

Listing A.3: Full XML definition of the TEAM22 Benchmark Problem Example

```
ï»¿<?xml version="1.0" encoding="utf-8"?>
<ModelCollection
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Model type="magnetostatics" symmetry="rotational"
   refinement="1.5" solver="FEMtastic">
    <MacroElements>
      <MacroElement>
        <Geometry>
          <Add>
            <Polygon>
              <Coords>
                <Coord x="1.865" y="-0.8" />
                <Coord x="2.135" y="-0.8" />
                <Coord x="2.135" y="0.8" />
                <Coord x="1.865" y="0.8" />
              </Coords>
            </Polygon>
          </Add>
        </Geometry>
                <Source>
                  22.5E6
                </Source>
      </MacroElement>
      <MacroElement>
        <Geometry>
          <Add>
            <Polygon>
              <Coords>
                <Coord x="2.9" y="-0.65" />
                <Coord x="3.1" y="-0.65" />
                <Coord x="3.1" y="0.65" />
                <Coord x="2.9" y="0.65" />
              </Coords>
            </Polygon>
          </Add>
        </Geometry>
                <Source>
                  -22.5E6
                </Source>
        </MacroElement>
      </MacroElements>
    </Model>
    <Model type="analytic" refinement="" solver="Python">
          <Input>
            <Value>22.5E6</Value>
            <Value>-22.5E6</Value>
            <Import>
                  <FromModel>1</FromModel>
                  <Type>maxFromRegion</Type>
                  <Threshold></Threshold>
            </Import>
            <Import>
                  <FromModel>1</FromModel>
                  <Type>maxFromRegion</Type>
                  <Threshold></Threshold>
            </Import>
          </Input>
          <Code><![CDATA[
# input = [B1, B2, J1, J2]

J1=(-6.4*input[0]+54)
J2=(-6.4*input[1]+54)

# validates whether quench condition is fulfilled (1) or not (0,
    error case)
if (J1 > input[2]) or (J2 > input[3]):
    retVal = 1
else:
    retVal = 0

returnValues = []
returnValues.append(retVal)]]>
          </Code>
    </Model>
</ModelCollection>
```

A.4 Full Problem Setup: Electromagnetic Actuator

Listing A.4: Full XML definition of the Electromagnetic Actuator Problem Example

```
ï»¿<?xml version="1.0" encoding="utf-8"?>
<ModelCollection xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Model type="currentflow" geometry="plane" solver="FEMtastic">
    <MacroElements>
      <MacroElement>
        <Geometry>
          <Add>
            <Polygon>
              <Coords>
                <Coord x="0" y="0" />
                <Coord x="0.3" y="0" />
                <Coord x="0.5" y="0.3" />
                <Coord x="0.6" y="0.3" />
                <Coord x="0.6" y="0.4" />
                <Coord x="0.5" y="0.4" />
                <Coord x="0.3" y="0.7" />
                <Coord x="0" y="0.7" />
              </Coords>
            </Polygon>
          </Add>
        </Geometry>
        <BoundaryConditions>
          <BoundaryCondition type="Dirichlet" from="1" to="1">
            <Value>12</Value>
          </BoundaryCondition>
          <BoundaryCondition type="Dirichlet" from="7" to="8">
            <Value>12</Value>
          </BoundaryCondition>
        </BoundaryConditions>
        <Material>
          <Value>59.1E6</Value>
        </Material>
      </MacroElement>
      <MacroElement>
        <Geometry>
          <Add>
            <Polygon>
              <Coords>
                <Coord x="0.6" y="0.3" />
                <Coord x="1.8" y="0.3" />
                <Coord x="1.8" y="0.4" />
                <Coord x="0.6" y="0.4" />
              </Coords>
            </Polygon>
          </Add>
        </Geometry>
        <Material>
          <Value>420</Value>
        </Material>
      </MacroElement>
      <MacroElement>
        <Geometry>
          <Add>
            <Polygon>
              <Coords>
                <Coord x="2.1" y="0" />
                <Coord x="2.4" y="0" />
                <Coord x="2.4" y="0.7" />
                <Coord x="2.1" y="0.7" />
                <Coord x="1.9" y="0.4" />
                <Coord x="1.8" y="0.4" />
                <Coord x="1.8" y="0.3" />
                <Coord x="1.9" y="0.3" />
              </Coords>
            </Polygon>
          </Add>
        </Geometry>
        <BoundaryConditions>
          <BoundaryCondition type="Dirichlet" from="1" to="3">
            <Value>0.1</Value>
          </BoundaryCondition>
        </BoundaryConditions>
        <Material>
          <Value>59.1E6</Value>
        </Material>
      </MacroElement>
    </MacroElements>
  </Model>
  <Model type="thermal" refinement="1.5" solver="FEMtastic">
    <MacroElements>
      <MacroElement>
        <Geometry>
          <Polygon>
            <Coords>
              <Coord x="0.6" y="0.3" />
              <Coord x="1.8" y="0.3" />
              <Coord x="1.8" y="0.4" />
              <Coord x="1.7" y="0.4" />
              <Coord x="0.7" y="0.4" />
              <Coord x="0.6" y="0.4" />
            </Coords>
          </Polygon>
        </Geometry>
        <BoundaryConditions>
          <BoundaryCondition type="Cauchy" to="3" from="3">
            <Value>300</Value>
            <Transition>5.6</Transition>
          </BoundaryCondition>
          <BoundaryCondition type="Cauchy" to="5" from="5">
            <Value>300</Value>
            <Transition>5.6</Transition>
          </BoundaryCondition>
          <BoundaryCondition type="Neumann" to="2" from="1">
            <Value>0</Value>
          </BoundaryCondition>
          <BoundaryCondition type="Neumann" to="6" from="6">
            <Value>0</Value>
          </BoundaryCondition>
        </BoundaryConditions>
        <Material>
          <Value>85</Value>
        </Material>
        <Source>
          <Import>
                        <FromModel>1</FromModel>
                        <Type>current-thermal</Type>
                        <Threshold>1%</Threshold>
                  </Import>
        </Source>
      </MacroElement>
      <MacroElement>
        <Geometry>
          <Polygon>
            <Coords>
              <Coord x="0.7" y="0.4" />
              <Coord x="0.7" y="0.6"></Coord>
              <Coord x="0.8" y="0.6"></Coord>
              <Coord x="0.9" y="0.8" />
```

```
              <Coord x="1.5" y="0.8" />
              <Coord x="1.6" y="0.6" />
              <Coord x="1.7" y="0.6" />
              <Coord x="1.7" y="0.4"></Coord>
            </Coords>
          </Polygon>
        </Geometry>
        <BoundaryConditions>
          <BoundaryCondition type="Cauchy" to="7" from="1">
            <Value>300</Value>
            <Transition>5.6</Transition>
          </BoundaryCondition>
        </BoundaryConditions>
        <Material>
          <Value>42</Value>
        </Material>
      </MacroElement>
    </MacroElements>
  </Model>
</ModelCollection>
```

Bibliography

[1] Alotto, P. (2009). *Data Structures for mesh-based electromagnetic simulation codes*. In Proceedings of the 8th International Symposium on Electric and Magnetic Fields (EMF 2009), Mondovi, Italy (2009). 8, 9

[2] Alotto, P., Baumgartner, U., Freschi, F., Jaindl, M., Kostinger, A., Magele, C., Renhart, W. and Repetto, M. (2008). *SMES Optimization Benchmark Extended: Introducing Pareto Optimal Solutions Into TEAM22*. Magnetics, IEEE Transactions on, 44(6):1066 –1069. 14, 70

[3] Alotto, P., Kuntsevitch, A., Magele, C., Molinari, G., Paul, C., Preis, K., Repetto, M. and Richter, K. (1996). *Multiobjective optimization in magnetostatics: a proposal for benchmark problems*. Magnetics, IEEE Transactions on, 32(3):1238 –1241. 70

[4] Alotto, P., Molfino, P. and Molinari, G. (2001). *A WWW-based tool for the remote optimization of electromagnetic devices*. Magnetics, IEEE Transactions on, 37(5):3592 –3595. 43

[5] Andrews, K. (2004). *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*, Graz University of Technology, Austria. `ftp://ftp2.iicm.edu/pub/keith/thesis/thesis.zip`. ii

[6] Baldwin, C. Y. and Clark, K. B. (1999). *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA. 0262024667. 17

[7] Ballard, D. (2006). Primes results for x86 vs. PPC vs. Arm. 86

[8] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Eliot, J., Moss, B., Phansalkar, A., Stefanovic;, D., VanDrunen, T., von Dincklage, D. and Wiedermann, B. (2006). *The DaCapo benchmarks: java benchmarking development and analysis*. In OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 169–190, New York, NY, USA (2006). ACM Press. 13

[9] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. and Yergeau, F. (2006). *Extensible markup language (XML) 1.0, Fourth Edition.* World Wide Web Consortium, Recommendation REC-xml-20060816. 8, 15, 17

[10] Bull, J. M., Smith, L. A., Pottage, L. and Freeman, R. (2001). *Benchmarking Java against C and Fortran for scientific applications.* In JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, pages 97–105, New York, NY, USA (2001). ACM Press. 13

[11] Choy, R. and Edelman, A. (2005). *Parallel MATLAB: Doing it Right.* Proceedings of the IEEE, 93(2):331 –341. 95

[12] Cignoni, P., Montani, C. and Scopigno, R. (1998). *A comparison of mesh simplification algorithms.* Computers & Graphics, 22(1):37–54. 49

[13] Costa, M., Coulomb, J.-L. and Marechal, Y. (2000). *An object-oriented optimization library for finite element method software.* Magnetics, IEEE Transactions on, 36(4): 1057 –1060. 8

[14] Cranganu-Cretu, B., Smajic, J., Köstinger, A., Jaindl, M., Renhart, W. and Magele, C. (2008). *Enhancement Procedure of Magnetic Shunting/Shielding Topologies in Transformer-like Configurations.* In Proceedings 13th Biennial IEEE Conference on Electromagnetic Field Computation (CEFC 2008), pages 581–586, National Technical University of Athens, Greece (2008). 14

[15] di Barba, P. (2009). *Multiobjective Shape Design in Electricity and Magnetism (Lecture Notes in Electrical Engineering).* Springer, London. 9048130794.

[16] di Barba, P. and Savini, A. (2010). *Optimal shape design with field subdomains modelled by Thevenin equivalent conditions.* In Proceedings of the XI-th International Workshop on Optimization and Inverse Problems in Electromagnetism OIPE 2010, Sofia, Bulgaria (2010).

[17] ECPD (1941). *The Engineers' Council for Professional Development.* Science, 94: 456–+. 1

[18] Engwirda, D. (2005). Unstructured Mesh Methods for the Navier-Stokes Equations. Master's thesis, School of Aerospace Engineering, The University of Sydney. 28

[19] Engwirda, D. (2010). *MESH2D - Automatic Mesh Generation.* MATLAB Central File Exchange. 28

[20] Folk, M. and Pourmal, E. (2010). *Balancing Performance and Preservation – Lessons Learned with HDF5.* In Digital Preservation Interoperability Framework (DPIF) Workshop, Gaithersburg, Maryland, USA (2010). National Institute of Standards and Technology (NIST). vii, 10, 11, 37

[21] Goryawala, M., Guillen, M., Bhatt, R., Mcgoron, A. and Adjouadi, M. (2010). *A Comparative Study on the Performance of the Parallel and Distributing Computing Operation in MatLab.* In Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on, pages 150 –157. 95

[22] Gosink, L., Shalf, J., Stockinger, K., Wu, K. and Bethel, W. (2006). *HDF5-FastQuery: Accelerating Complex Queries on HDF Datasets using Fast Bitmap Indices.* In Scientific and Statistical Database Management, 2006. 18th International Conference on, pages 149 –158. 9

[23] Gropp, W., Lusk, E. and Skjellum, A. (1999). *Using MPI: portable parallel programming with the message passing interface.* 95

[24] IGTE (1978-2010). *Computer program package* EleFAnT2D*, IGTE TUGraz.* `http://www.elefants.at/`. 2, 45, 75

[25] IGTE (2010). *Institute for Fundamentals and Theory in Electrical Engineering.* `http://www.igte.tugraz.at/`. 2, 14

[26] Jaindl, M., Köstinger, A., Magele, C. and Renhart, W. (2007). *Optimal design of a disk type magneto-rheologic fluid clutch.* e&i Elektrotechnik und Informationstechnik, 124: 266–272. 10.1007/s00502-007-0453-4. 14

[27] Jaindl, M. (2005). Conference Management Support. Master's thesis, Graz University of Technology, Department for Fundamentals and Theory in Electrical Engineering.

[28] Jaindl, M., Baumgartner, U., Grumer, M., Köstinger, A., Magele, C., Preis, K., Reinbacher, M. and Voller, S. (2003). *e-Courseware authoring tools for teaching electrodynamics.* COMPEL: Int J for Computation and Maths. in Electrical and Electronic Eng., 22(3):603–615. 14

[29] Jaindl, M., Reinbacher-Köstinger, A., Kutschera, R. and Magele, C. (2010). *Numerical Optimization Framework for Weakly Coupled Multiphysical Problems.* In Proceedings of the 14th International IGTE Symposium on Numerical Field Calculation in Electrical Engineering, Graz, Austria (2010). 59

[30] Jaindl, M., Reinbacher-Köstinger, A., Kutschera, R., Magele, C. and Renhart, W. (2009). *Optimization of Weakly coupled Multiphysical Problems using Object Oriented Programming.* In Proceedings of the 17th Conference on the Computation of Electromagnetic Fields, volume 17, pages 430 – 431, Florianópolis, Brazil (2009). 59

[31] Jaindl, M., Reinbacher-Köstinger, A., Magele, C. and Renhart, W. (2009). *Multi-Objective Optimization Using Evolution Strategies.* Facta universitatis / Series electronics and energetics, 22(2):159 – 174. 14

[32] Jiao, X., Campbell, M. T. and Heath, M. T. (2003). *Roccom: an object-oriented, data-centric software integration framework for multiphysics simulations*. In Proceedings of the 17th annual international conference on Supercomputing, ICS '03, pages 358–368, New York, NY, USA (2003). ACM. 4, 5, 7, 94

[33] Jung, Michael; Langer, Ulrich (2001). *Methode der finiten Elemente für Ingenieure*. Teubner. 3-519-02973-1. 25, 33

[34] Kanerva, S. and Arkkio, A. (2006). *Extraction of circuit parameters from time stepping FEM computation for coupled field-circuit simulation*. COMPEL: The International Journal for Computation and Mathematics in Electrical and Electronic Engineering, 25 (2):334 – 343. 19

[35] Kennicott, P. (1995). *The initial graphics exchange specification (IGES) Version 5.3*. IGES/PDES Organisation. 94

[36] Knuth, D. E. (1998). *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA. 0-201-89685-0. 9

[37] Kutschera, R. (2010). Numerical simulation framework for multiphysical problems by the use of the finite element method. Master's project, Graz University of Technology, Department for Fundamentals and Theory in Electrical Engineering. ii, 12, 22, 31, 73, 75, 86

[38] Lowther, D., Silvester, P., Freeman, E., Rea, K., Trowbridge, C., Newman, M. and Simkin, J. (1981). *Ruthless–A general purpose finite element post-processor*. Magnetics, IEEE Transactions on, 17(6):3402 – 3404. 55

[39] Magele, C., Köstinger, A., Jaindl, M., Renhart, W., Cranganu-Cretu, B. and Smajic, J. (2010). *Niching Evolution Strategies for Simultaneously Finding Global and Pareto Optimal Solutions*. Magnetics, IEEE Transactions on, 46(8):2743 –2746. 14

[40] Magele, C., Preis, K., Renhart, W., Dyczij-Edlinger, R. and Richter, K. (1993). *Higher order evolution strategies for the global optimization of electromagnetic devices*. Magnetics, IEEE Transactions on, 29(2):1775 –1778. 70

[41] Magele, C., Fürntratt, G., Bardi, I., Richter, K. R., Schönwetter, G., Alotto, P., Molinari, G. and Repetto, M. (1995). *SMES Optimization Benchmark - TEAM Workshop Problem 22*. International Compumag Society. 70

[42] Magele, C., Jaindl, M., Köstinger, A., Renhart, W., Cranganu-Cretu, B. and Smajic, J. (2010). *Niching evolution strategy finding global and Pareto optimal solutions*. COMPEL: Int J for Computation and Maths. in Electrical and Electronic Eng., 29(6):1514 – 1523.

[43] Mai, W. and Henneberger, G. (2000). *Object-oriented design of finite element calculations with respect to coupled problems*. Magnetics, IEEE Transactions on, 36(4):1677 –1681. 18

[44] Martínez, F., Rueda, A. J. and Feito, F. R. (2009). *A new algorithm for computing Boolean operations on polygons*. Computers and Geosciences, 35(6):1177–1185. 28, 46

[45] Mattsson, M., Bosch, J. and Fayad, M. E. (1999). *Framework integration problems, causes, solutions*. Commun. ACM, 42:80–87. v, 6

[46] McGarrity, S. (2008). *Introduction to Object-Oriented Programming in MATLAB®*. Matlab Digest. 13

[47] Meunier, G., Coulomb, J. and Savalle, D. (1988). *Use of formulae in finite element post processing*. Magnetics, IEEE Transactions on, 24(1):389 –392. 54

[48] Moler, C. (1995). *Why there isn't a parallel MATLAB*. Matlab News and Notes, Spring 1995:12 ff. 95

[49] Moro, M. M., Braganholo, V., Dorneles, C. F., Duarte, D., Galante, R. and Mello, R. S. (2009). *XML: some papers in a haystack*. SIGMOD Rec., 38:29–34. 15

[50] Nicolet, A., Delince, F., Bamps, N., Genon, A. and Legros, W. (1993). *A coupling between electric circuits and 2D magnetic field modeling*. Magnetics, IEEE Transactions on, 29(2):1697 –1700. 19

[51] Pacheco, P. (1997). *Parallel programming with MPI*. Morgan Kaufmann. 1558603395. 94

[52] Parnas, D. L. (1972). *On the criteria to be used in decomposing systems into modules*. Commun. ACM, 15:1053–1058. 5, 17

[53] Persson, P.-O. and Strang, G. (2004). *A Simple Mesh Generator in MATLAB*. SIAM Review, 46:2004. 28

[54] Poinot, M. (2010). *Five Good Reasons to Use the Hierarchical Data Format*. Computing in Science and Engineering, 12:84–90. 8, 9

[55] Python Software Foundation (2011). *Python Programming Language*. `http://www.python.org/`. 12

[56] Reinauer, V., Wendland, T., Scheiblich, C., Banucu, R. and Rucker, W. (2010). *Object-oriented development and runtime investigation of 3-D electrostatic FEM problems in pure Java*. In Electromagnetic Field Computation (CEFC), 2010 14th Biennial IEEE Conference on, page 1. 45, 90

[57] Reinbacher-Köstinger, A., Weinhappl, A., Magele, C. and Jaindl, M. (2010). *Web Application for Multi-Objective Optimization Benchmark Problems*. In Proceedings 14th International IGTE Symposium on Numerical Field Calculation in Electrical Engineering IGTE2010, Graz, Austria (2010). 70

[58] Reselman, B. (1998). *Using Visual Basic 6*. Using... Series. Que. 9780789716330. 11

[59] Sanner, M. F. (1999). *Python: A Programming Language for Software Integration and Development*. J. Mol. Graphics Mod., 17:57–61. 11, 35

[60] Santini, E. and Silvester, P. (1996). *Thevenin equivalent fields*. Magnetics, IEEE Transactions on, 32(3):1409 –1412.

[61] Schönwetter, G., Magele, C., Preis, K., Paul, C., Renhart, W. and Richter, K. (1995). *Optimization of SMES solenoids with regard to their stray fields*. Magnetics, IEEE Transactions on, 31(3):1940 –1943. 70

[62] Shasharina, S., Li, C., Wang, N., Pundaleeka, R. and Wade-Stein, D. (2007). *Distributed Technologies for Remote Access of HDF Data*. In Enabling Technologies: Infrastructure for Collaborative Enterprises, 2007. WETICE 2007. 16th IEEE International Workshops on, pages 255 –260. 9

[63] Smajic, J., Cranganu-Cretu, B., Kostinger, A., Jaindl, M., Renhart, W. and Magele, C. (2009). *Optimization of Shielding Devices for Eddy-Currents Using Multiobjective Optimization Methods*. Magnetics, IEEE Transactions on, 45(3):1550 –1553. 14

[64] Smith, B. and Wellington, J. (1984). *IGES- A key to CAD/CAM systems integration*. CAD/CAM Technology, 3:33–35. 94

[65] Sullivan, K. J., Griswold, W. G., Cai, Y. and Hallen, B. (2001). *The structure and value of modularity in software design*. SIGSOFT Softw. Eng. Notes, 26:99–108. 5, 17

[66] Sur, S., Koop, M. J. and Panda, D. K. (2006). *High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis*. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06, New York, NY, USA (2006). ACM. 95

[67] The Engineering Toolbox (2010). `http://www.engineeringtoolbox.com`. 67

[68] The MathWorks, Inc. (2010). Latest features in Matlab. `http://www.mathworks.de/products/matlab/whatsnew.html`, [Online; accessed, 16-October-2010]. 12

[69] Tuinenga, P. (1991). *SPICE: a guide to circuit simulation and analysis using PSpice*. Prentice Hall PTR Upper Saddle River, NJ, USA. 0137472706. 36

[70] Unpingco, J. (2008). *Some Comparative Benchmarks for Linear Algebra Computations in Matlab and Scientific Python*. In Proceedings of the 2008 DoD HPCMP Users Group Conference, pages 503–505, Washington, DC, USA (2008). IEEE Computer Society. 86

[71] van der Velde, P. and Mallinson, G. D. (2007). *The design of a component-oriented framework for numerical simulation software*. Adv. Eng. Softw., 38:182–192. vii, 5

[72] van Rossum, G. (1993). *An Introduction to Python for Unix/C Programmers*. In Proceedings of the NLUUG najaarsconferentie. Dutch UNIX users group. 12, 35, 73

[73] Wen, T., Su, J., Colella, P., Yelick, K. and Keen, N. (2007). *An adaptive mesh refinement benchmark for modern parallel programming languages*. In Proceedings of the 2007 ACM/IEEE conference on Supercomputing, SC '07, pages 40:1–40:12, New York, NY, USA (2007). ACM. 22

[74] Wilde, E. and Glushko, R. J. (2008). *XML fever*. Commun. ACM, 51:40–46. 9, 17

[75] Zeng, J., Bai, B. and Wang, X. (2005). *A new method of organizing the result data of 3D electromagnetic field analysis based on XML technology*. In Electrical Machines and Systems, 2005. ICEMS 2005. Proceedings of the Eighth International Conference on, volume 3, pages 2133 – 2135 Vol. 3. 8

[76] Zeng, P., Hao, Y., Shao, W. and Liu, Y. (2008). *Towards a Software Integration Framework in Product Collaborative Design Environment*. In Computer Science and Software Engineering, 2008 International Conference on, volume 2, pages 527 –530. 4