Numerical simulation framework for multiphysical problems by the use of the finite element method

Ralph Kutschera

Numerical simulation framework for multiphysical problems by the use of the finite element method

Thesis

at

Graz University of Technology

submitted by

Ralph Kutschera

Institute for Fundamentals and Theory in Electrical Engineering (IGTE),
Graz University of Technology
A-8010 Graz, Austria

October 26th, 2010

Advisor: Ao.Univ.-Prof. Dipl.Ing. Dr.techn. Christian Magele

Simulationsumgebung zur numerischen Behandlung multiphysikalischer Probleme unter Verwendung der Methode der Finiten Elemente

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Ralph Kutschera

Institut für Grundlagen und Theorie der Elektrotechnik (IGTE),
Technische Universität Graz
A-8010 Graz

26. Oktober 2010

Diese Arbeit ist in englischer Sprache verfasst.

Betreuer: Ao.Univ.-Prof. Dipl.Ing. Dr.techn. Christian Magele

Abstract

The finite element method (FEM) can be seen as the crowning discipline of all simulation techniques used nowadays. Own software packages have been implemented at the Institute for Fundamentals and Theory in Electrical Engineering that realize this method and apply it to assignments in an educational and industrial environment.
Recent researches on optimization strategies come with the need of a more flexible framework that makes the process of automation more manageable and independent from user intervention.
This thesis introduces a new simulation framework with a modern object-oriented design, specified user interfaces and the aspect of extensibility. It does electrostatic, magnetostatic, static current flow and thermal model computations and allows multiphysics by coupling these models. Furthermore, the user is supported in the process of modeling using an up-to-date algorithm that computes boolean operations on polygons and therefor helps on setting up a geometry by defining primitives and their relations.

Kurzfassung

Unter allen Simulationstechniken, welche heutzutage weltweit Anwendung finden, ist wohl die Methode der Finiten Elemente (kurz FEM) die bekannteste. Das Institut für Grundlagen und Theorie der Elektrotechnik verwendet seine eigenen Softwarepakete sowohl im industriellen wie auch im akademischen Umfeld zur Berechnung gegebener Aufgaben.
Fortschreitende Forschungsergebnisse im Bereich von Optimierungsstrategien machen nun jedoch den Einsatz von flexibleren Berechnungsanwendungen nötig, welche eine bessere Automatisierung ohne fortlaufende Benutzerintervention ermöglichen. Diese Diplomarbeit stellt eine neue Simulationsumgebung vor, welcher ein modernes objektorientiertes Design zugrunde liegt, deren Schnittstellen klar definiert sind und die mit dem Bedacht auf Erweiterbarkeit umgesetzt worden ist. Es unterstützt Berechnungen in den Bereichen Elektrostatik, Magnetostatik, statisches Strömungsfeld und thermische Problemstellungen. Multiphysikalische Modelle können durch Kopplung simuliert werden. Weiters wird der Anwender beim Modellierungsprozess unterstützt, indem ein aktueller Algorithmus, welcher Mengenoperationen auf Polygone berechnet, integriert wurde. Dieser erlaubt den schnellen Aufbau einer Geometrie durch Definition einfacher Primitive und deren Zusammanhänge.

Eidesstattliche Erklärung

Ich versichere hiermit, diese Arbeit selbständig verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfsmittel bedient zu haben.

I hereby certify that the work presented in this thesis is my own and that work performed by others is appropriately cited.

Ralph Kutschera

Danksagung

Ein großer Dank gilt meinen Eltern Eva und Manfred, meinen Großeltern Luise und Gerhard sowie Amalia und Franz.
Ihr wart es, die mich zur Umsetzung meines bislang größten Lebensprojektes ermutigt und mich bis zur Fertigstellung in jeder Gegebenheit unterstützt und gefördert habt. Einen herzlichen Dank will ich Euch dafür aussprechen!

Mein Dank gilt weiters meinen Betreuern Christian Magele und Michl Jaindl vom Institut für Grundlagen und Theorie der Elektrotechnik an der Technischen Universität Graz.
Für die Ermöglichung dieser Diplomarbeit, deren erste Züge bereits in der Bakkalaureatsarbeit vor wenigen Jahren enstanden, und für die großzügige Unterstützung bei der Umsetzung der vorliegenden Arbeit möchte ich Euch danken!

Auch allen weiteren Mitarbeitern des IGTE danke ich für spannende und humorvolle Stunden im Umfeld eines äußerst interessanten Themengebietes.

Ein großes Anliegen ist mir die Erwähnung aller Freunde vom Hafnerriegel Studentenheim.
Für eine grandiose Zeit in den ersten Jahren meines Studiums in Graz, für unzählige ausgelassene Stunden in der CAF und im sonstigen Umfeld des Heimes bin ich Euch, im Besonderen allen Mitbewohnern von C14, von Herzen dankbar!

Weiters will ich keinen meiner Freunde beim 1. Österreichischen HAPKIDO Verein Graz unerwähnt lassen.
Gemeinsam mit Euch hab ich gerne den schweißtreibenden, nötigen Ausgleich zu meinem Studienalltag gefunden. Danke!

Meinen Freunden im entfernten Wien gelten diese Zeilen.
Danke, dass Ihr trotz der Entfernung nie den Kontakt abbrechen lasst!

Allen weiteren Freunden in Graz, in der oberösterreichischen Heimat, allen Arbeitskollegen der letzten Jahre und allen lieben Verwandten:

Daunk schen!

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Motivation

Within the last two decades, computer-aided design (CAD) has been integrated enormously into industrial manufacturing and therefor also into the educational sector. The *Institute for Fundamentals and Theory in Electrical Engineering (IGTE)* at *Graz University of Technology (TUG)* started with computational numerics in the early 1980s by designing and implementing its own software tools. The institute, therefor, is the optimal environment to become acquainted with up-to-date techniques that are used all over the world nowadays, not only in electrical engineering.
One of these techniques – namely the finite element method – can be seen as the crowning discipline of all simulation methods. It is used, as already mentioned, in electrical engineering, but rather comes from structural analysis in aeronautics and is also used in any other engineering fraction like thermodynamics, computational fluid dynamics and also in the financial sector.

What makes this thesis very attractive is the fact that it does not only deal with the application of FEM onto specific assignments but also gives a deep insight into the core of this method. The realization of it merges mathematics, algorithm design and software engineering to an interesting piece of work whose gift is a deep understanding of how physics can be brought into a computing machine.

1.1. Thesis definition

This thesis is part of a reaserch project at the Institute for Fundamentals and Theory in Electrical Engineering at University of Technology, Graz, Austria. The project team works on optimization strategies in electrical engineering. The current field of studies deals with optimizing electrical problems in two-dimensional space with regard to minimizing e.g. ohmic drop or magnetic losses of a given arrangement. Since the early 1980's the ***Ele**ctromagnetic **F**ield **An**alysis **T**ool**s*** - a software package in short **EleFAnTs** - have been developed at the institute. EleFAnTs does electromagnetic field numerics also by means of the finite element method. Though the software has been subject to a 25-year development period it has not emerged as the optimal tool for applying optimization strategies. Thus, a demand has come up for alternative attempts. One of them is subject of this thesis. Figure 1.1 shows the basic idea of the optimization strategy.

The point of origin is any given problem to be subject of optimization. An intention is that to the choice of optimization parameters only the sky should be a limit. Thus, boundary conditions, material

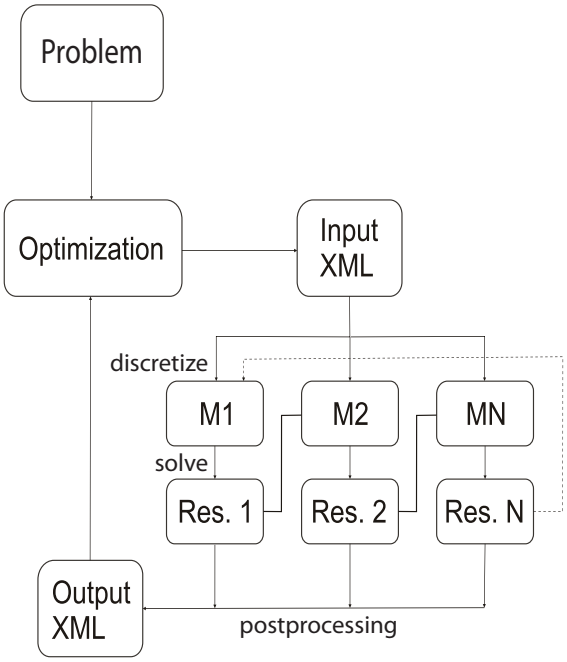

Figure 1.1.: Optimization of weakly coupled multiphysical problems

properties or geometry might be some of many unknowns.

As a problem definition schema, XML has been chosen as the interface between the user and the upcoming software. With the help of XML templates, problems are defined as shown in figure 1.2.

Based on an XML file the problem is discretized and solved. The solution might either be used for the next optimization step or can also be the source for a different physical model within the same optimization step.

Example: A static current flow model is set up in order to simulate a current flow within a heating plate. As a result, power dissipation acts as a thermal source for a secondary thermal model (Model 1, figure 1.1). This current flow model is base for a thermal temperature distribution simulation (Model 2, figure 1.1).

Illustration 1.3 shows a more detailed view to the project and the scope of this thesis highlighted with a rounded rectangle. All other parts including the one of the actual optimization strategy are carried out by other team members and can – due to its initial development state – not be mentioned any further. Starting point herein is a fully set up and validated XML file defining any electrostatic, magnetostatic, static current flow or thermal model coming from the controller or the model builder respectively (shown as box ”Model 1”). The model in a further step – depicted as box ”Discretization” – is discretized which means that the macro elements defined within the XML file will be meshed to a triangulation and the attributes (boundary conditions, material properties, ...) are applied to the resulting finite elements.
The finite element method itself will be generously discussed in this document. After the discretization, assembly is the next execution step and results in the FEM system of equations. The solver that evaluates the stiffness matrix and the load vector that set up the system of equations is made available by MATLAB and yields the solution to the defined model.
The postprocessor is finally a set of routines that offers data collection and visualization to the evalu-

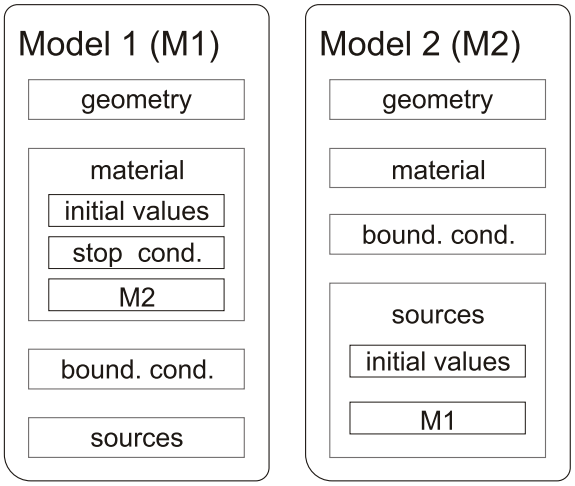

Figure 1.2.: Setting up a problem model

ator that either can initialize the next optimization step or set up a different physical model within the same optimization step.

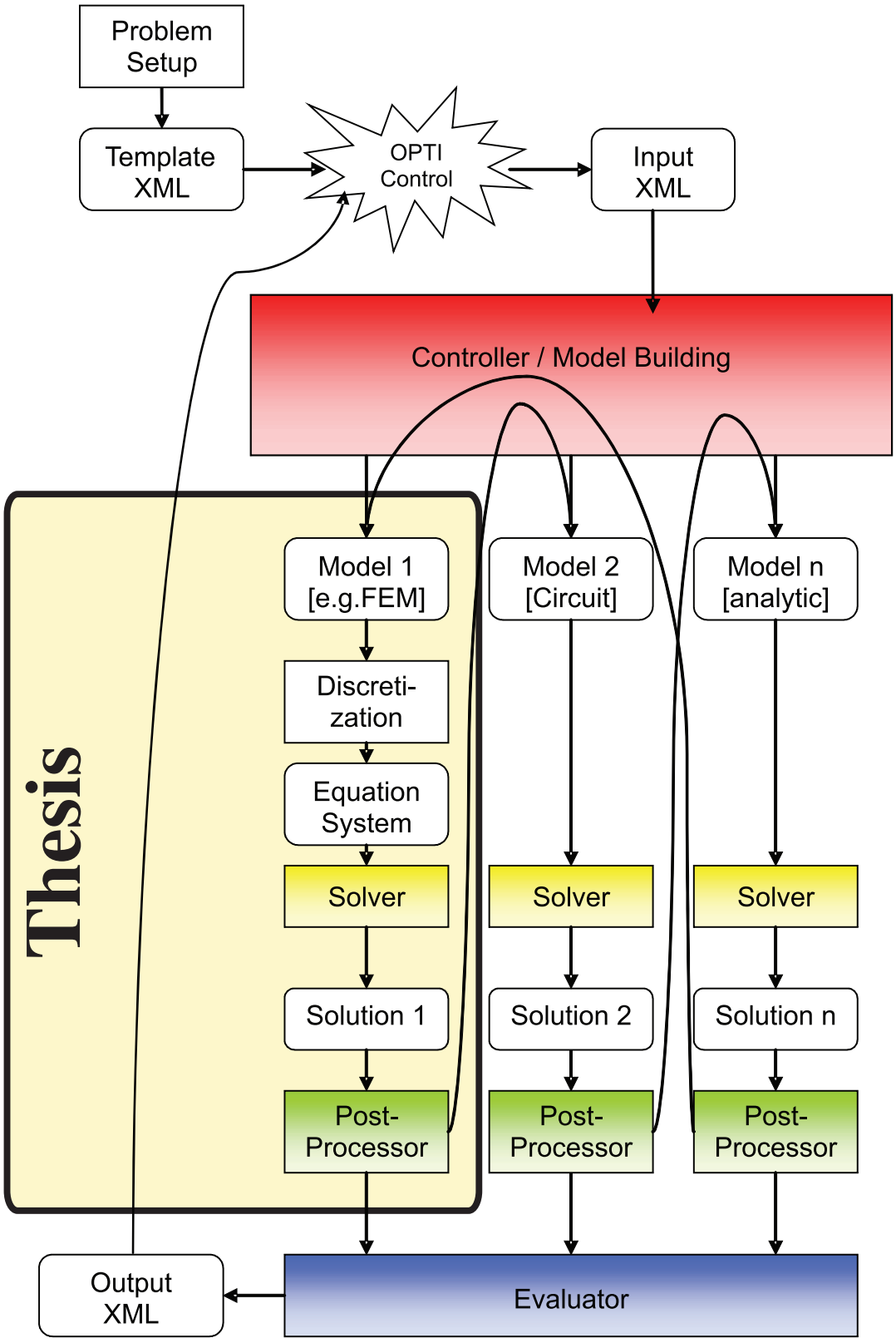

Figure 1.3.: A detailed view to the optimization model

Chapter 2

State-of-the-art

First attempts of the finite element method were done in the 1940s [14]. The method, as it is known today, was introduced by the end of the 50s [34]. Software engineering as a discipline had not yet emerged because computing power was available only at a small number of facilities at that time. NASA's NASTRAN was the first known finite element software but not publicly available. Other code came up from plenty of university departments with different approaches. Some of them were marketed and do still exist in a modified form [11]. Much of the code was written in procedure-oriented FORTRAN language which was fast but did not offer as many possibilities in design as does object-oriented programming nowadays.

Besides any software paradigms, approaching an engineering problem to be numerically examined, involves three basic steps that are common over any finite element software [9]. These are:

- preprocessing
- solution
- postprocessing

The preprocessing might be seen as the interface between the real world, where a physical phenomenon does exist or a component is to be constructed, and the computer, where it is simulated. The preprocessor takes a description of the problem as its input and prepares a model to be solved by means of the finite element method. This usually involves *meshing* the problem geometry to discrete finite elements whereupon the solution is approximated by simple mathematical functions such as polynomials – so called basis functions.
The solution is obtained by solving a system of linear equations (SLE) which is *assembled* by taking into consideration the geometry set up by the finite elements, material definitions, source definitions and boundary conditions.
The obtained solution is then prepared for interpretation by the postprocessor. This might include diagrams, physical values obtained by integration or a preparation of further models for a different physical domain. Latter one is known as multiphysics.

Also the *Institute for Fundamentals and Theory in Electrical Engineering* has its main focus on numerical computations [12]. In the early 1980s, the first version of **EleFAnTs** - the **Ele**ctromagnetic **F**ield **A**nalysis **T**ool**s** – was released. These are a set of programs for a numerical treatment of

electromagnetic problems. Nowadays, EleFAnTs comprises a huge amount of assignments in two and three dimensions:

- electrostatics
- time harmonic electric fields
- current flow
- time harmonic current flow
- transient current flow
- magnetostatics
- time harmonic eddy currents
- transient eddy currents
- steady state thermal
- transient thermal

EleFAnTs is used in industry, in an educational environment and also has interfaces to the world wide web [23, 24]. Due to its long development period and thus, its reliability, EleFanTs will be used to prove the results of this thesis.

The programs of EleFAnTs are also written in FORTRAN and therefor – as any other programs written with a procedural or functional paradigm [35] – suffer from some constraints in:

- maintenance
- re-usability
- code readability
- expandability

Maintenance of procedural code is not an easy task as changing or adopting a small number of code lines often forces to rewrite an extensive code thereafter.
Re-usability is mostly not restricted to programming code itself. Documentation of design and description of the program parts and especially distributing work areas to employees is difficult because modules often overlap.
Code readability requires coding standards. Nonetheless, many employees are working in the same code fragments in procedural programming that, of course, incorporates individual coding, commenting and documentation.
Expanding procedural code with new functionality in many cases yields parallel code because new features are not permitted by old code. This results into code running into million of lines and, certainly, also enlarges maintenance efforts[15].

The keywords above are substantially connected to modern software engineering and are somewhat a *must-have*.

In 1990, a first paper on object oriented finite element analysis was released [7] that attracted many other code developers and brought up a multiplicity of new approaches and design recommendations for the three steps preprocessing, solution and postprocessing of finite element analysis.

Almost all publications suggest the same object-oriented architecture in the FEM kernel which is shown in figure 2.1. It has macroelements that consist of several finite elements. In object-oriented terminology this is a so called *has-a relation*. A finite element of triangular shape is set up of three

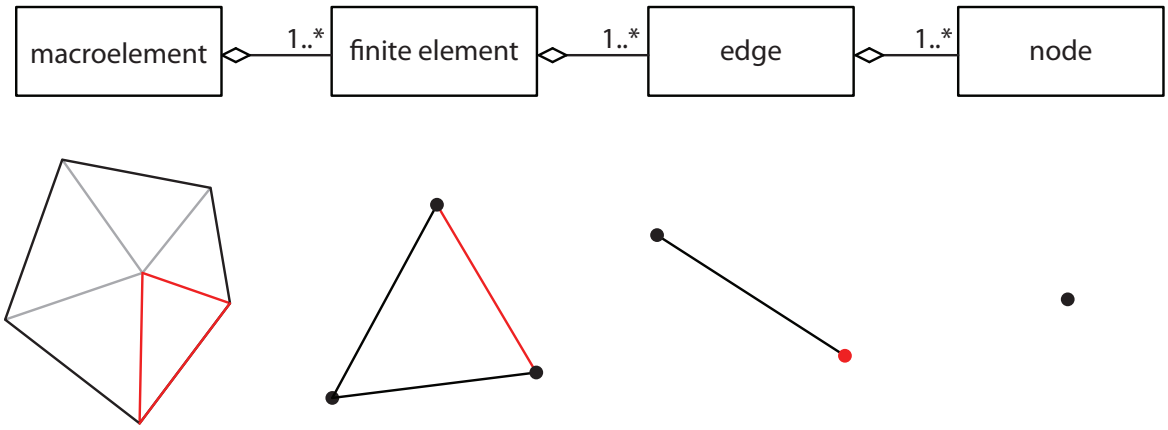

Figure 2.1.: Object-oriented finite element analysis

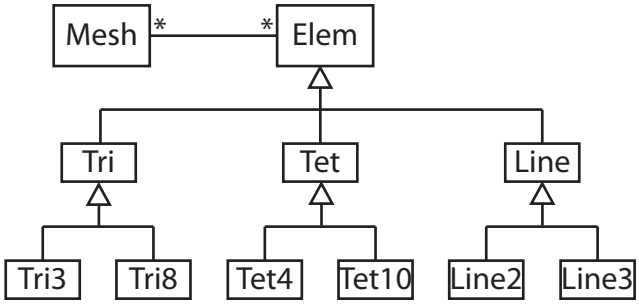

Figure 2.2.: Generalization of the element classes, source: [4]

edges or four edges when it is a quadrilateral. An edge does exist of two nodes. In three dimensional space, where a finite element could be a tetrahedra or hexahedra, this concept would be supplemented by a *face* between the finite element and the edge. [8, 9, 16, 22, 29, 36]
In [22] an even more natural technique is inspired which breaks up the strict correlations from figure 2.1. Identifying the fact that a finite element is either defined by a set of nodes or a set of edges, this is also offered within the object oriented design. The same is valid in three dimensions, where a finite element is defined either by a set of nodes, or a set of edges or additionally by a set of faces. A face would furthermore be given by a set of nodes or edges. Because no colossal advantages could be identified, though, the strict hierarchy from figure 2.1 is obeyed within this thesis. The triangular finite elements are set up by their three nodes each and the edges are created and numbered automatically.

Two basic ideas of object-oriented programming paradigm are *encapsulation* and *inheritance*. The first one is the consolidation of stored data together with the methods that manipulate the data within an object. The second one is the specialization of an object by inheriting its data and methods to another object that describes the parent object more specifically.

Example: Figure 2.2 shows the concept proposed by [4]. It outlines a mesh that is a composition of a number of general finite elements. The *Elem* class will hold properties and methods that are common over any type of finite element in any dimension – e.g. the number of a finite element or the computation of the local stiffness matrix.

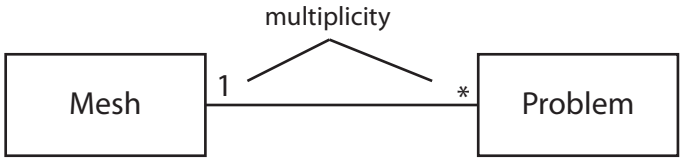

Figure 2.3.: Suggestion by Mai and Henneberger, source: [16]

The next level specializes the general *Elem* class. These are the *Tri* class for two-dimensional triangular shaped finite elements, the *Tet* class for tetrahedrons in three dimensions and a *Line* class for 1-D computations. A triangular finite element can furthermore be a linear element with three nodes and the corresponding *Tri3* class or a triangular finite element with quadratic basis functions and corresponding six nodes and *Tri6* class. Latter two types are currently supported by the implementation introduced by this thesis.
The design of the supported implementation herein omits the second level, because only two-dimensional computations have been implemented. The respective classes are the general *FiniteElement* class and the *FiniteElementT3* as well as the *FiniteElementT6* class.

An in-depth suggestion is given in [16] for a general purpose object-oriented design of finite element software with special attention to coupled problems. The problem and the mesh themselves are treated as objects and can be linked to each other (figure 2.3). For example, one geometry mesh could be linked to a current flow problem and a thermal problem.
To either problem (having the same mesh) different material and source definitions are taken into consideration. E.g., for the current flow problem the material properties represent the electric conductivity while the thermal conductivity must be supplied to the thermal problem. Herein, the problem is a conglomerate of geometry, material and source definitions. Chapter 5 gives a detailed description.

Another very abstract approach can be found within the finite element class library *FEMSTER* [3]. It was also motivated by the need of electromagnetic computations and is aimed to support any kind of partial differential equation (PDE). The object-oriented design comes from a very abstract mathematical definition of the finite element method and relies on a uniquely determined 4-tuple $(\Sigma, \mathcal{P}, \mathcal{A}, \mathcal{Q})$ where Σ is a geometric element, $\mathcal{P}$ is a finite element space defined on Σ, $\mathcal{A}$ is the set of degrees of freedom defined on Σ and $\mathcal{Q}$ is a quadrature rule defined on Σ.

Implementations of object-oriented finite element software could be substantiated mainly in the programming languages **C++** and **Java** [16, 22, 28, 29]. Also the appliance of the Unified Modeling Language (UML) – which is an international standard (ISO) for information technology [9] – is being highly encouraged. The general-purpose modeling language UML is standardized and used for object-oriented software engineering. Many proprietary as well as freely available tools supporting UML do exist. The design for this thesis was made using the open-source tool StarUML [33].

Since March, 2008, MATLAB – the well known development environment for scientific and espe-

cially numerical computations – has come with native object-oriented programming support [18, 31]. It is an easy-to-learn language that contains all aspects of modern object-orientation. The IDE comes with memory management similar to the Java runtime environment and has a very well integrated debugger. Because no experience could be obtained in applying above design suggestions to object-oriented MATLAB code, the idea was born to do this experiment oneself. Chapter 6 will discuss the results.

Chapter 3

MATLAB

Talking about MATLAB many think of just another programming language of plenty. In fact, this is only half of the truth. The **MAT**rix **LAB**oratory is an *integrated development environment (IDE)* for technical algorithm design. It offers an interactive mathematical shell for doing quick calculations or debugging purposes and a source code editor for writing bigger scripts and applications. MATLAB moreover comes with many tools for managing code, files and data and offers a huge collection of so called toolboxes for almost any engineering discipline.

The current version of MATLAB is 7.10 (R2010a), runs on Windows, Linux and MAC each with 32 bit and 64 bit as well. It is written in C and Java and has proprietary licensing.

MATLAB in its current development status is claimed to run specific tasks faster than comparatively with C or C++. It is also called a *fourth generation programming language*, a term that mainly pursues marketing strategies. Some of the benefits, MATLAB offers, are:

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

As a programming language MATLAB refers to multiple programming paradigms. It is *imperative* meaning that it is formulated in *statements* each of which changes the program *state*. It is *procedural*. This is often used as a synonym for imperative but introduces the concept of *subroutines*. A series of statements can be executed at any point of the program. MATLAB is also an array programming language which is a major difference to other popular programming languages. In addition to e.g. integer and floating point types an array is also a *basic type*. Finally, since version 7.6 (R2008a) released in March 2008, MATLAB comes with extensive support for *object oriented programming*. That has made it so attractive for this diploma thesis.

Some of the features are:

- the `classdef` keyword defines a class that can have properties, methods and events
- event handling (not used herein)

- enhanced support by the development environment for object oriented programming

MATLAB features that were specifically used in this work are

- the superclass `handle` allows call-by-reference evaluation which was awkward in earlier releases
- function handles allow the referencing of functions (e.g. used for the numerical quadrature integration)
- packages allow a better modularization of an application (e.g. separation of preprocessor, FEM and postprocessor)
- inheritance allows the specialization of classes (e.g. a finite element can be a triangle or quadrilateral)
- abstraction (e.g. every finite element has a Jacobian matrix but its calculation depends on the shape)
- encapsulation is a basic object-oriented programming paradigm and defines the strict relationships of data and its manipulation methods within an object

A weak point of MATLAB is that it is an interpreted programming language rather than a compiled one which leads to heavy losses in computation speed.

Chapter 4

Theoretical qualification

4.1. Historical retrospective

While the nineteenth century physicist James Clerk Maxwell evolved the theoretical background of electromagnetism, the twentieth century brought up the practical approach that is used today and called **The Finite Element Method** or simply **FEM**. It is a numerical analysis technique in order to obtain an approximated solution to a given engineering problem. An introduction is presented in section 4.4.

Though nowadays FEM – sometimes also referred to as *Finite Element Analysis (FEA)* – mainly emerged in the middle of the twentieth century one could say that it was already used centuries before. Ancient mathematicians, for example, were using it to calculate the circumference of a circle by taking the perimeter of a polygon that approximated the circle. Using an ”inner” polygon and an ”outer” polygon a lower and an upper boundary of the true circumference could be obtained and each edge of the polygon is comparable to a finite element.

With regard to modern numerical computing the idea of FEM first came up in the years 1941 with Russian-Canadian structural engineer Alexander Hrennikoff and 1942 with German mathematician Richard Courant. Both independently had the idea of discretizing a second order partial differential equation problem by meshing its domain into a finite number of subdomains. Courant was the first to do this with triangular shape subdomains on the *St. Venant torsion problem* influenced by earlier contributions of John William Strutt, 3rd Baron Rayleigh, Walther Ritz and Boris Galerkin. [14]

The basic method, as it is used today, was presented the first time by M.J. Turner, R.W. Clough, H.C. Martin and L.J. Topp in their paper *Stiffness and Deflection Analysis of Complex Structures* [34] released in 1956. In his paper *The Finite Element Method in Plane Stress Analysis* [27] R.W. Clough from University of California, Berkley introduced the term *finite element method*. In Europe John Argyris from University of Stuttgart gave momentum to advances in finite element science.

The finite element method in its originating time was mainly used for analysis in aircraft structure. Olgierd Cecil Zienkiewicz, a Polish-British mathematician and civil engineer, together with Y.K. Cheung brought up a broad interpretation and application of FEM for any type of field problems. [19] With the awareness that the FEM equations could also be derived from the weighted residual Galerkin method or the least squares approach the finite element method became attractive for applied mathematics in order to solve both linear and nonlinear differential equations. This opened the door

for applying FEM to electrical engineering, precisely: to Maxwell's equations.

In 1965, NASA requested for proposal of the finite element software *NASTRAN* to which was given a rigorous mathematical foundation in 1973 by Gilbert Strang and George Fix. [30] Finite element software for electrical engineering is to be indebted to the work of the Estonian electrical engineer Peter P. Silvester who released *Finite Elements for Electrical Engineers* [21] in 1983, still a standard textbook on the subject.

To demonstrate the huge interest and the fast development that the finite element method has experienced, consider these facts: Subjecting FEM, in 1974 less than 10 books, in 1982 less than 40 books, in 1990 approximately 400 books had been published. In the period from 1964 to 1991 more than 200 FEM symposia were proceeded. Searching for the terminology "finite element" with AltaVista search engine in 1991 brought up about 200.000 hits. Searching the same term in 2010 with Google finds more than eight million entries. [14]

4.2. Field theory

Before actually introducing Maxwell's equations it seems advisable to take a brief reminiscence of field theory which is a prerequisite of understanding Maxwell's theory.

4.2.1. Field

A field is a state in space. This means that to every point of space a physical quantity is associated. The physical quantity can either be scalar (e.g. the temperature in a room) or vectorial (e.g. the gravitation at a certain location on the surface of the earth). Moreover there also exist tensor and spinor fields though these are of minor importance herein.

Mathematically a scalar field is described by

$$u(x, y, z) = u,$$

where u introduces the electric potential. A vector field is described by

$$\mathbf{E}(x, y, z) = \begin{pmatrix} E_x(x, y, z) \\ E_y(x, y, z) \\ E_z(x, y, z) \end{pmatrix} = \mathbf{E},$$

where E_x, E_y and E_z are the components of the electrical field in x-, y- and z-direction.

4.2.2. Flux

Flux is the sum (integral) of lines of forces (e.g. electrical field or magnetic field) through a given surface in space. Consider water flowing through a pipe and the magnetic field through a conductor loop. Flux always implies a vector field and is defined as

$$\phi = \iint_a \mathbf{B} d\mathbf{a}, \tag{4.1}$$

where $\mathbf{B}$ is the magnetic induction and $d\mathbf{a}$ the area element.

4.2.3. Sources and drains as field cause

Free electric charges cause an electric field whereupon these charges have been separated from their neutralizing countercharges. Positive charges are sources of the electrical field whereas negative charges are drains. Therefor one also talks about *source fields*.

For mathematical purposes one has to distinguish between four occurrences of electric charge:

- free charges at discrete positions (e.g. electrons q_- and ions q_+),
- a density of volume charge ρ and
- a density of surface charge σ
- a density of line charge τ

Note: There is no comparable magnetic charge though there do exist sources of the magnetic field intensity $\mathbf{H}$ (not of $\mathbf{B}$) e.g. on ferromagnetic materials.

Now consider a **closed** surface a_c within an electric field $\mathbf{E}$ that encapsulates a finite volume v. A quantity of field vectors is "pointing into" the surface, another quantity is "pointing out" of it. The electric field is caused by sources and drains inside and outside of v. We are now investigating the flux through a_c using equation (4.1):

$$\phi_c = \oiint_{\partial v} \mathbf{E} d\mathbf{a_c}. \tag{4.2}$$

The value of ϕ_c is called surface flux or **fertility** of the volume and gives information about the volume v.

- If $\phi_c > 0$ more field vectors are pointing out of the surface than into it meaning that the volume has more sources than drains.
- If $\phi_c < 0$ the volume has more drains than sources.
- If $\phi_c = 0$ there are as many sources as there are drains inside the volume also permitting that there are neither sources nor drains.

We have now gathered information about the volume in its entirety and the next arising question is the exact position of the sources and drains within the volume. This leads to the term of **divergence**.

4.2.4. Divergence or source density

The fertility of a volume is an integral statement meaning that the information is gathered by summing up all drains and sources within the volume. For the derivation of the source density (where are the sources and drains) we do the thought experiment of shrinking the surface more and more until only one volume element dv remains. Mathematically this is represented by limit calculation of equation (4.2):

$$\lim_{v \to 0} \frac{1}{v} \oiint_{\partial v} \mathbf{E} d\mathbf{a_c} = \operatorname{div} \mathbf{E} \tag{4.3}$$

The divergence is a differential (local) statement with scalar value and denotes the fertility of an infinitesimal volume element dv. Its calculation specification in Cartesian coordinates, demonstrated with the electrical field is:

$$\operatorname{div} \mathbf{E} = \frac{\partial \mathbf{E_x}}{\partial x} + \frac{\partial \mathbf{E_y}}{\partial y} + \frac{\partial \mathbf{E_z}}{\partial z} = \nabla \mathbf{E} \tag{4.4}$$

where ∇ is the Nabla operator.

4.2.5. Gauß' theorem

By definition the divergence is the fertility of an infinite volume element. Multiplying this fertility (per volume element) with the volume element dv itself and doing a summation (integration) of all volume elements of a finite volume v gives the fertility of the whole volume. We did already calculate this fertility in equation (4.2), therefor:

$$\iiint_v \operatorname{div} \mathbf{E} dv = \oiint_{\partial v} \mathbf{E} d\mathbf{a_c} \tag{4.5}$$

This is **Gauß' theorem** stating that the electric flux through a surface is caused by (and therefor equal to) the sources and drains inside of the surface.

An extension regards the different types of charges we have to take into consideration. Using the electric displacement field $\mathbf{D} = \epsilon \mathbf{E}$ with the *permittivity* ϵ, (4.5) becomes:

$$\iiint_v \operatorname{div} \mathbf{D} dv = \oiint_{\partial v} \mathbf{D} d\mathbf{a_c} = \underbrace{\underbrace{\sum q_i}_{\text{discrete}} + \underbrace{\iiint \rho dv}_{\text{volume}} + \underbrace{\iint \sigma d\mathbf{a}}_{\text{surface}} + \underbrace{\int \tau d\mathbf{l}}_{\text{line charge}}}_{\text{sources of the electric field}} \tag{4.6}$$

4.2.6. Eddy fields

In 4.2.3 source fields have been discussed. Their lines of forces (e.g. electric flux lines) start in positive charges and end in negative countercharges and therefor have an origin (source) and a termination (drain). There do also exist fields whose lines of forces are closed and have neither an originating point nor a terminating one. Consequently sources and drains cannot be the cause of these fields.

An experiment conducted by Michael Faraday gives information about this type of fields. Consider a vertically aligned conductor and a magnetic needle. Moving the needle around the conductor adjusts the needle in tangential direction to the conductor at any point. Hence, the force is axially symmetric and the Faraday lines of force (or magnetic flux lines) are closed. The field cause is the current flow within the conductor. Fields of this type are called **eddy fields**.

Whilst for source fields the surface integral is characteristic for investigating sources and drains, for eddy fields the line integral along the **closed** boundary line of an area is representative for locating eddy causes. Finding such an integral having a value different from zero gives indication for eddy causes. Consider the magnetic field intensity $\mathbf{H}$:

$$\Theta = \oint_{\partial a} \mathbf{H} d\mathbf{l} \tag{4.7}$$

The scalar value Θ is called the **circulation**. It gives information as to whether there are eddy causes within the area that is encapsulated by the closed line integral and their quantity. However, it is an integral (global) statement of the area and therefor does not contain information about the exact position of eddy causes. In the above case, Θ corresponds to the magnetic circuit voltage.

4.2.7. Curl or eddy density

Doing the same thought experiment that uncovered the divergence leads to the **curl** of an eddy field. Instead of shrinking a volume to one volume element, the loop of the closed line integral (4.7) is reduced until it circles only one area element $d\mathbf{a}$. Mathematics allows this by appliance of limit calculation of equation (4.7):

$$\lim_{a \to 0} \frac{1}{a} \oint_{\partial a} \mathbf{H} d\mathbf{l} = (\operatorname{rot} \mathbf{H}) \cdot \mathbf{n} \tag{4.8}$$

Whilst (4.7) is an integral statement in Cartesian coordinates (4.8) is a differential statement. The expression $\operatorname{rot}\mathbf{H}$ is called **curl** and describes the infinitesimal rotation of a three-dimensional vector field. It gives a vector at every point of the field whose direction is the axis of rotation and whose length is the magnitude of rotation of the field. The right side of (4.8) introduces the normal vector $\mathbf{n}$ of the area element $d\mathbf{a}$ and makes it consistent with the left side that is a scalar expression. The calculation specification of the **curl** in Cartesian coordinates is

$$\operatorname{rot}\mathbf{H} = \begin{vmatrix} \mathbf{e}_x & \mathbf{e}_y & \mathbf{e}_z \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ H_x & H_y & H_z \end{vmatrix} = \nabla \times \mathbf{H} \tag{4.9}$$

4.2.8. Stokes' theorem

Similarly to Gauß' theorem the curl is the circulation of an infinitesimal area element. Again, we multiply the circulation per area element with the area element $d\mathbf{a}$ itself and sum up (integrate) all area elements of a finite area a. This gives the circulation of the area which is already given with equation (4.7). Therefor:

$$\iint_a \operatorname{rot}\mathbf{H} d\mathbf{a} = \oint_{\partial a} \mathbf{H} d\mathbf{l} \tag{4.10}$$

This is **Stokes' theorem.** Curls are intensifying or lessening each other in small regions and complement to a resulting circulation along the closed boundary of the area.

4.3. Maxwell's equations

Having field theory returned to mind now gives the opportunity to eventually introduce James Clerk Maxwell's famous equations. These do

- relate the electric field to its sources: the electric charge density
- relate the magnetic field to its sources: the electric current density
- relate the electric field to the magnetic field and
- relate the magnetic field to the electric field.

The relations are described by four *partial differential equations*:

Gauß' law Electric charges are the source of an electric field (**or:** electric charges cause an electric field) – this is the differential statement. The field starts in positive charges and ends in negative countercharges. **And:** The electric flux through a closed Gaußian surface is caused by the sources and drains within the volume that is enclosed by the surface – this is the integral statement.

differential form	**integral form**
$\operatorname{div} \mathbf{D} = \rho$	$\oiint_{\partial v} \mathbf{D} d\mathbf{a_c} = \iiint_v \rho dv = Q(v)$

(4.11)

Gauß' law for magnetism In contrast to electric charges there are no comparable magnetic charges - this is the differential statement. Magnetic lines of forces are closed and have neither an origin nor an ending. The magnetic flux through a closed Gaußian surface is zero - this is the integral statement. Magnetic lines of forces that "flow into" a volume must also "flow out" of the volume because there cannot be sources or drains. Consequently the total flux is zero.

differential form	**integral form**
$\operatorname{div} \mathbf{B} = 0$	$\oiint_{\partial v} \mathbf{B} d\mathbf{a_c} = 0$

(4.12)

Faraday's law of induction A changing magnetic field induces an electric field. Every change of the $\mathbf{B}$-field leads to a counteracting $\mathbf{E}$-field - this is the differential statement. The electrical circulation along the closed boundary of an area equals the negative time varying aspect of the magnetic flux through the area - this is the integral statement.

differential form	**integral form**
$\operatorname{rot} \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$	$\oint_{\partial \mathbf{a}} \mathbf{E} d\mathbf{l} = -\frac{\partial}{\partial t} \iint_{\mathbf{a}} \mathbf{B} d\mathbf{a} = -\frac{\partial \Phi_{\mathbf{B}}}{\partial t}$

(4.13)

Ampére's circuital law is the analogy to Faraday's law of induction for magnetic fields. A magnetic field can be generated either by electric current or a changing electric field. The eddy causes of a magnetic field are electric current densities (e.g. currents in a conductor) and displacement currents (e.g. when loading a capacity) - this is the differential statement. The magnetic circulation along the closed boundary of an area is equal to the electric current and the displacement current through the area - this is the integral statement.

differential form	**integral form**
$\text{rot}\,\mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$	$\oint_{\partial \mathbf{a}} \mathbf{H} d\mathbf{l} = \iint_{\mathbf{a}} \mathbf{J} d\mathbf{a} + \frac{d}{dt} \iint_{\mathbf{a}} \mathbf{D} d\mathbf{a} = I + \frac{\partial \Phi_{\mathbf{D}}}{\partial t}$

(4.14)

Summarizing above equations in terms of free charges and currents and in terms of total charges and currents and combing them with the material equations gives the final set of Maxwell's equations:

Formulation in terms of total charges

$$\text{div}\,\mathbf{E} = \frac{\rho}{\epsilon_0} \tag{4.15a}$$

$$\text{div}\,\mathbf{B} = 0 \tag{4.15b}$$

$$\text{rot}\,\mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{4.15c}$$

$$\text{rot}\,\mathbf{B} = \mu \mathbf{J} + \mu \epsilon \frac{\partial \mathbf{E}}{\partial t} \tag{4.15d}$$

Formulation in terms of free charges

$$\text{div}\,\mathbf{D} = \rho_{\text{free}} \tag{4.16a}$$

$$\text{div}\,\mathbf{B} = 0 \tag{4.16b}$$

$$\text{rot}\,\mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{4.16c}$$

$$\text{rot}\,\mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t} \tag{4.16d}$$

Material equations

$$\mathbf{D} = \epsilon \mathbf{E} \qquad \epsilon = \epsilon_0 \epsilon_r \tag{4.17a}$$

$$\mathbf{B} = \mu \mathbf{H} \qquad \mu = \mu_0 \mu_r \tag{4.17b}$$

4.3.1. Electro- and magnetostatics - The Laplace-Poisson equation

Static in general means that there is both no temporal change and no change of the energy of a system. The corresponding mathematical notation is

$$\frac{\partial}{\partial t} = 0 \tag{4.18a}$$

$$dW = 0 \tag{4.18b}$$

Applying this presumption to Maxwell's theory gives the basic equations for electro- and magnetostatics:

$$\operatorname{div} \mathbf{D} = \rho \tag{4.19a}$$

$$\operatorname{rot} \mathbf{E} = 0 \tag{4.19b}$$

$$\mathbf{D} = \epsilon_0 \epsilon_r \mathbf{E} = \epsilon \mathbf{E} \tag{4.19c}$$

$$\operatorname{div} \mathbf{B} = 0 \tag{4.19d}$$

$$\operatorname{rot} \mathbf{H} = \mathbf{J} \tag{4.19e}$$

$$\mathbf{B} = \mu_0 \mu_r \mathbf{H} = \mu \mathbf{H} \tag{4.19f}$$

The first three equations of (4.19) involve electrical quantities only while the last three equations involve magnetic quantities only. The coupling between the electric and the magnetic field disappears in a static system. This is because there is no temporal change and no current flow. Consequently the electric and magnetic field can be treated independently from each other.

Because the electric field is irrotational – $\operatorname{rot} \mathbf{E} = 0$ – it can be described as the gradient of a scalar potential field u:

$$\mathbf{E} = -\operatorname{grad} u, \text{because}$$

$$\begin{aligned} \operatorname{rot} \mathbf{E} &= \operatorname{rot} (-\operatorname{grad} u) \\ &= \nabla \times (-\nabla u) \\ &= (\nabla \times \nabla) \cdot (-u) = 0. \end{aligned} \tag{4.20}$$

With the given sources of the electric flux and assuming an isotropic permittivity $\epsilon = \epsilon_r \epsilon_0$

$$\operatorname{div} \mathbf{D} = \epsilon \operatorname{div} \mathbf{E} = \rho \tag{4.21}$$

and using the scalar potential u yields the

Laplace-Poisson equation

$$\operatorname{div} (\operatorname{grad} u) = \Delta u = -\frac{\rho}{\epsilon} \tag{4.22}$$

This is called the *Laplace-Poisson equation* with the **Laplace operator** Δ.

$$\Delta = \text{ div grad } = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \tag{4.23}$$

in Cartesian coordinates. Hence,

$$\Delta u = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) u(x, y, z) = -\frac{\rho(x, y, z)}{\epsilon(x, y, z)} \tag{4.24}$$

Solving this equation for electrostatic, magnetostatic, static current flow and thermal problems is the subject of this work. Its practical implementation is the topic of the following chapter.

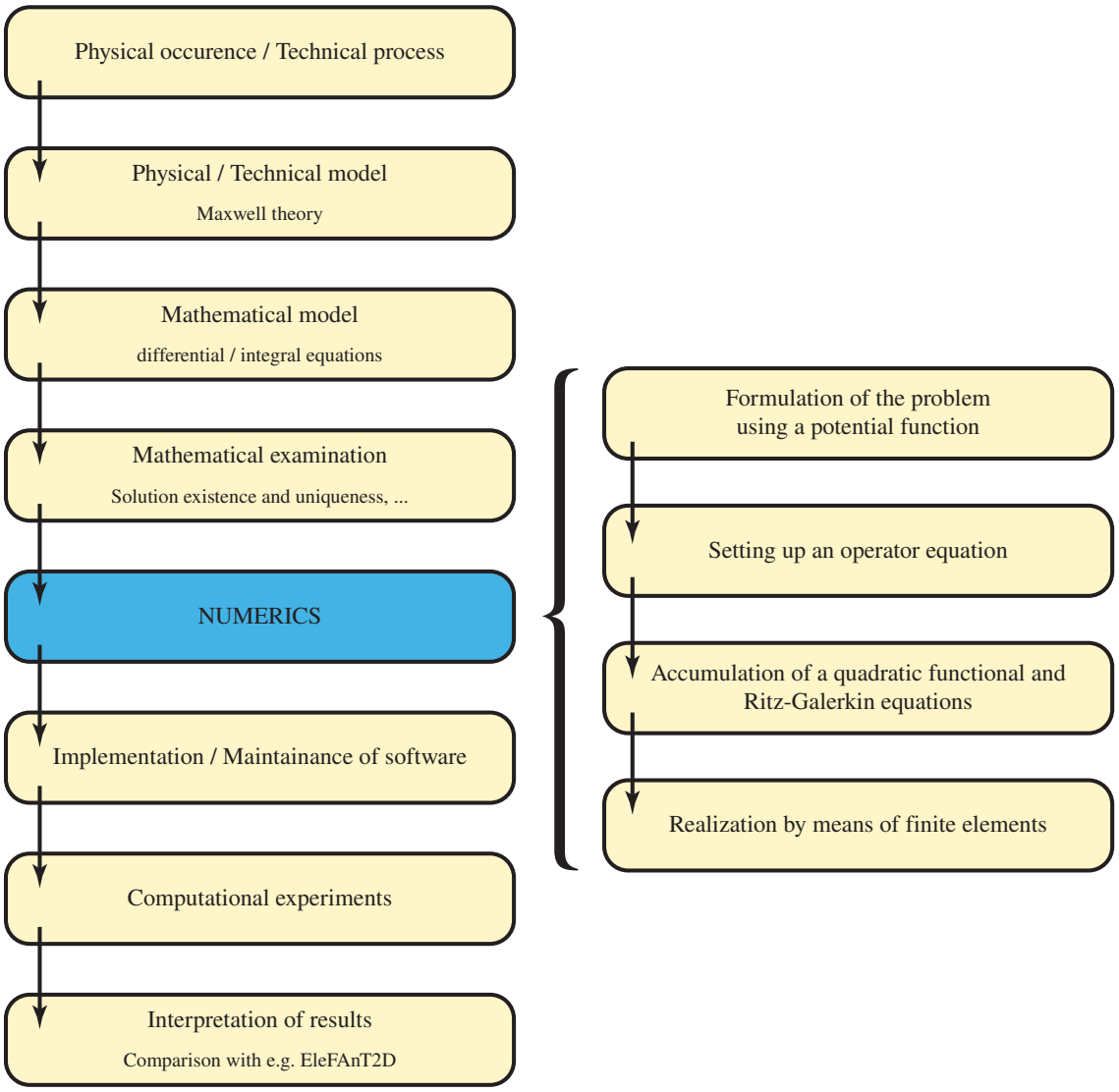

Figure 4.1.: Simulating a technical process

4.4. Finite element Method

The last section has acquainted the reader with Maxwell's equations in three-dimensional space. Within the next pages we do a preparation of the theory in order to develop the practical implementation. Thus the focus is now switched from three- to two-dimensional space with coordinates x and y. As an introduction to the finite element method a first global view to the whole process of simulating a technical process is highlighted in figure 4.1 which is leaned to [13]:

Figure 4.1 shows all steps that need to be carried out in order to come from a physical occurrence in nature to interpretable and mathematically correct results with the help of computerized simulation. The actual focus in this thesis is numerics whose four basic steps of application flow should be stated once more at this point:

- Formulation of the problem using a potential function.

- Setting up an operator equation.
- Accumulation of a quadratic functional and Ritz-Galerkin equations.
- Realization by means of finite elements.

4.4.1. Ritz-Galerkin method

Getting to know the finite element method with all mathematical aspects makes a deep insight into **variational calculus** indispensable. For an understanding of the essentials of this work this is not mandatory and the interested reader is referred to [2] and [25]. Rather, we apply the results of variational calculus to the subject Laplace-Poisson equation (4.22).

The assignment to be solved is

$$\mathcal{A}u \stackrel{!}{=} f \tag{4.25}$$

where $\mathcal{A}$ denotes a **linear**, **symmetric** and **positive definite** operator to the variable u and f is any function.

Following to variational calculus assignment (4.25) can be solved by minimizing

$$W(u) = \frac{1}{2}\iint_\Omega u\mathcal{A}ud\Omega - \iint_\Omega ufd\Omega \tag{4.26}$$

Hence, assuming $\bar{u}$ is the exact solution to assignment (4.25) then (and only then!)

$$W(\bar{u}) = \text{minimum!} \Longleftrightarrow \mathcal{A}\bar{u} = f \tag{4.27}$$

Until this point, we have looked at the continuous potential function $u(x, y)$. Because analytical solutions do exist only to a very limited amount of impractical assignments one has to be content to switch from continuous space to discrete space. The potential function u is now approximated with the discrete potential function u_n:

$$u \approx u_n = \sum_{i=1}^{n} a_i f_i + \underbrace{\sum_{k=n+1}^{n+m} u_k f_k}_{u_D} \tag{4.28}$$

Some notes:

- A limited amount of discrete points $m + n$ in two-dimensional space is considered now.
- The index k refers to m points whose potential values u_k are known from the problem definition – so called *Dirichlet points* (or *Dirichlet boundary conditions*).
- The index i refers to n points whose potential values a_i are not known and therefor subject of the computation. **The determination of the coefficients a_i is the solution of the problem.**
- The problem has n unknowns – therefor n is the degree of freedom of the problem.
- All potential values at continuous coordinates between this set of discrete points are interpolated by the *basis functions* f_k and f_i respectively. Their shape is subject of an oncoming discussion.

Before inserting the discretized potential function (4.28) into functional (4.26) two important derivatives are illustrated:

$$\frac{\partial u_n}{\partial a_i} = \frac{\partial}{\partial a_i}\left(u_D + a_1 f_1 + a_2 f_2 + \cdots + a_i f_i + \cdots + a_n f_n\right) = f_i \tag{4.29}$$

$$\frac{\partial}{\partial a_i}\left(\mathcal{A}u_n\right) = \frac{\partial}{\partial a_i}\left[\mathcal{A}\left(u_D + a_1 f_1 + a_2 f_2 + \cdots + a_i f_i + \cdots + a_n f_n\right)\right] = \\ = \frac{\partial}{\partial a_i}\left[\mathcal{A}\left(a_i f_i\right)\right] = \mathcal{A}f_i \tag{4.30}$$

Inserting (4.28) into (4.26) gives

$$W(u_n) = \frac{1}{2}\iint_\Omega u_n \mathcal{A}u_n d\Omega - \iint_\Omega u_n f d\Omega \tag{4.31}$$

Minimizing this functional gives the solution to assignment (4.25) with the discretized potential:

$$\mathcal{A}u_n = f \text{ with } u_n \approx u \tag{4.32}$$

The coefficients a_i are the unknowns to be determined. The functional has its minimum when all partial derivatives in respect to the coefficients a_i are zero. This is also known as the *first variation* δ of the functional:

$$\delta(W) = \frac{\partial W(u_n)}{\partial a_i} = 0 \qquad\qquad i = 1, 2, \ldots, n \tag{4.33}$$

Hence, solving these n equations yields the solution of the problem.

Evaluating (4.33) gives

$$\frac{\partial W(u_n)}{\partial a_i} = \frac{1}{2}\iint_\Omega \underbrace{\frac{\partial u_n}{\partial a_i}}_{=f_i} \mathcal{A}u_n d\Omega + \frac{1}{2}\iint_\Omega u_n \underbrace{\frac{\partial}{\partial a_i}\left(\mathcal{A}u_n\right)}_{=\mathcal{A}f_i} d\Omega - \iint_\Omega \underbrace{\frac{\partial u_n}{\partial a_i}}_{=f_i} f d\Omega = \\ = \underbrace{\frac{1}{2}\iint_\Omega f_i \mathcal{A}u_n d\Omega + \frac{1}{2}\iint_\Omega u_n \mathcal{A}f_i d\Omega}_{\text{Symmetry: } f_i\mathcal{A}u_n = u_n\mathcal{A}f_i} - \iint_\Omega f_i f d\Omega = \\ = \iint_\Omega f_i \mathcal{A}u_n d\Omega - \iint_\Omega f_i f d\Omega = \\ = \iint_\Omega f_i\left(\mathcal{A}u_n - f\right) d\Omega = 0 \tag{4.34}$$

Summary:

Ritz-Galerkin equations

$$u_n = u_D + \sum_{i=1}^{n} a_i f_i \tag{4.35a}$$

$$\iint_\Omega f_i \mathcal{A}u_n d\Omega = \iint_\Omega f_i f d\Omega \qquad\qquad i = 1, 2, \ldots, n \tag{4.35b}$$

Considering that f_i are the basis functions and f is the right side of equation (4.25). Equation (4.35b) is equal to (4.34) with known function f brought to the right side.

Having elaborated the Ritz-Galerkin equation (4.35b) the operator $\mathcal{A}$ can be substituted with the negative Laplace operator $-\Delta$ that is linear, symmetric and positive definite. Therefor the Laplace-Poisson equation is written as

$$-\Delta u = \frac{\rho}{\epsilon} \tag{4.36}$$

Appliance of the Laplace-Poisson equation to the Ritz-Galerkin equation (4.35b) gives

$$\iint_\Omega f_i(-\Delta u_n)d\Omega = \iint_\Omega f_i\frac{\rho}{\epsilon}d\Omega \qquad\qquad i = 1, 2, \ldots, n \tag{4.37}$$

Bringing ϵ to the left side and using div grad as notation has

$$-\iint_\Omega f_i(\,\text{div}\,\epsilon\,\text{grad}\,u_n)d\Omega = \iint_\Omega f_i\rho d\Omega \qquad\qquad i = 1, 2, \ldots, n \tag{4.38}$$

Note: ϵ is a tensor that can represent anisotropic materials. This has also been implemented into the software. Further considerations will treat ϵ as a scalar dependent on a position.

By the use of *partial differential integration* and *Green's identity* [see 13, sec. 4.2] (4.38) becomes

$$-\iint_\Omega (\,\text{grad}\,f_i)^T\epsilon\,\text{grad}\,u_n d\Omega = \iint_\Omega f_i\rho d\Omega + \int_{\Gamma_D} f_i u_D d\Gamma_D \qquad i = 1, 2, \ldots, n \tag{4.39}$$

and substituting u_n from (4.28)

$$-\iint_\Omega (\,\text{grad}\,f_i)^T\epsilon\,\text{grad}\sum_{j=1}^{n} a_j f_j d\Omega = \iint_\Omega f_i\rho d\Omega + \int_{\Gamma_D} f_i u_D d\Gamma_D \qquad i = 1, 2, \cdots, n \tag{4.40}$$

Because u_D is defined on boundary $\Gamma_D \subseteq \partial\Omega$ only and $\Omega \cap \partial\Omega = \emptyset$ it does not appear on the left side of the equation. Index i is already used for equations (4.34) and (4.35b) respectively, thus (4.40) introduces index j.

Considering the linearity of the Laplace operator the sum can enclose the integral:

$$-\sum_{j=1}^{n} a_j \iint_\Omega (\,\text{grad}\,f_i)^T\epsilon\,\text{grad}\,f_j d\Omega = \iint_\Omega f_i\rho d\Omega + \int_{\Gamma_D} f_i u_D d\Gamma_D \qquad i = 1, 2, \cdots, n \tag{4.41}$$

Summing up the left and right side for all indexes i gives

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_j \iint_\Omega (\,\text{grad}\,f_i)^T\epsilon\,\text{grad}\,f_j d\Omega = -\sum_{i=1}^{n}\left(\iint_\Omega f_i\rho d\Omega + \int_{\Gamma_D} f_i u_D d\Gamma_D\right) \tag{4.42}$$

which can also be written in matrix form:

$$\mathbf{K}\mathbf{a} = \mathbf{f} \tag{4.43}$$

with

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

$$\mathbf{a} = \begin{Bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{Bmatrix} \quad \text{and} \quad \mathbf{f} = \begin{Bmatrix} -\iint_\Omega f_1 \rho d\Omega - \int_{\Gamma_D} f_1 u_D d\Gamma_D \\ -\iint_\Omega f_2 \rho d\Omega - \int_{\Gamma_D} f_2 u_D d\Gamma_D \\ \vdots \\ -\iint_\Omega f_n \rho d\Omega - \int_{\Gamma_D} f_n u_D d\Gamma_D \end{Bmatrix} \tag{4.44}$$

Some notes:

- Matrix $\mathbf{K}$ is an $(n \times n)$ square matrix. Coming from structural analysis it is called the **stiffness matrix**.
- Vector $\mathbf{f}$ has n entries and is announced as the **load vector**.
- Setting up the stiffness matrix and the load vector is called **assembly** of the finite element system of equations.

This has been a very brief inspection of the Ritz-Galerkin method that should now be introduced as the finite element method implemented within this thesis. This inspection misses a lot of preliminary mathematical examinations like the existence of a solution, its uniqueness and the demand for its smoothness. The mathematically experienced reader can find an excellent introduction – at first into one-dimensional, on with higher-dimensional – FEM with lots of examples and mathematical background in book *Methode der finiten Elemente für Ingenieure* [13] by Michael Jung and Ulrich Langer.

The missing part at this point is the basis functions f_i and will be outlined within the next few pages.

4.4.2. Finite Elements

Although the code implementation is prepared for any kind of finite element shapes and even for the third dimension, the focus at the current development state – and in this work – is laid on two-dimensional triangular shaped finite elements. As already described in 4.3.1, the static problem is modeled within a domain Ω with boundary $\partial\Omega = \Gamma = \Gamma_E \cup \Gamma_D$ (for electrostatic problems). Because analytical solutions practically do not exist in continuous space the domain Ω is discretized in a finite amount of subdomains – so called **finite elements** $T^{(r)}$ with r being the index of the elements.

The discretization must provide these assumptions:

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

$$\Omega \approx \bigcup_{r=1}^{R} T^{(r)} \tag{4.45}$$

- Any two finite elements $T^{(r)}$ and $T^{(r')}$ must apply to:

$$T^{(r)} \cap T^{(r')} = \begin{cases} \emptyset \\ \text{one common node} \\ \text{one common edge} \end{cases} \tag{4.46}$$

We call this discretization a **triangulation**. Every node of the triangulation gets a unique **global** identification number.

4.4.2.1. Basis functions

Basis functions are (simple) functions that are defined for each node of the triangulation. Take any node of the triangulation: An important characteristic of the basis function defined for this node is that

- its value is 1 at the position of the node and
- its value is 0 at the position of any other node and
- its value is different from 0 only within the finite elements that share this node.

These basis functions will "span" the solution over the **global** domain Ω. Figure 4.2 illustrates the basis functions for two neighboring nodes P_{14} and P_{43} of a sample triangulation. The adjacent finite elements of node P_{14} are highlighted.

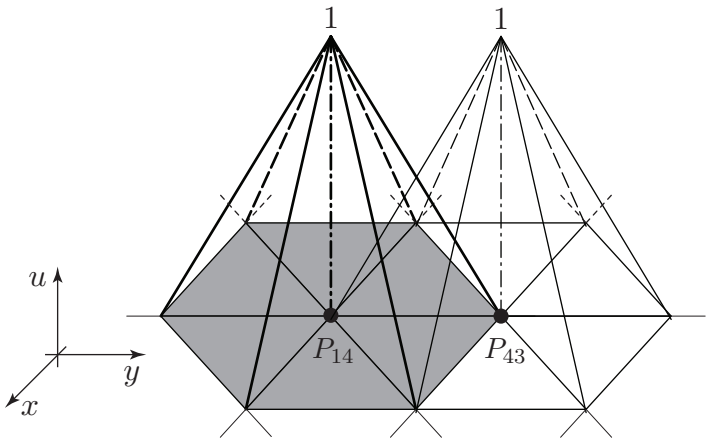

Figure 4.2.: Basis functions

The global shape of a finite element is obtained by a projection from a **local reference element**. The reference element is defined by the triangle

$$\hat{T} = \{(\xi, \eta) : 0 \leq \xi, \eta \leq 1, \xi + \eta \leq 1\} \tag{4.47}$$

with local coordinates ξ and η. The left diagram of figure 4.3 shows the reference element in local space.

The three basis functions corresponding to the three local nodes $\alpha = \{1, 2, 3\}$ over this reference triangle are

$$\hat{f}_\alpha(\xi, \eta) = \begin{cases} \alpha = 1 : 1 - \xi - \eta \\ \alpha = 2 : \xi \\ \alpha = 3 : \eta \end{cases} \tag{4.48}$$

Of course, these functions are not defined arbitrarily and must also apply to certain presumptions. A deeper investigation can be found in [13].

In order to link local coordinates (ξ, η) of the reference element to global coordinates (x, y) of the problem domain the following transformation is applied:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{J}^{(r)} \begin{pmatrix} \xi \\ \eta \end{pmatrix} + \begin{pmatrix} x_1^{(r)} \\ y_1^{(r)} \end{pmatrix} \tag{4.49}$$

with the **Jacobian matrix**

$$\mathbf{J}^{(r)} = \begin{bmatrix} x_2^{(r)} - x_1^{(r)} & x_3^{(r)} - x_1^{(r)} \\ y_2^{(r)} - y_1^{(r)} & y_3^{(r)} - y_1^{(r)} \end{bmatrix} \tag{4.50}$$

where $(x_1, y_1)^{(r)}$, $(x_2, y_2)^{(r)}$ and $(x_3, y_3)^{(r)}$ are the global coordinates of nodes 1, 2 and 3 of finite element r. The structure of the Jacobian matrix $\mathbf{J}^{(r)}$ is defined for each finite element and dependent on its global deformation.

The inverse transformation from global coordinates (x, y) to local coordinates (ξ, η) is non-linear for finite elements of higher order and cannot be stated in a similar simple form. It is described in the section dealing with the postprocessor of the next chapter.

Figure 4.3 shows the transformation from local to global coordinates. The reference element $\hat{T}$ on the left has local nodes $\hat{P}_1$, $\hat{P}_2$ and $\hat{P}_3$ and represents – for example – finite element $T^{(4)}$ with global nodes P_2, P_5 and P_7 of a sample triangulation.

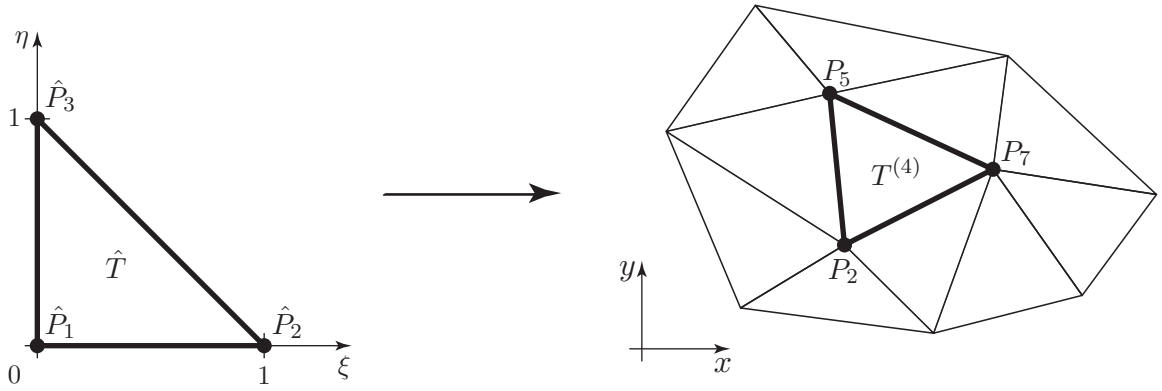

Figure 4.3.: Transformation from local to global coordinates

4.4.3. Assembly

With the concept of finite elements and its basis functions it is now time to develop the stiffness matrix $\mathbf{K}$ and the load vector $\mathbf{f}$ shown in (4.44). This is called **assembly**.

The left side of equation (4.42) is:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_j \iint_{\Omega} (\operatorname{grad} f_i)^T \epsilon \operatorname{grad} f_j d\Omega \tag{4.51}$$

The continuous domain Ω has been discretized by a triangulation – see (4.45). Thus, the integration is approximated using all subdomains of the triangulation or the finite elements $T^{(r)}$, respectively:

$$\sum_{r=1}^{R} \left\{ \underbrace{\sum_{i=1}^{n}\sum_{j=1}^{n} a_j \iint_{T^{(r)}} (\operatorname{grad} f_i(x,y))^T \epsilon \operatorname{grad} f_j(x,y) dxdy}_{=0 \text{ if node } P_i \text{ or node } P_j \notin T^{(r)}} \right\} \tag{4.52}$$

Considering that the basis functions are different from zero only in one node and the scope of its adjacent finite elements it is sufficient to evaluate integration (4.52) only for nodes that belong to the current finite element r of the outmost sum:

$$\sum_{r=1}^{R}\sum_{i^{(r)}}\sum_{j^{(r)}} a_j \iint_{T^{(r)}} (\operatorname{grad} f_i(x,y))^T \epsilon \operatorname{grad} f_j(x,y) dxdy \tag{4.53}$$

with

$$i^{(r)} = \left\{ i : P_i \in T^{(r)} \right\} \tag{4.54}$$

$$j^{(r)} = \left\{ j : P_j \in T^{(r)} \right\} \tag{4.55}$$

Basis functions f_i and f_j in equation (4.53) are still formulated in global space with coordinates x and y. If these were formulated in terms of local space it was possible to do all computations by means of the local reference element $\hat{T}$.

A linkage between global and local nodes has to be implemented:

$$\alpha \leftrightarrow i = i^{(r)}(\alpha) \tag{4.56}$$

$$\beta \leftrightarrow j = j^{(r)}(\beta) \tag{4.57}$$

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

With some conversions that can be reconstructed in [13, pages 208-211] equation (4.53) becomes

$$\sum_{r=1}^{R}\sum_{\alpha=1}^{3}\sum_{\beta=1}^{3} a_\beta \iint_{\hat{T}} (\text{ grad } \hat{f}_\alpha(\xi,\eta))^T \hat{\epsilon}\, \text{grad } \hat{f}_\beta(\xi,\eta) d\xi d\eta = \sum_{r=1}^{R} \mathbf{K}^{(r)} \tag{4.58}$$

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

$$\mathbf{K}^{(r)} = \sum_{\alpha=1}^{3}\sum_{\beta=1}^{3} a_\beta \iint_{\hat{T}} (\text{ grad } \hat{f}_\alpha(\xi,\eta))^T \hat{\epsilon}\, \text{grad } \hat{f}_\beta(\xi,\eta) d\xi d\eta \tag{4.59}$$

In analogy there is the **element load vector** $\mathbf{f}^{(r)}$. Neglecting the boundary condition term on the right side of (4.42) – $\int_{\Gamma_D} f_i u_D d\Gamma_D$ – and using local scope formulation, it becomes:

$$\mathbf{f}^{(r)} = \sum_{\alpha=1}^{3} \iint_{\hat{T}} f(x(\xi,\eta), y(\xi,\eta)) \hat{f}_\alpha(\xi,\eta) \left|\det J^{(r)}\right| d\xi d\eta \tag{4.60}$$

where $x(\xi,\eta)$ respectively $y(\xi,\eta)$ is the local-to-global transformation (4.49).

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

$$\mathbf{C}^{(r)}(\alpha, i) = \begin{cases} 1, \text{ if } i \text{ is the global node number of local node } \alpha \\ 0, \text{ otherwise} \end{cases} \tag{4.61}$$

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

The element stiffness matrix $\mathbf{K}$ is now constructed with:

$$\mathbf{K} = \sum_{r=1}^{R} \left(\mathbf{C}^{(r)}\right)^T \mathbf{K}^{(r)} \mathbf{C}^{(r)} \tag{4.62}$$

and the load vector $\mathbf{f}$ with:

$$\mathbf{f} = \sum_{r=1}^{R} \left(\mathbf{C}^{(r)}\right)^T \mathbf{f}^{(r)} \tag{4.63}$$

The solution of the linear equation system (4.43) are the coefficients a_i:

$$\mathbf{a} = \{a_1, a_2, \ldots, a_n\} = \mathbf{K}^{-1}\mathbf{f} \tag{4.64}$$

Practical problems usually yield several thousand to million coefficients a_i and the solution cannot simply be gathered by inverting the stiffness matrix to $\mathbf{K}^{-1}$. For small problems *Newton's method* might be sufficient to solve equation (4.64). Large problems and higher-dimension problems claim for *gradient descent methods* like the *conjugate gradient method* and numerous varieties of it.

The solution to (4.64) is an issue to further developments beyond this thesis. In the implementation the solution step is abandoned to MATLAB.

4.4.4. Integration of boundary conditions

Neumann boundary conditions Integrating the Neumann boundary condition term of (4.42) into the FEM system of equations changes the load vector $\mathbf{f}$ as can be seen in (4.44). A similar approach, as done setting up the load vector $\mathbf{f}$, is depicted in [13] in section 4.5.3 (pages 219 et seq.).

As with the whole domain Ω, the boundary $\partial\Omega$ is partitioned into finite elements. These are not two-dimensional triangles but one-dimensional straight lines $E^{(e)}$. The letter E means *edge* and e is the index. There also exists a reference element $\hat{E}$ which is a one-dimensional straight line with local coordinate ξ within the interval $[0, 1]$. It has two local nodes at $\xi = 0$ and $\xi = 1$. This construction leads to an *edge element load vector* $\mathbf{f}^{(e)}$:

$$\mathbf{f}^{(e)} = \sum_{\alpha=1}^{2} \int_{\hat{E}} u_D(\xi)\hat{f}_\alpha(\xi)d\xi \tag{4.65}$$

Together with an *edge element coherence matrix* $\mathbf{C}^{(e)}$, that relates 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:

$$\mathbf{f} = \mathbf{f} + \mathbf{C}^{(e)T}\mathbf{f}^{(e)} \tag{4.66}$$

For Cauchy boundary conditions, also an *edge element stiffness matrix* must be computed and integrated into the stiffness matrix $\mathbf{K}$. The interested reader is referred to [13] for further details on this technique.

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

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \cdot \begin{Bmatrix} a_1 \\ \bar{a}_2 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \\ f_3 \end{Bmatrix} \tag{4.67}$$

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

$$\begin{bmatrix} k_{11} & 0 & k_{13} \\ k_{21} & 0 & k_{23} \\ k_{31} & 0 & k_{33} \end{bmatrix} \cdot \begin{Bmatrix} a_1 \\ \bar{a}_2 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} f_1 - k_{12}\bar{a}_2 \\ f_2 - k_{22}\bar{a}_2 \\ f_3 - k_{32}\bar{a}_2 \end{Bmatrix} \tag{4.68}$$

Deleting the second row of the stiffness matrix and the load vector and deleting the second column of the stiffness matrix yields the final FEM system of equations:

$$\begin{bmatrix} k_{11} & k_{13} \\ k_{31} & k_{33} \end{bmatrix} \cdot \begin{Bmatrix} a_1 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} f_1 - k_{12}\bar{a}_2 \\ f_3 - k_{32}\bar{a}_2 \end{Bmatrix} \tag{4.69}$$

This is called *homogenizing*. The general approach is:

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

$$f_i = f_i - \sum_j \mathbf{K}(i,j)a_j \qquad \forall i \neq j \tag{4.70}$$

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

4.5. Quadrature integration

Quadrature integration is the common way to numerically approximate integral terms. Section 4.5.5 of [13] shows a deduction.

The approximation is

$$\iint_{\Omega} f(\xi,\eta)d\xi d\eta \approx \sum_{i=1}^{l} w_i f(\xi_i,\eta_i) \tag{4.71}$$

The method is, to evaluate and sum up a function $f(\xi,\eta)$ at so called *supporting points*. These l points are weighted with weights w_i. Table 5.3 on page 67 lists different configurations of quadrature integration.

Of course, this chapter was only an outline of the vast theoretical complexity of the finite element method. Anyway, it is assumed that the reader does already have experience with this topic. Therefor, the theoretical overview should be sufficient in order to understand the actual implementation of the algorithm that will be discussed in-depth within the next chapter.

Chapter 5

Practical approach

This chapter describes and discusses the practical realization of the concepts as depicted by the last chapter. Since the initiation of the project on August 5th, 2008, the implementation was referred to as ***FEMtastic***. The name is derived from the word *fantastic*, obviously, and is thought to set the goal of the project.

5.1. Implementation schemata

In order to obtain a solution to an electrostatic, magnetostatic, static current flow or thermal problem, the basic approach is similar to any other finite element application. Figure 5.1 shows the schema of FEMtastic. Each block on the left side resembles one module in FEMtastic and represents one basic step of the solution approach.

Preprocessor The preprocessor is the leadoff interface between the user and the framework. The user defines the assignment by providing an adequate XML file. It is interpreted by the parser (that is depicted as the top right **XML** block) and its information is incorporated into the preprocessor.

The geometry of an assignment is composed of macro elements. FEMtastic is thought to be capable of building any two-dimensional geometry. This has been realized by implementing an efficient up-to-date algorithm within MATLAB. The algorithm applies boolean operations to polygons. These operations are *union* (A or B), *intersection* (A and B) and *subtraction* (A and not B). It was engineered by Francisco Martínez, Antonio Jesús Rueda and Francisco Ramón Feito and presented in their article *A new algorithm for computing Boolean operations on polygons* [17] in *Computers and Geosciences* journal in 2009. The **Martinez** block of the diagram corresponds to the algorithm. It applies the boolean operations to the macro elements (both are defined within the XML file) and passes back the finalized geometry to the preprocessor.

The final job of the preprocessor is to provide a triangulation of the geometry to the finite element method. This is achieved with the MATLAB toolbox `mesh2d` [6] by Darren Engwirda. It uses quadtree decomposition and Delaunay triangulation and produces meshes of very high quality. The algorithm is described in [5]. It is a modification of [20].

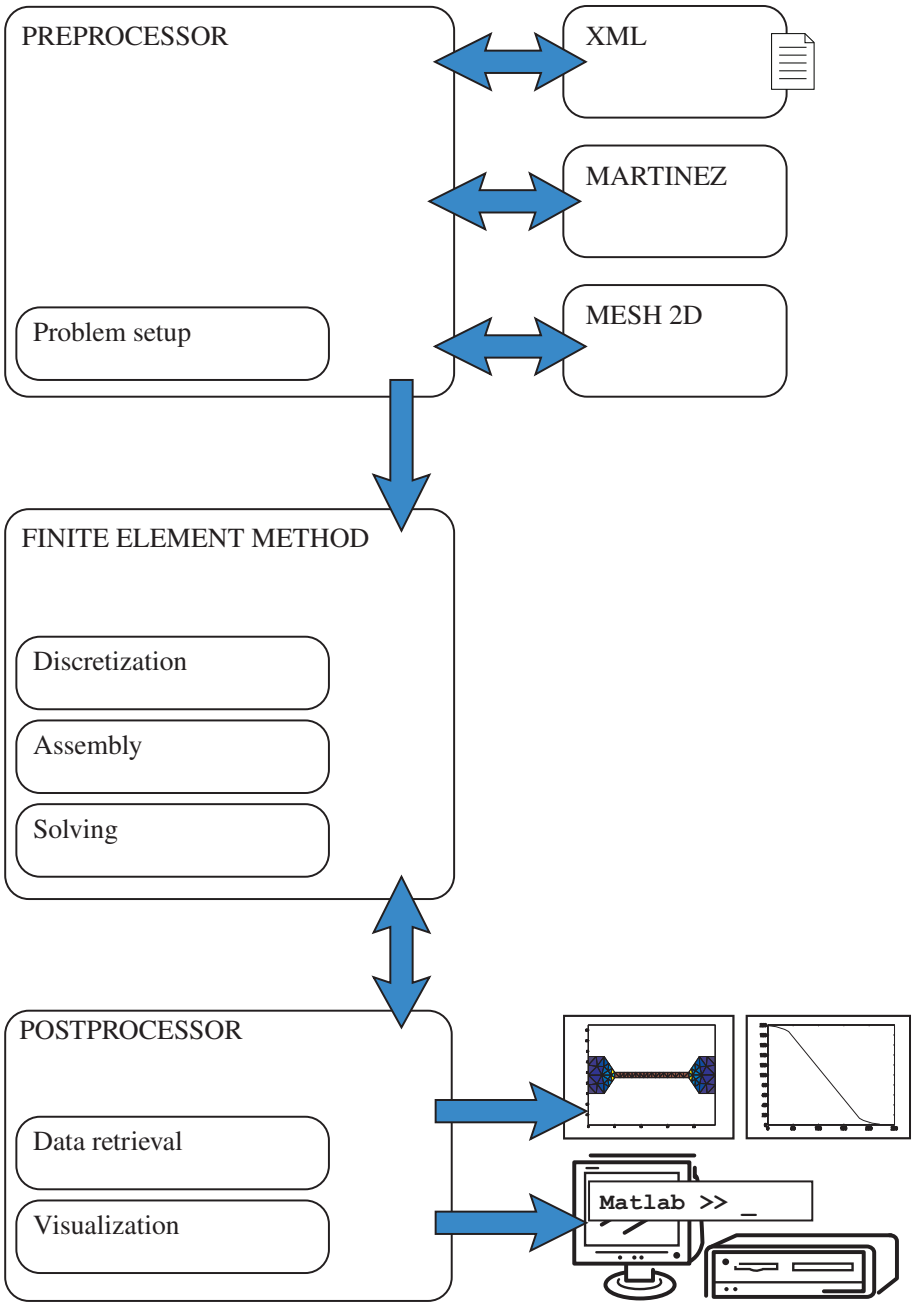

Figure 5.1.: Overview of FEMtastic modules

Finite Element Method The finite element method is the next block in the application flow. It takes the geometry data provided by the preprocessor as its input. In a first step the input data gets discretized. That is, for each node of the triangulation a *node object* is created and for each triangle a *finite element object* is created. Both types of objects are linked to each other appropriately and initial computations (e.g. the Jacobian matrices) are done.

In the following assembly step the stiffness matrix and the load vector are created by evaluating the integrals as shown in 4.4.1 using *numerical quadrature*. While Neumann boundary conditions are considered implicitly within the FEM system of equations, Dirichlet boundary conditions are worked in using *homogenization* as discussed in 4.4.4.

Solving the system of equations is the last duty of the finite element method. As mentioned in the introduction of this thesis this is completely delegated to MATLAB inbuilts. The solution are the unknown node potential values. These officiate as the input of the final postprocessing step.

Postprocessor The postprocessor, again, can be seen as an interface between the framework and the user. It assists in interpreting the results and preparing data for further computations. A set of routines provides data acquisition and visualization. Though the postprocessor is a self-contained module, it accesses methods implemented within the FEM module. This is useful as some functionality can be easily programmed e.g. into finite element objects and do not need to be duplicated as part of the postprocessor module.

Each module illustrated in 5.1 is realized as a MATLAB object. The necessary routines are executed upon the object and it is passed to the next one. A complete application workflow is shown with the next listing:

Listing 5.1: FEMtastic workflow

```
% example.m

% create modules
PREPROC  = PREPROCpack.PREPROC();
FEM      = FEMpack.FEM();
POSTPROC = POSTPROCpack.POSTPROC();

% Preprocessor, process XML file
PREPROC = ProcessXMLFile(PREPROC, 'example.xml');

% FEM, discretize, assemble and solve
FEM = discretize(FEM, PREPROC); % PREPROC -> FEM
FEM = assemble(FEM);
FEM = solve(FEM);

% Postprocessor, plot a graph
POSTPROC = init(POSTPROC, FEM); % FEM -> POSTPROC
plotValue(POSTPROC);
```

The creation and initialization of the three modules is done on lines 2 to 4. The next command on line 9 handles an XML file to the preprocessor and instructs it to *process the file*. The execution steps of the first module are now finished and on line 12 one can see how the state of the preprocessor is passed to the FEM object. There, the *discretization*, *assemble* and *solve* routines are executed and the state is passed to the postprocessor as shown on line 17. Finally, the last command `plotValue()` tells the postprocessor to create a graph of the solution and present it to the user.

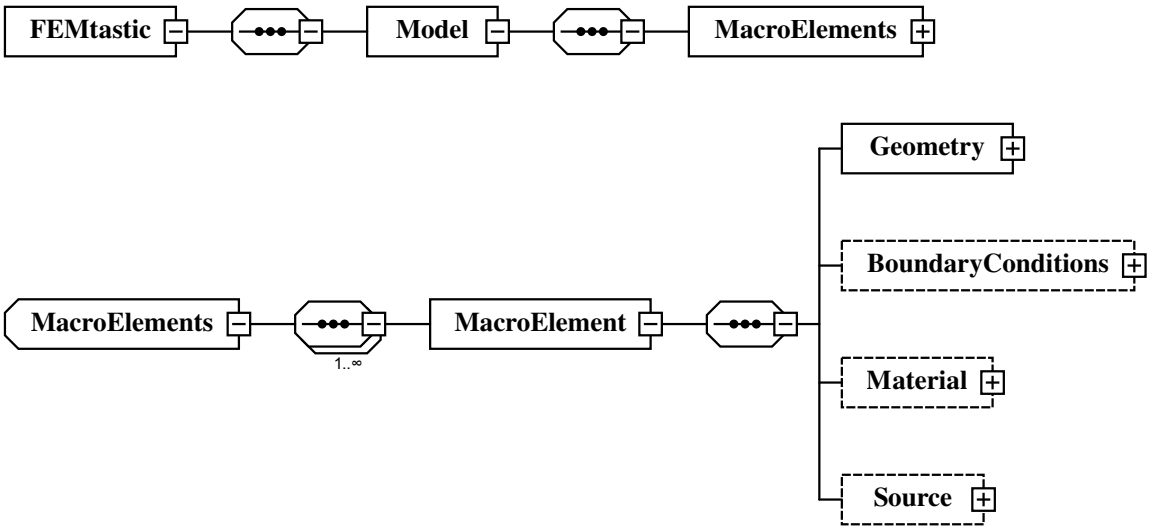

Figure 5.2.: FEMtastic XML schema definition

5.2. Input handling – XML, XSD

An input XML file is used in order to define the actual problem that is subject to be solved. The structure of this file is defined within the *XML schema definition file FEMtastic.xsd*. Figure A.1 in appendix A.1 on page 112 shows the complete XML tree.

The root element must have the name *FEMtastic* and is shown in figure 5.2. Its children are one or more *models* (problems) to be solved. Indeed, FEMtastic supports only one model at the current development state. Of course, splitting up different models to different input XML files is a workaround.

A model furthermore is composed of *MacroElements* (plural!). This is a set of containers that hold at least one *MacroElement* (singular!). Typically a problem will be set up with one *MacroElements* container. The motive of having more than one container is the combination of a set of macro elements that share common properties. This has not been implemented so far.

Eventually, a *MacroElement* is set up of four properties:

- A polygon defines the **geometry** of the macroelement.
- The **boundary conditions** set up properties on the border (one or more edges) of the macroelement.
- Depending on the type of problem the **material** defines
  - the permittivity ϵ of dielectric material in electrostatic,
  - the electrical conductivity σ of conducting material in static current flow,
  - the permeability μ of magnetic material in magnetostatic or
  - the thermal conductivity k for thermal problem setups.
- The **source** is used to define volume charge densities in electrostatic problems.

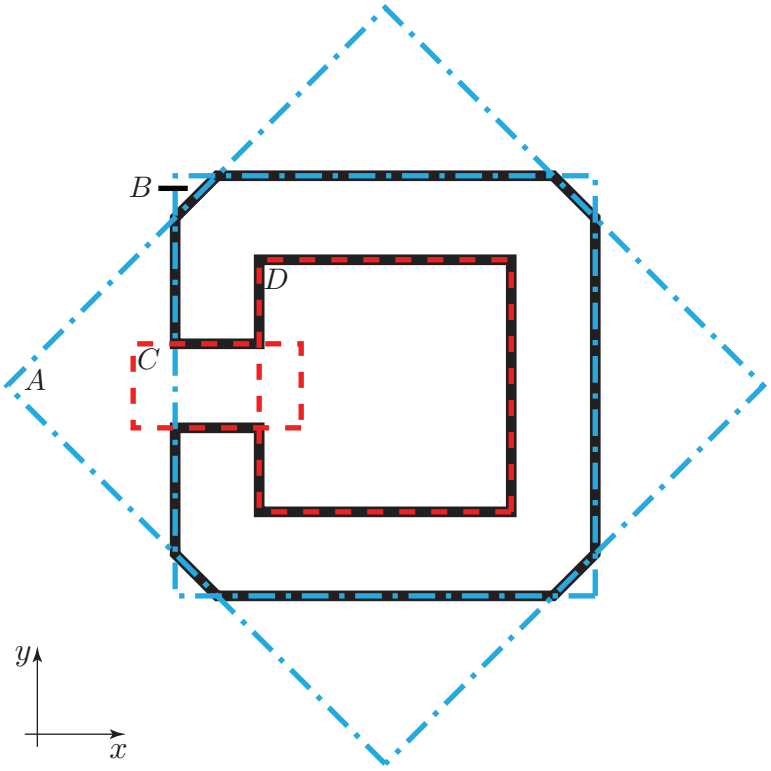

Figure 5.3.: Iron core with air gap

5.2.1. Geometry

To understand how any two-dimensional geometry can be defined, it is advisable to give a description in this own subchapter.

Figure 5.3 shows an iron core with air gap that was constructed by applying boolean operation to four rectangles (polygons). The complete setup without any boundary conditions or material properties can be found in appendix A.2 on page 113.

The figure shows four rectangles A, B, C and D. The iron core is the result of applying the rule $((A \cap B) \setminus C) \setminus D$. In words: Intersection of A and B minus C minus D. The appropriate XML statement is

Listing 5.2: 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 boolean operations **Union** (<*Add*>), **Difference** (<*Subtract*>) and **Intersection** (<*Intersect*>). The boolean operations can be nested. The only restriction is, that the <*Geometry*> entry must have exactly one child that is <*Add*>, <*Subtract*>, <*Intersect*> or <*Polygon*>.

The boolean operations are evaluated from top to bottom, hence

Listing 5.3: Evaluation from top to bottom

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

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

Listing 5.4: 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 is

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

Polygon The polygon directive is simply a set of two-dimensional coordinates that define the nodes of the polygon.

This defines an arbitrary triangle:

Listing 5.5: Defining a triangle

```
<Polygon>
  <Coords>
    <Coord x="-53.2" y="106" />
    <Coord x="44" y="121.803" />
    <Coord x="-16" y="199.20" />
  </Coords>
</Polygon>
```

Note: Polygons must have at least three nodes – needless to say. The last node defined is automatically connected to the first one. The number of nodes is unlimited.

Figure 5.4 shows the XML schema definition for the <*Geometry*> entry.

5.2.2. Boundary Conditions

FEMtastic can handle *Dirichlet*, *Neumann* and *Cauchy boundary conditions*.

Listing 5.6: Defining 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>
```

This listing defines the boundary conditions for the heating device shown in figure 5.5. The edges of the macroelement are numbered counterclockwise. Edges 1, 2 and 8 are the electrodes of the

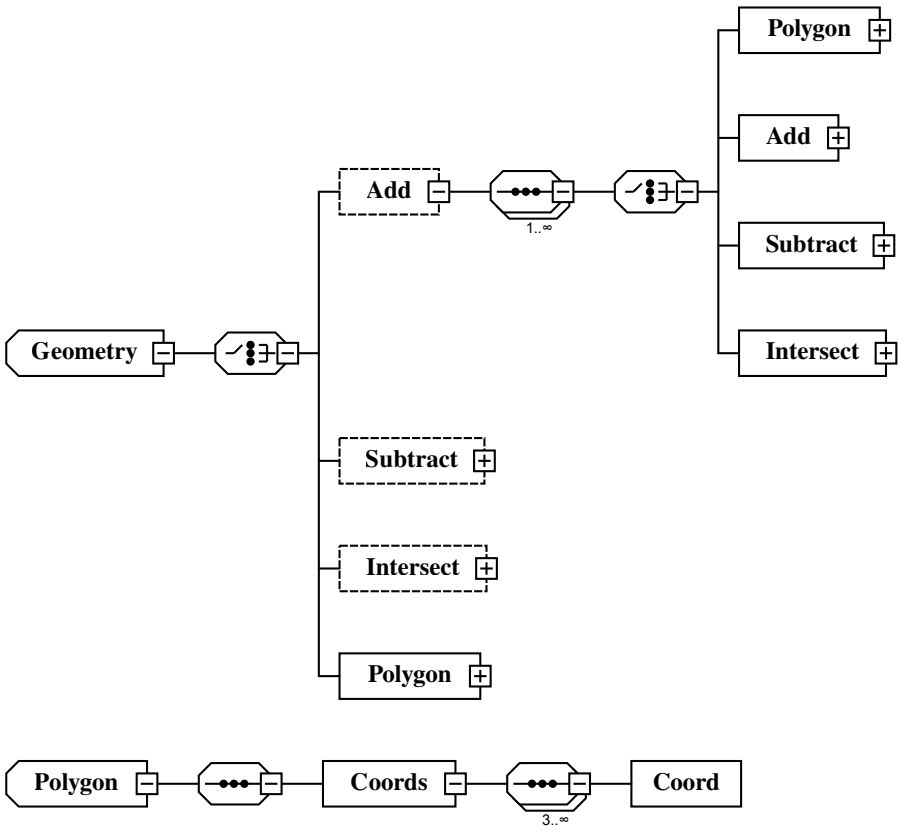

Figure 5.4.: XML schema definition of the geometry

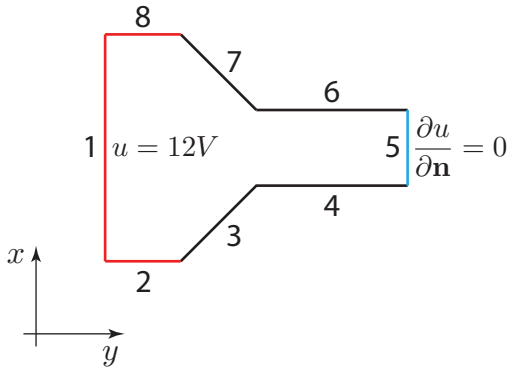

Figure 5.5.: Edges and boundary conditions

device and their electric potential is defined with $12V$ (Dirichlet boundary condition).

The second boundary condition defined, is a Neumann boundary condition for edge 5. It might represent a symmetry surface.

Note: You can define a range of edges with the attributes *from* and *to*. Because a polygon must always be closed the *from*-value might be bigger than the *to*-value. In this case all edges until the last one and beginning from the first one are accounted.

The next listing defines a Cauchy boundary condition. These are usually of interest only for thermal problems. Beneath the <*Value*> entry an additional <*Transition*> entry is defining the transition coefficient α of a Cauchy boundary condition.

Listing 5.7: Defining a Cauchy boundary condition

```
<BoundaryConditions>
  <BoundaryCondition type="Cauchy" from="3" to="4">
    <Value>
      0
    </Value>
    <Transition>
      5
    </Transition>
  </BoundaryCondition>
</BoundaryConditions>
```

5.2.3. Material

Defining the material property for a macroelement yields the following piece of XML code:

Listing 5.8: Material definition

```
<Material>
  <Value>
    29.0
  </Value>
</Material>
```

If the problem is of electrostatic type this might represent wet soil with a relative permittivity $\epsilon_r = 29$. For static current flow setups defining a value of `36.59E6` would represent aluminum with an electrical conductivity $\sigma = 36.59 \cdot 10^6 S/m$.

This table summarizes the intent of the value corresponding to the problem type:

problem type	**symbol**	**interpretation**	**default value**	**unit**
electrostatic	ϵ_r	relative permittivity	1.0	*non-dimensional*
magnetostatic	μ_r	relative permeability	1.0	*non-dimensional*
static current flow	σ	electrical conductivity	1.0	S/m
thermal	k	thermal conductivity	1.0	$W \cdot K^{-1} \cdot m^{-1}$

Table 5.1.: Problem types and their parameter intent

5.2.4. Source

The last property that can be defined for a macroelement is its source. If the *<Source>* entry is defined and different from 0 the macroelement represents a volume charge density for electrostatic problems or a thermal source. The unit of the value defined is $\frac{Q}{m^2}$.

Listing 5.9: Source definition

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

Figure 5.6 shows the XML schema definition for the *¡MacroElement¿* entry.

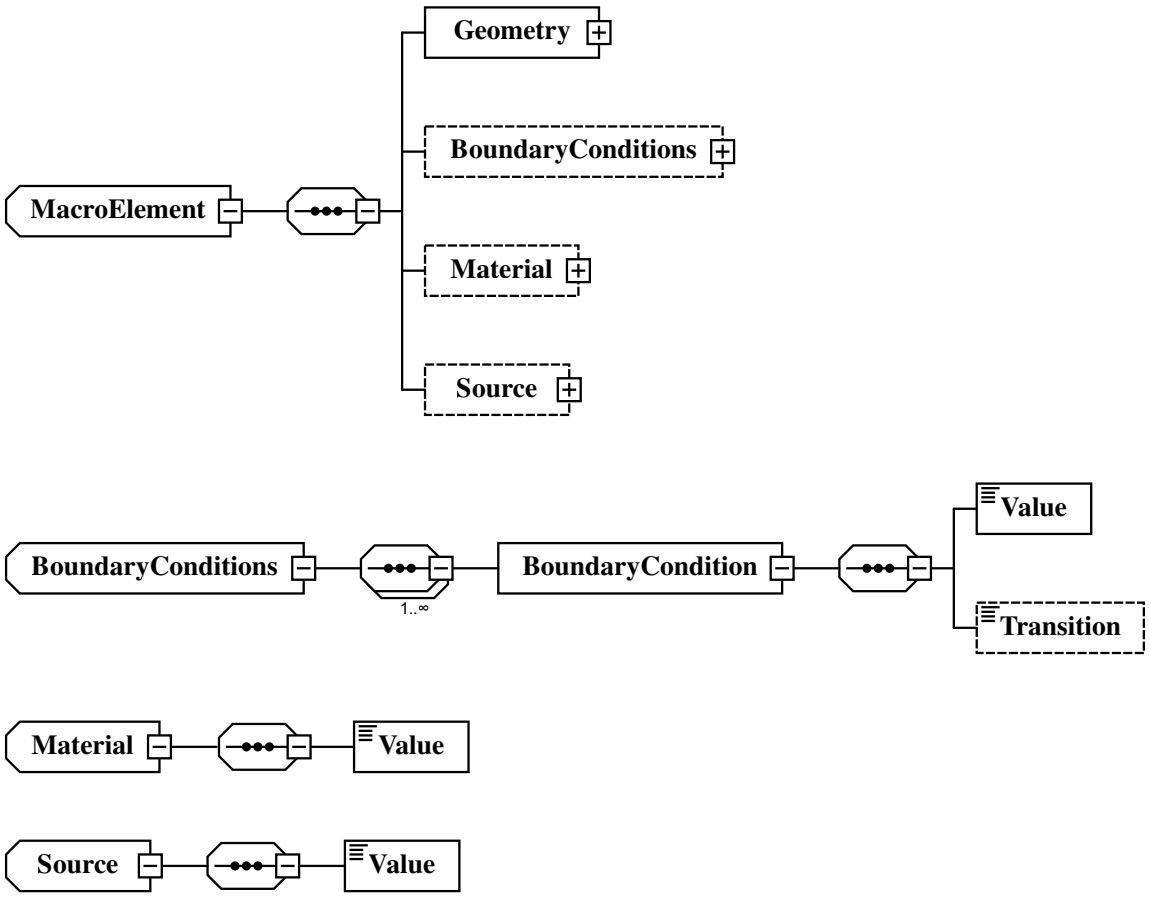

Figure 5.6.: XML schema definition of a MacroElement

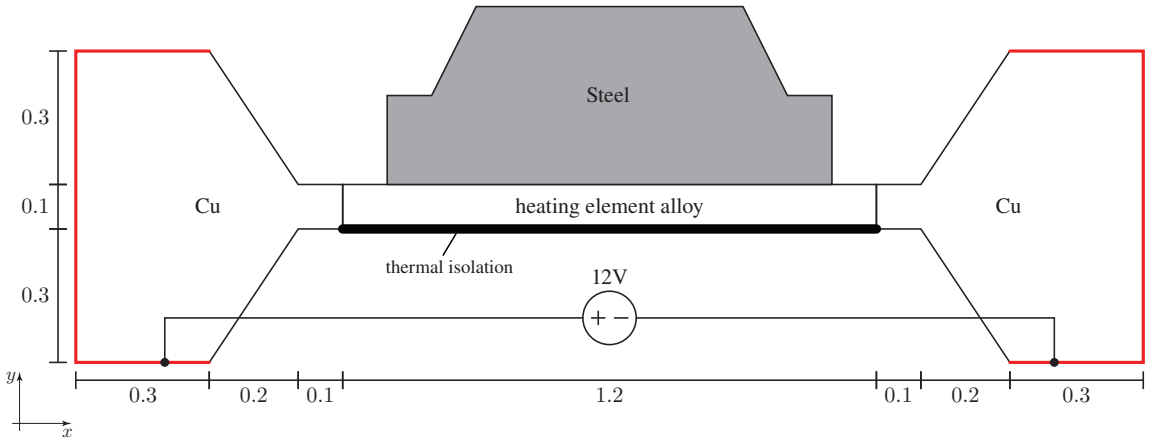

Figure 5.7.: Example heating device and workpiece

5.3. Object oriented model

The next sections describe each of the FEMtastic modules in particular as shown in figure 5.1 on page 36. The upcoming section 5.4 starts with the *preprocessing module* called `PREPROCpack`. The modules are organized in MATLAB packages. These are a summarization of classes comparable to Java packages. MATLAB organizes packages into folders starting with the +-sign. All classes within the package are saved to folders starting with the `@`-sign. Hence, `+FEMpack/@FiniteElement` is the path to the `FiniteElement`-class that belongs to the `FEMpack`-package.

Figure 5.15 on page 59, for example, shows the UML class diagram of the `FEMpack` package which realizes the *finite element method module*. Each of the classes of all packages is shown with a brief description on the following pages and the most important methods and attributes are discussed. There is no way around to also studying the code which is well documented and gives additional important information.

For an insight view of how FEMtastic works an example from industry will accompany this chapter.

> **Example – Hardening device:** The process of hardening steel components is commonly done by the method of annealing. A workpiece is heated up to, and kept, at a certain point of temperature in order to effectuate desirable material properties.
> Figure 5.7 shows the heating device with measurements in meters and a steel workpiece. The heating plate is made of an alloy that is energized by two copper electrodes. The electric connections are illustrated with red color. The voltage between the two electrodes is $12V$.
> The result that is asked for, is the static temperature distribution of the steel workpiece.

In order to gain the result, two subtasks will be performed:

- Setting up a *static current flow model*. This gives the current density and ohmic losses within the heating plate.
- The heat emission of the heating plate is a consequence of the ohmic losses due to the current density. This is simulated with a second *static thermal model* that will have the temperature distribution within the steel workpiece as a result.

How this example can be modeled and computed with FEMtastic will be shown on the next pages. A kind of ”debug run” is performed and at certain ”breakpoints” the mechanisms of FEMtastic are explained in detail. The computation of the example is highlighted by the keyword **Example** subsequently.

5.4. Preprocessing – PREPROCpack

The three technically independent modules `PREPROCpack`, `XMLpack` and `MARTINEZ` together with the interface to the `mesh2d`-toolbox have been combined to form the actual *preprocessing* of FEMtastic.

MATLAB has not yet complete XML support. An important feature – *XPath* – is missing. This mechanism is used to address nodes in an XML tree. The XML tree is loaded into MATLAB using the `xmltree` toolbox by Guillaume Flandin [10]. The XMLpack combines the toolbox with an implementation of the missing XPath functionality and serves as an interface between MATLAB and XML in general. It provides the necessary methods in order to read the XML input file, validate it against the *FEMtastic.xsd* schema definition file and assist the preprocessor in interpreting the problem setup.

The preprocessor state after reading the input XML file and processing it is that it knows about macroelements and their properties and polygons that set up the geometry of the macroelements. The polygons though have not yet been pieced together to form the macroelements. This is done by the excellent Martinez algorithm that has been implemented for MATLAB by means of the `MARTINEZ` package. (The suffix `pack` has been omitted as it is a standalone implementation and might also be used for different purposes outside FEMtastic.) It performs the already discussed boolean operations union, difference and intersection on the polygons as defined in the XML file and outputs the geometry of a completed macroelement. This output, in fact, is a sequence of coordinates that represent the closed boundary of the discretized domain.

Note: The Martinez algorithm is applied to polygons only. The output of the Martinez algorithm is one completed macroelement. Currently, the user is responsible to define macroelements in a way so they **do not overlap** and yet **build up a consistent geometry**.

The final preprocessing step is done by the `mesh2d` toolbox that has also been introduced so far. It computes a triangulation for the geometry output of the Martinez algorithm and as a result provides a set of nodes and a set of coherences. Latter one build up a consistent triangulation as demanded by equations (4.45) and (4.46) on page 28. The `mesh2d` toolbox supports so called *faces*. Each macroelement will be represented by exactly one face. This ensures that boundary nodes of two macroelements are coherent. Figure 5.9 points out an invalid and a valid triangulation of two faces whereupon the red line separates two faces (macroelements).

After the preprocessor has successfully finished its work it has saved all macroelements and their properties as well as the triangulation. This information is passed to the `FEMpack`.

Example Simulating the first subtask – the static current flow model – involves two files:

- *hardening_device.m*
- *hardening_device-currentflow.xml*

These can be found in directory *hardening_device*. The XML input file sets up the geometry and properties of the hardening device only. The steel workpiece is insignificant for the static current flow model as only the current density within the heating plate is of interest.

File *hardening_device-currentflow.xml* sets up the geometry as shown in figure 5.10.

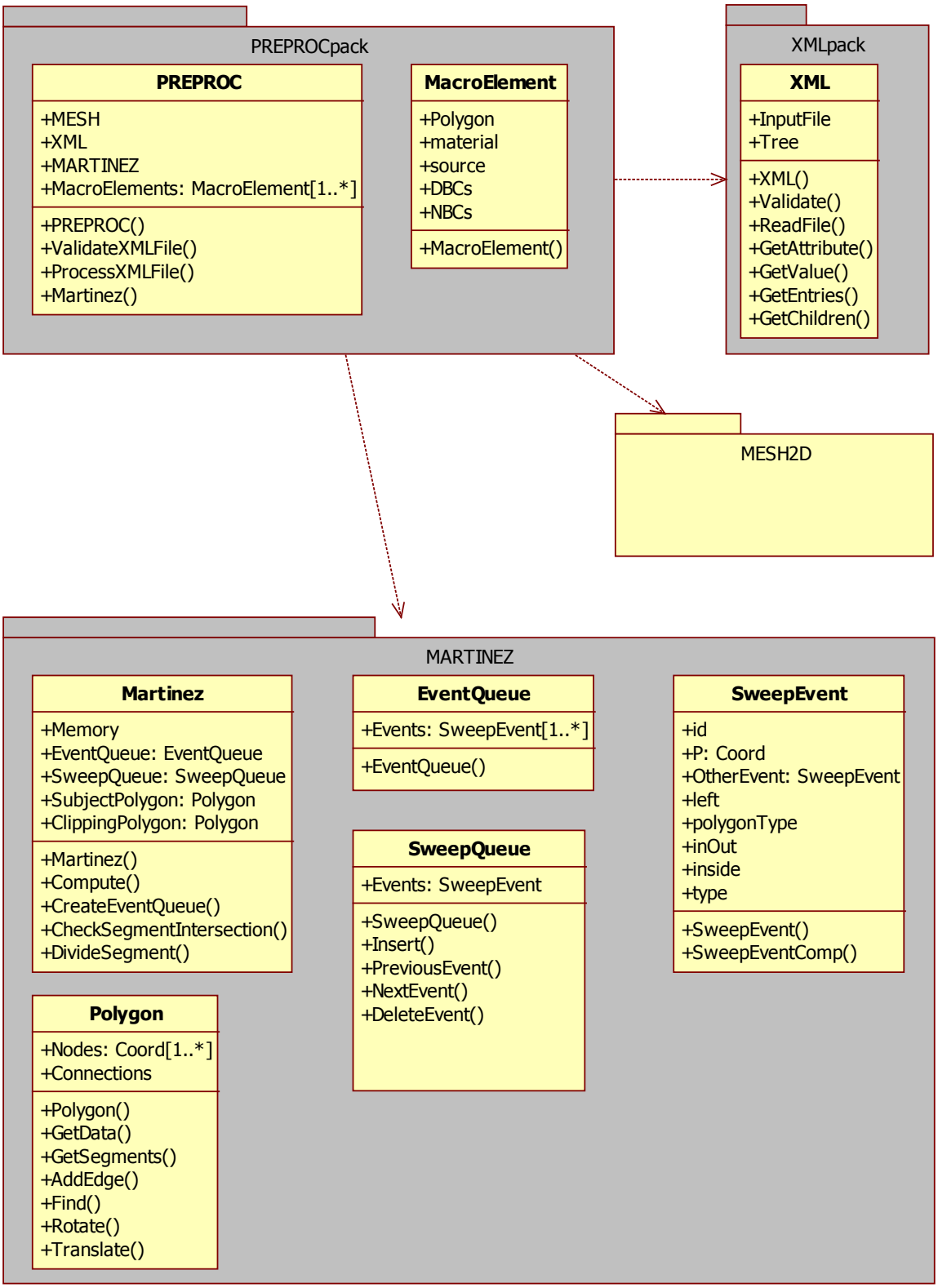

Figure 5.8.: PREPROCpack UML class diagram

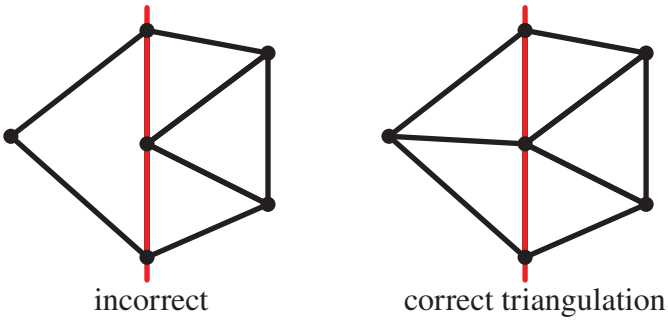

Figure 5.9.: Invalid and correct triangulation

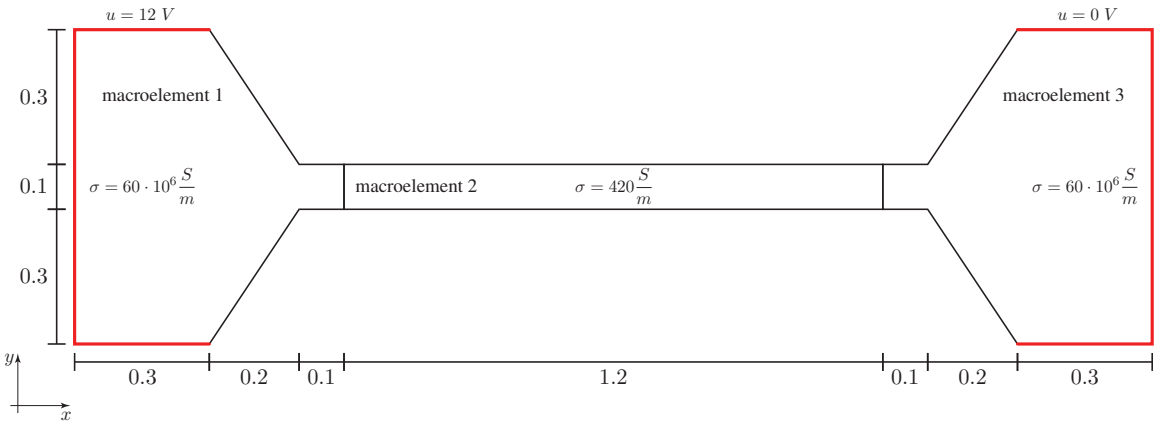

Figure 5.10.: Macroelements of the hardening device

Three macroelements build up the problem. Macroelements 1 and 3 represent the electrodes, macroelement 2 models the heating plate of the device. The electric conductivity is $\sigma = 60 \cdot 10^6 S/m$ for the copper electrodes and $\sigma = 420 S/m$ for the heating plate made of nickel.

The MATLAB file *hardening_device.m* contains control statements like `clc` or `clear all` as well as file system path declarations. Furthermore a debugging backend has been implemented which outputs messages about the current operations performed by FEMtastic. These debugging instructions are done by the command `be(...)` (backend). Because the reader is assumed programming experienced, these instructions are left uncommented.

When starting a run of the MATLAB file, the first instructions of interest are:

Listing 5.10: Module initialization

```
1 PREPROC = PREPROCpack.PREPROC();
2 FEM = FEMpack.FEM();
3 POSTPROC = POSTPROCpack.POSTPROC();
```

These three commands – as already mentioned on page 38 – initiate the *preprocessor*, *FEM* and *postprocessor modules* and assign the module objects returned by the constructor within the class definitions to the variables `PREPROC`, `FEM` and `POSTPROC` respectively. User interaction is usually done using one of these objects. For example `plotValue(POSTPROC)` will show a plot of the results of a successful simulation.

Listing 5.11: Validating and processing the XML input file

```
4 [valid, ME] = ValidateXMLFile(PREPROC, xml_file, schema_file);
5 PREPROC = ProcessXMLFile(PREPROC, xml_file);
```

The first command, the preprocessor should execute, is `ValidateXMLFile()`. It validates the XML input file *hardening_device-currentflow.xml* against the XML schema definition file *FEMtastic.xsd*. It is advisable to not continue working with an erroneous XML input file as the XML module relies on a validated input stream. The second command in the above code requests the preprocessor to prepare the input data in order to be used by the FEM module. If this command successfully terminates the preprocessor's work is completed. A deeper look to what this command does is appropriate.

5.4.1. PREPROC::ProcessXMLFile()

This command interprets the input XML file and saves the data to the preprocessor's internal properties. The most important ones are illustrated in figure 5.8.

These are:

Listing 5.12: Preprocessor properties

```
classdef PREPROC < handle
  properties
    % References to other modules
    XML;
    MARTINEZ;

    % Internal properties
    MacroElements;

    % ... %
  end

  % ... %
end
```

- `XML`: A reference to the the XMLpack module.
- `MARTINEZ`: A reference to the Martinez algorithm toolbox.
- `MacroElements`: This is an array of references to `MacroElement` instances.

There is no special interface to the `mesh2d` toolbox as only the `meshfaces` command is needed and called immediately where required.

For every macroelement defined in the input XML file one `MacroElement` object is created and saved to the internal `MacroElements` array of the preprocessor. Some declarations of the `MacroElement` class outline the most important properties:

Listing 5.13: Macroelement properties

```
classdef MacroElement < handle
  properties
    Polygon;

    material;
    source;

    DBCs;
    NBCs;
    CBCs;

    % ... %
  end

  % ... %
end
```

- `Polygon`: A polygon object that saves the geometry of the macroelement. This is **one** polygon. It might be the result of multiple polygons, to which the Martinez algorithm was applied to, before.
- `material`: Holds the material tensor which is a 2×2 tensor that can define isotropic as well as anisotropic material. Currently the input XML file only allows defining isotropic materials. The copper electrodes from the example will internally be treated as:

$$\Lambda = \begin{bmatrix} 60E6 & 0 \\ 0 & 60E6 \end{bmatrix} \tag{5.1}$$

- `source`: Saves the source density of the material defined by the macroelement. No source has been defined for the example macroelements, hence:

$$\rho = 0 \tag{5.2}$$

- `DBCs`: An array holding Dirichlet boundary conditions of the macroelement.
- `NBCs`: Another array for Neumann boundary conditions.
- `CBCs`: And an array for Cauchy boundary conditions.

Note: Different material properties of the problem must be modeled using different macroelements. The properties defined for one macroelement are valid and constant for the whole (sub)domain of the macroelement. MATLAB allows the definition of so called anonymous functions (or function handles). Using these will make it possible to define macroelements with varying properties. Unfortunately this is a desperate slow mechanism in MATLAB's current version and has been disabled because of its "un"-usability. Another restriction is the use of isotropic materials only. This is because the XML engine of FEMtastic does not interpret anisotropic material tensors at present.

The first important commands of the `ProcessXMLFile()` method are:

Listing 5.14: Processing the model

```
% scope: PREPROC::ProcessXMLFile()

Models = GetEntries(this.XML, '/FEMtastic/Model');
this = ProcModel(this, Models{1});
```

This loads the <Model> ... </Model> XML subtree and processes it. The `ProcModel()` routine contains the `ProcMacroElements()` routine which calls the `ProcMacroElement()` routine, and so on. Working off the subtree is demonstrated by table 5.2 on page 55.

After working off the XML tree FEMtastic has all information provided by the XML input file saved to its internal object properties. The last step in responsibility of the preprocessor is to perform the triangulation of the geometry. This is done with the help of the `mesh2d` toolbox.

The sample geometry in figure 5.11 has two faces (macroelements) and is used to demonstrate the input arguments of the `mesh2d` toolbox. A demo-file can be found in folder *mesh2d_demo*. The toolbox is called using the command

Listing 5.15: Meshing the geometry

```
[p, t, f] = meshfaces(nodes, edges, faces)
```

`ProcModel()` ← <Model type=' electrostatic ' refinement='0.5'>

The `ProcModel()` routine saves the `type` and `refinement` attributes. The `type` attribute is a string either being *'electrostatic'*, *'magnetostatic'*, *'currentflow'* or *'thermal'*. It is passed to the *FEM module* as is. Currently it only influences diagram descriptions. The `refinement` attribute is passed to the `mesh2d` toolbox and controls the density of the mesh. It is of type double where 1.0 is the default value. A value of 0.5 yields a finer mesh, 2.0 a coarse one.

`ProcMacroElements()` ← <MacroElements>

The <MacroElements> (plural) currently is a "container" of at least one <MacroElement> (singular) entry. In later versions it might be used to define properties common over all macroelements.

`ProcMacroElement()` ← <MacroElement>

This procedure creates an object for each macroelement as shown on the previous page.

`ProcGeometry()` ← <Geometry>

This routine creates the geometry of one macroelement. It takes use of the Martinez algorithm. The creation is done by interpreting a nesting of <Add>, <Subtract>, <Intersect> and <Polygon> statements as already shown in 5.2.1 on page 40.

`ProcBoundaryConditions()` ← <BoundaryConditions>

In analogy to the `<MacroElements>` container this currently holds an enumeration of possible `<BoundaryCondition>` entries. In later versions it is also supposed to define common properties over all boundary conditions it contains.

`ProcBoundaryCondition` ← <BoundaryCondition>

For the three possible types of boundary conditions (Dirichlet, Neumann, Cauchy) this routine adds an entry to the macroelement's `DBCs`, `NBCs` or `CBCs` arrays that will hold all boundary conditions defined for the current macroelement.

`ProcMaterial()` ← <Material>
Saves the material value to the macroelement's `material` property.

TODO: FEMtastic does already handle anisotropic material. The input XML file definition should be enhanced to also support the definition of tensors.

`ProcSource` ← <Source>
Saves the source value to the macroelement's `source` property.

Table 5.2.: Processing the XML input file

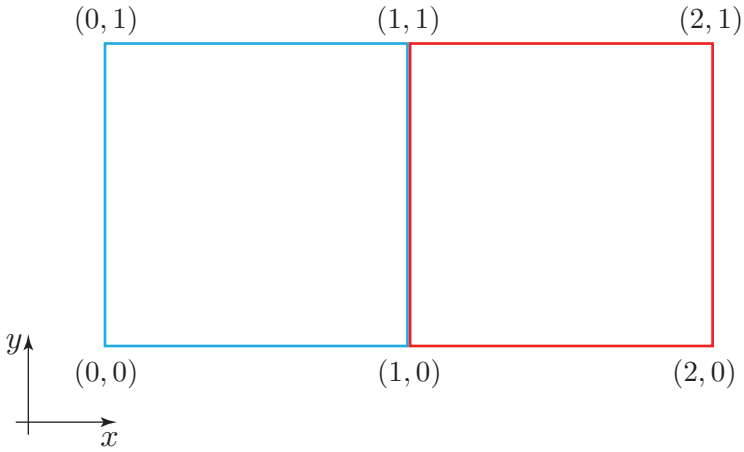

Figure 5.11.: Geometry containing two faces

The input arguments are:

- `nodes`: An $(n \times 2)$ matrix containing all n nodes of the whole geometry. For the geometry of figure 5.11 this is:

```
nodes = [0 0;
         1 0;
         1 1;
         0 1;
         2 0;
         2 1];
```

- `edges`: A $(c \times 2)$ matrix defining the c edges by supplying their respective first and second node:

```
edges = [1 2; % edge 1, face 1
         2 3; % edge 2, face 1, ...
         3 4;
         4 1;
         2 5; % edge 1, face 2
         5 6; % edge 2, face 2, ...
         6 3;
         3 2];
```

- `faces`: A cell array that contains another array for each face and has the edges that belong to that face:

```
faces = {[1 2 3 4]   % face 1
         [5 6 7 8]}; % face 2
```

The result of the triangulation is shown in figure 5.12 and the output arguments `p`, `t` and `f` are:

Listing 5.16: Preprocessor output

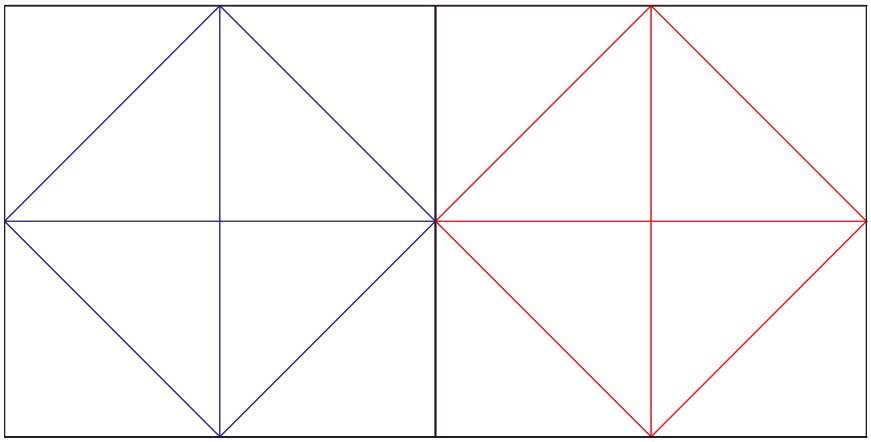

Figure 5.12.: Output of the `mesh2d` toolbox

p =		t =			f =
0	0	2	5	4	1
0	0.5000	1	5	2	1
0	1.0000	4	5	8	1
0.5000	0.5000	8	5	7	1
0.5000	0	6	2	4	1
0.5000	1.0000	3	2	6	1
1.0000	0	4	8	6	1
1.0000	0.5000	6	8	9	1
1.0000	1.0000	8	10	11	2
1.5000	0	7	10	8	2
1.5000	0.5000	11	10	14	2
1.5000	1.0000	14	10	13	2
2.0000	0	12	8	11	2
2.0000	0.5000	9	8	12	2
2.0000	1.0000	11	14	12	2
		12	14	15	2

The vertices of the triangulation are the first output argument `p` which is a $(m \times 2)$ matrix – in this case $m = 15$ vertices. The second argument `t` is a $(t \times 3)$ coherence matrix – e.g. the first row tells that points 2, 5 and 4 are the vertices of triangle (finite element) 1. The third output argument f has dimensions $(t \times 1)$ and contains the faces (macroelements) the triangles from t correspond to.

These three arguments together with the information saved within the `MacroElements` objects of the preprocessor are used by the `FEM` module in order to set up the finite elements with their nodes and boundary conditions.

Figure 5.13 shows the triangulation computed by the `mesh2d` toolbox for the heating device ex-

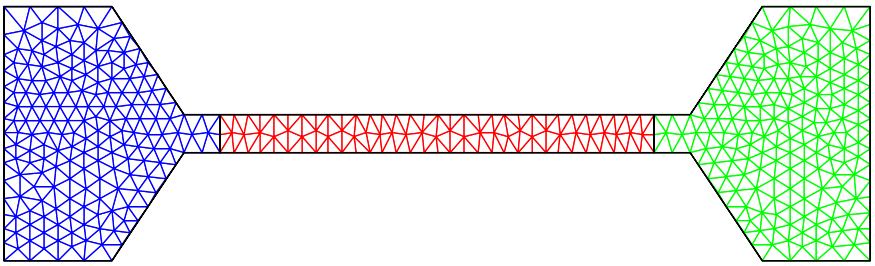

Figure 5.13.: Output of the `mesh2d` toolbox for the heating plate example

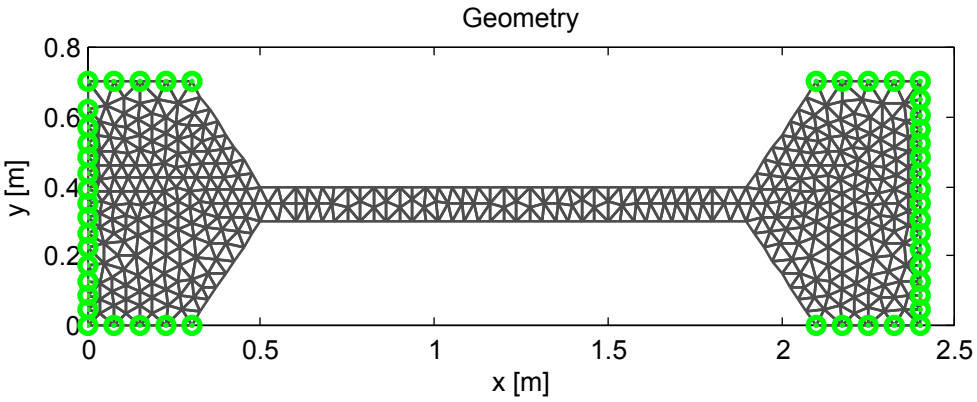

Figure 5.14.: FEMtastic geometry of the heating plate example

ample. The three macroelements (faces) are shown in three different colors. The geometry output of FEMtastic is shown in figure 5.14. The green nodes are those being part of the Dirichlet boundary defined.

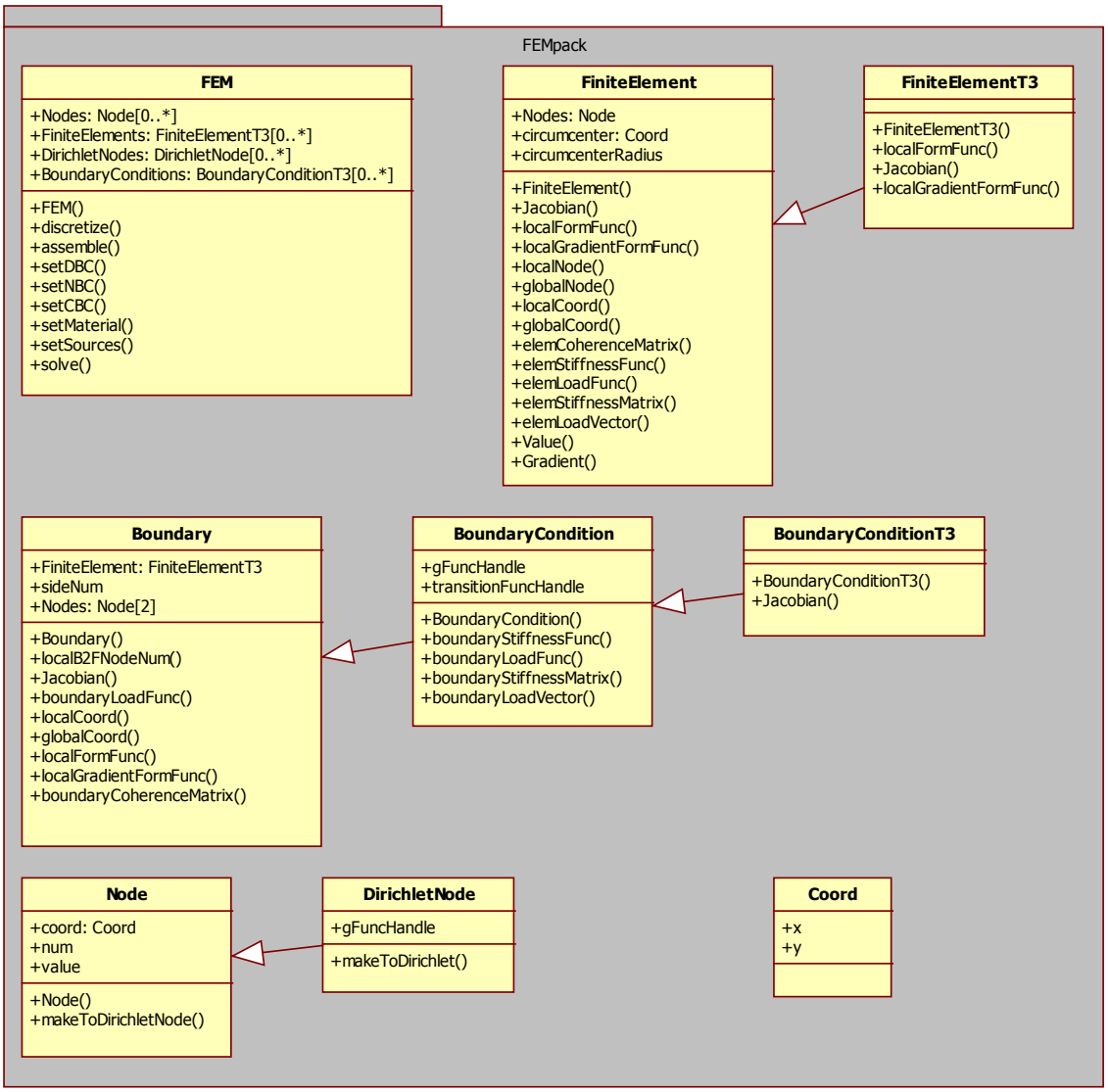

Figure 5.15.: FEMpack UML class diagram

5.5. Finite element method – FEMpack

The `FEMpack` does the actual work of finding a solution to the problem specified and constructed by means of the preprocessor. It uses the triangulation provided, applies the information saved as macroelements to the finite elements, assembles the system of equations and solves it. The output of this module is the value of all nodes of the triangulation. This is the electric potential for electrostatic and current flow, the magnetic scalar potential – also known as the magnetomotive force – for magnetostatic or the temperature for thermal problems.

The `FEM`-class is the de facto controller of the the module (package) and the interface to the preprocessor and to the user respectively. It holds the important functions `discretize()`, `assemble()` and `solve()` which do the basic steps of the finite element method.

Two further classes within this package together build up the data structure for the geometry rep-

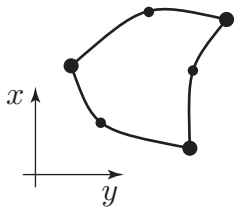

Figure 5.16.: Triangular shape finite element with quadratic form functions

resentation within the module. These are:

- `Node` and
- `FiniteElement`

Instances of the `Node` class certainly represent the point set of a triangulation and instances of the `FiniteElement` class the subdomain set. Currently the subdomains are triangles each having three vertices. This *generalization* is specified by the inherited `FiniteElementT3` class with **T3** meaning triangle with three vertices. The code also has the **T6** class of finite elements implemented. These are distorted triangles with quadratic form functions and set up by 6 nodes as shown in figure 5.16.

An instance of the `Node` class is simply an intersection point of the triangulation. A `Node` object will also contain the value of the solution. The `DirichletNode` class is an inheritance (specialization) of the `Node` class. A *node* can be converted to a *Dirichlet node* meaning that it is part of the Dirichlet boundary and its value is known a priori.
This is done using the `makeToDirichletNode()` routine of the `Node` object itself.

The `Boundary` class instances represent one side of a finite element each. Boundary conditions are inherited to the basic `BoundaryCondition` class that is meant to represent any boundary for any type of finite elements.

The `Coord` class also mentioned in figure 5.15 is implemented using a 2×1 MATLAB matrix holding the x- and $y-$ coordinate of a node.

5.5.1. Discretizing – FEM::discretize()

The description of the `FEMpack` module will be done by continuing the debug run that "halted" at the end of the successful preprocessing of the heating device example. The preprocessor's output are the three parameters `p`, `t` and `f` together with the properties stored in the `PREPROC.MacroElements` array that holds information about all macroelements. An example has been given on page 56 for the very simple geometry shown in figure 5.11. Figure 5.14 shows the FEMtastic geometry plot of the hardening device. It has 478 nodes which are related to 803 finite elements.

The next statement of the code within file *hardening_device-currentflow.m* transfers this information from the preprocessor to the FEM module and tells the latter one to do the discretization:

Listing 5.17: Interface between preprocessor and FEM module

```
10 FEM = discretize(FEM, PREPROC);
```

Although the actual geometric discretization has already been done by the meshing toolbox, "discretization" here means:

- creating a `Node` object for every vertex of the triangulation,
- creating a `FiniteElementT3` object for every triangle, linking it to the nodes and
- applying information stored within `PREPROC.MacroElements` to the corresponding objects.

The first two tasks are done by the subroutine `discretize_geom()` within `FEMpack.FEM` and are rather straight forward. This piece of pseudocode shall demonstrate what the routine does using the preprocessor output of listing 5.16 on page 56 as an example:

for all `p` **do**
  N ← new `Node` object
  N.x ← `p(1)`
  N.y ← `p(2)`
  Nodes{end + 1} ← N
end for
for all `t` **do**
  F ← new `FiniteElementT3` object
  F.Nodes{1} ← Nodes{`t(1)`}
  F.Nodes{2} ← Nodes{`t(2)`}
  F.Nodes{3} ← Nodes{`t(3)`}
  FiniteElements{end + 1} ← F
end for

An important point here is, that the **global numeration** of each node follows the output of the meshing toolbox. Each node has a unique global identification number and can be addressed within the FEM module as:

Listing 5.18: Addressing a finite element's nodes

```
FEM.Nodes{n}
```

This call returns a reference to the `Node` object of the node with global number n. The node's properties can be gathered by - e.g.:

Listing 5.19: Addressing a node's properties

```
FEM.Nodes{3}.x
FEM.Nodes{3}.y
FEM.Nodes{3}.value
```

Note: Within the MATLAB implementation of FEMtastic, for all arrays that hold references to other objects, MATLAB *cell arrays* are used. While typical arrays can hold values of type *double* only, a *cell* of a cell array can hold any type of object including another array or cell array. The items of a cell array are addressed using curly braces `{}`. This will also be considered by the upcoming pseudocode snippets.

The `value` property, of course, can only be queried if the solution has been computed or the node

is part of the Dirichlet boundary. This can be tested against

Listing 5.20: A Dirichlet node

```
FEM.Nodes{3}.isDirichlet
```

that returns a boolean *true* or *false* value.

Another important issue is, to have a look at the initialization of a finite element. This includes the

- computation of the **Jacobian matrix** $\mathbf{J}^{(r)}$ defined by equation (4.50) on page 29,
- computation of the **inverse Jacobian matrix** $\mathbf{J}^{-\mathbf{1}}_{(\mathbf{r})}$ and
- computation of the **Jacobi determinant** $\det \mathbf{J}^{(\mathbf{r})}$.

For T3 finite elements, these are invariants and therefor computed once for each finite element. Further initialization is done:

- Computation of the finite elements' *circumcenter* in global space and
- the *radius of the circumcircle*.

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

Storing the properties of the macroelements to the related finite elements is somewhat tricky. The result of the meshing algorithm gives information about the macroelement each finite element belongs to. Unfortunately it does not contain information whether a finite element is positioned on the edge or in the interior of the problem domain. Thus, applying boundary conditions involves testing every edge of the triangulation. Changing this behavior would demand an intervention into the meshing toolbox `mesh2d`. This has not been done so far.

Transferring the information stored within the macroelements to the corresponding finite elements is depicted with the next snippet of pseudocode:

```
for all M ← MacroElements do
  for all F ← M.FiniteElements do
    for all N1 ← F.Nodes do {This is N1 = 1, 2, 3 for T3 finite elements.}
      N2 ← next counter-clockwise Node {(N1, N2) = (1, 2), (2, 3) or (3, 1), the edges of the
      finite element}

      for all B ← M.DBCs do {These are the Dirichlet boundary conditions.}
        if N1 lies on B then
          makeToDirichletNode(N1, B.value)
        end if
      end for

      for all B ← M.NBCs do {Neumann boundary conditions.}
        if (N1, N2) is part of B then
          setNBC(F.N1, F.N2, B.value) {Adds current edge (N1, N2) as Neumann
          boundary condition.}
        end if
      end for

      for all B ← M.CBCs do
        {Do the same as with Neumann boundary conditions.}
      end for
    end for
  end for
end for
```

The routine iterates through all finite elements' edges and nodes using the object oriented model from top to bottom (macroelement → finite element → edge → node).

At this point the discretization is done. FEMtastic now "knows" all information provided by the XML input file and has stored it to the correct locations. Hence, the assembly of the system of equations can be performed.

5.5.2. Assembly – FEM::assemble()

Assembly means building up the FEM system of equations which is effectively done following the three steps:

- Building up the stiffness matrix without consideration of boundary conditions (equation (4.62), page 31).
- Building up the load vector without consideration of boundary conditions (equation (4.63), page 31).
- Integration of boundary conditions.

As a data structure for the $(n \times n)$ stiffness matrix $\mathbf{K}$ and the $(n \times 1)$ load vector $\mathbf{f}$ with n being the number of unknowns, MATLAB **sparse matrices** are used. These only save values different from zero. Because a large part of the entries of the stiffness matrix is zero, this is a storage saving data structure. Moreover, MATLAB supports the same matrix operations to both "typical" matrices and sparse matrices.

The assembly without consideration of boundary conditions is demonstrated by the next pseudocode snippet:

$\mathbf{K} \leftarrow \mathbf{0}$ {init}
$\mathbf{f} \leftarrow \mathbf{0}$ {init}
for r = 1 to R **do** {R = number of finite elements}
  get $\mathbf{C}^{(r)}$
  compute $\mathbf{K}^{(r)}$
  compute $\mathbf{f}^{(r)}$

  $\mathbf{K} \leftarrow \mathbf{K} + \mathbf{C}^{(r)T} \cdot \mathbf{K}^{(r)} \cdot \mathbf{C}^{(r)}$
  $\mathbf{f} \leftarrow \mathbf{f} + \mathbf{C}^{(r)T} \cdot \mathbf{f}^{(r)}$
end for

This is nothing more than the implementation of equations (4.62) and (4.63) on page 31. But a deeper insight into the computation of the element coherence matrix, element stiffness matrix and element load vector is appropriate at this point.

These computations take place (and can be found) within the functions

- `FEMpack::FiniteElement::elemCoherenceMarix()`
- `FEMpack::FiniteElement::elemStiffnessMatrix()`
- `FEMpack::FiniteElement::elemLoadFunc()`.

For demonstrating the results, figure 5.17 shows a representative illustration of a parallel plate capacity. Its modeling yields 13 nodes (black numbering) and 12 finite elements (red numbering). Dirichlet boundary conditions have been defined for the left and right border of the rectangular geometry, Neumann boundary conditions for the top and bottom border.

5.5.2.1. FiniteElement::elemCoherenceMatrix()

The element coherence matrix $\mathbf{C}^{(r)}$ is a rather simple one. Its dimensions are $(3 \times n)$ and it is defined by equation 4.61 on page 31.

The element coherence matrix of finite element 1 as shown in figure 5.17 is:

Listing 5.21: Coherence matrix of a finite element

```
Cr =
 0  0  0  0  0  0  0  1  0  0  0  0  0
 0  0  0  0  0  0  1  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  1  0  0  0
```

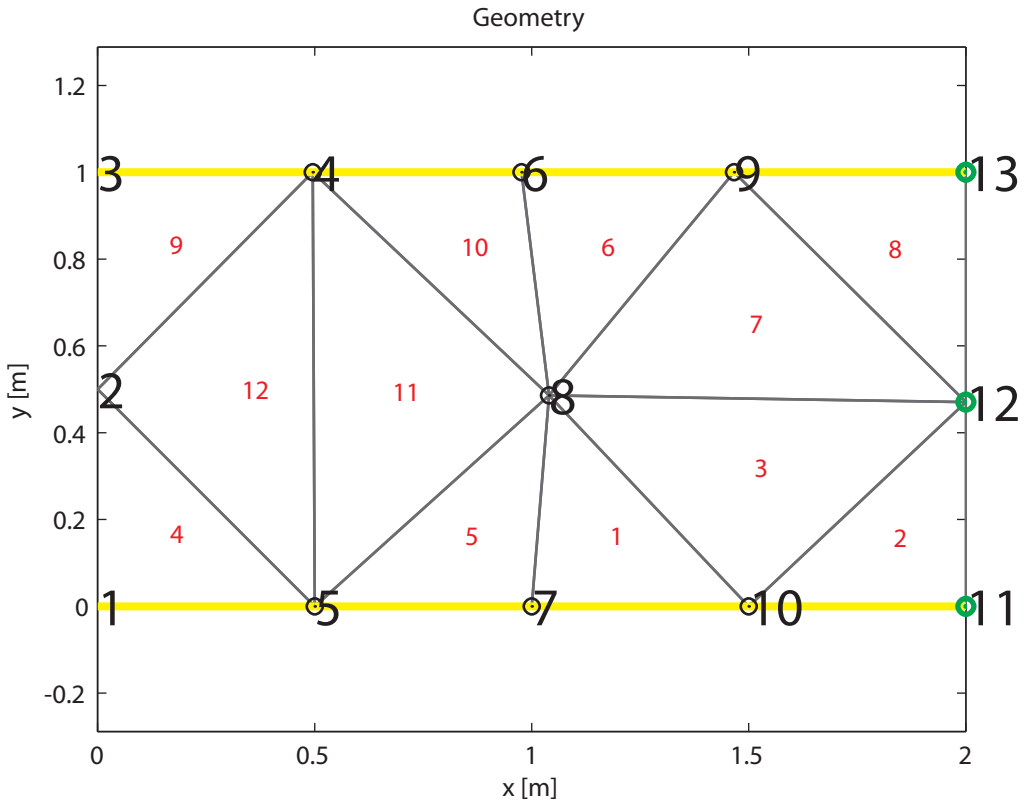

Figure 5.17.: Parallel plate capacity

The output tells that

- local node 1 (first row) corresponds to global node 8 (eighth column),
- local node 2 corresponds to global node 7 and
- local node 3 of finite element 1 corresponds to global node 10.

The computation of this matrix is trivial.

5.5.2.2. Numerical quadrature - QuadInt()

Before actually computing the stiffness matrix and load vector of the FEM system of equations, the application of numerical integration – as established by equation (4.71) on page 34 – is shown. Quadrature integration is used to approximate the integrals in (4.59) and (4.60) on page 31. Also the integral of the boundary condition term in (4.42) is evaluated using one-dimensional quadrature integration.

Figure 5.18 shows the different quadrature integration types that are supported by FEMtastic. The following table lists their configurations:

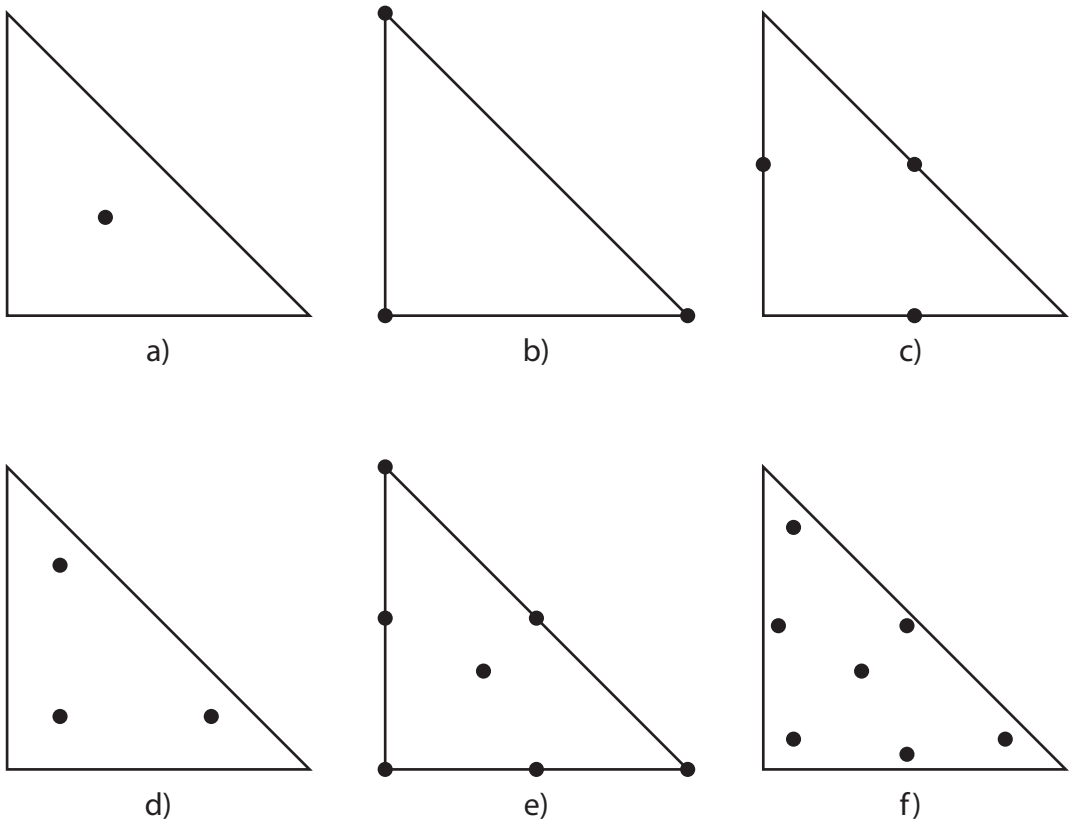

Figure 5.18.: Quadrature integration over the reference element

	supporting points (ξ_i, η_i)	**weight** w	**exact for polynomials of order**
a)	$(0.5000, 0.5000)$	0.5000	1
b)	$(0.0000, 0.0000)$	0.1667	1
	$(1.0000, 0.0000)$	0.1667	
	$(0.0000, 1.0000)$	0.1667	
c)	$(0.5000, 0.0000)$	0.1667	2
	$(0.5000, 0.5000)$	0.1667	
	$(0.0000, 0.5000)$	0.1667	
d)	$(0.1667, 0.1667)$	0.1667	2
	$(0.6667, 0.1667)$	0.1667	
	$(0.1667, 0.6667)$	0.1667	
e)	$(0.0000, 0.0000)$	0.0250	3
	$(1.0000, 0.0000)$	0.0250	
	$(0.0000, 1.0000)$	0.0250	
	$(0.5000, 0.0000)$	0.0667	
	$(0.5000, 0.5000)$	0.0667	
	$(0.0000, 0.5000)$	0.0667	
	$(0.3333, 0.3333)$	0.2250	
f)	$(0.1013, 0.1013)$	0.0630	5
	$(0.7974, 0.1013)$	0.0630	
	$(0.1013, 0.7974)$	0.0630	
	$(0.4701, 0.4701)$	0.0662	
	$(0.0597, 0.4701)$	0.0662	
	$(0.4701, 0.0597)$	0.0662	
	$(0.3333, 0.3333)$	0.1125	

Table 5.3.: Different configurations for numerical integration

The type that is used for FEMtastic currently must be hard-coded within file *QuadInt.m*. Because for T3 finite elements the basis functions are of first order, it is advisable to use type a) or b) as these give the exact integral evaluation and therefor do not additionally adulterate the approximated solution.

Listing 5.22: Quadrature integration of an anonymous function

```
v = QuadInt(w, dim)
```

Function `QuadInt()` requires two input arguments. The first argument `w` is a MATLAB function handle that allows to be evaluated as `w(xi, eta)`. The second argument `dim` defines the dimension of the quadrature integration. Two-dimensional evaluation is performed computing the stiffness matrix and load vector. One-dimensional evaluation is required for the boundary condition terms.

5.5.2.3. FiniteElement::elemStiffnessMatrix()

Equation (4.59) on page 31 defines the element stiffness matrix $\mathbf{K}^{(r)}$. The computation for each finite element is done as:

for α = 1 to 3 **do**
  for β = 1 to 3 **do**
    $\mathbf{K}^{(r)}(\alpha,\beta)$ = `QuadInt((` grad $\hat{f}_\alpha(\xi,\eta))^T \hat{\epsilon}^{(r)}$ grad $\hat{f}_\beta(\xi,\eta)$`,2)`
  end for
end for

The basis functions $\hat{f}_\alpha(\xi,\eta)$ (equation (4.48), page 29) and their gradients grad $\hat{f}_\alpha(\xi,\eta)$ are constructed with simple MATLAB functions:

Listing 5.23: Basis function

```
function phi = localFormFunc(localNodeNum, xi)
  switch localNodeNum
    case 1
      phi = 1 - xi(1) - xi(2);
    case 2
      phi = xi(1);
    case 3
      phi = xi(2);
  end
end
```

and

Listing 5.24: Gradient of a basis function

```
function gradphi = localGradientFormFunc(localNodeNum, xi)
  switch localNodeNum
    case 1
      gradphi = [-1; -1];
    case 2
      gradphi = [1; 0];
    case 3
      gradphi = [0; 1];
  end
end
```

Note: `xi` is a vector with `xi(1)` $= \xi$ and `xi(2)` $= \eta$.

For the numerical integration `QuadInt()`, `localGradientFormFunc()` is being evaluated at the supporting points (ξ_i,η_i) as shown prior.

For finite element 1 of the parallel plate capacity, the element stiffness matrix computes to:

Listing 5.25: Local stiffness matrix of a finite element

```
Kr =
    0.0257   -0.0237   -0.0021
   -0.0237    0.0461   -0.0224
   -0.0021   -0.0224    0.0245
```

5.5.2.4. FiniteElement::elemLoadVector()

In an analogous way the element load vector, that is defined by equation (4.60) on page 31, is computed:

for α = 1 to 3 **do**
$\mathbf{f}^{(r)}(\alpha)$ = `QuadInt(`$f(x(\xi),y(\eta))\hat{f}_\alpha(\xi,\eta)\left|\det J^{(r)}\right|$`,2)`
end for

Consider the difference between function f which is the *load* (resp. *source*) as defined by the problem setup and function $\hat{f}_\alpha$ which is the basis function for local node α.

Evaluating function f yields the *local-to-global transformation* $x^{(r)}(\xi,\eta)$ and $y^{(r)}(\xi,\eta)$. Because the source (or load) is static over a macroelement and therefor static over a finite element, this is trivial:

$$f(x^{(r)}(\xi,\eta), y^{(r)}(\xi,\eta)) = \rho \tag{5.3}$$

And ρ is stored within the finite element as its `source` value.

Note: As already mentioned earlier, *varying* macroelement properties can be developed using MATLAB function handles. This is also supported by FEMtastic but disabled at the moment because of function handle's inefficiency.

Because no sources have been defined for the parallel plate capacity the element load vector of finite element 1 (and all others) is:

Listing 5.26: Local load vector of a finite element

```
fr =
     0
     0
     0
```

After the computation of the element stiffness matrix and the element load vector it might be interesting to see, how these get integrated into the system stiffness matrix $\mathbf{K}$ and the system load vector $\mathbf{f}$.

The algorithm on page 64 shows the integration as

$\mathbf{K} \leftarrow \mathbf{K} + \mathbf{C}^{(r)T} \cdot \mathbf{K}^{(r)} \cdot \mathbf{C}^{(r)}$
$\mathbf{f} \leftarrow \mathbf{f} + \mathbf{C}^{(r)T} \cdot \mathbf{f}^{(r)}$

for a finite element r. Assuming an initialized empty matrix $\mathbf{K}$ and load vector $\mathbf{f}$ the first finite element is integrated:

Listing 5.27: Integrating the first local stiffness matrix

```
K = Cr' * Kr * Cr =
  0 0 0  *  0.0257  -0.0237  -0.0021  *  0 0 0 0 0 0 0 1 0 0 0 0 0
  0 0 0    -0.0237   0.0461  -0.0224     0 0 0 0 0 0 1 0 0 0 0 0 0
  0 0 0    -0.0021  -0.0224   0.0245     0 0 0 0 0 0 0 0 0 1 0 0 0
  0 0 0
  0 0 0
  0 0 0
  0 1 0
  1 0 0
  0 0 0
  0 0 1
  0 0 0
  0 0 0
  0 0 0
```

The result is:

Listing 5.28: System stiffness matrix with intetgrated local stiffness matrix

```
K =
  0  0  0  0  0  0   0        0       0   0       0  0  0
  0  0  0  0  0  0   0        0       0   0       0  0  0
  0  0  0  0  0  0   0        0       0   0       0  0  0
  0  0  0  0  0  0   0        0       0   0       0  0  0
  0  0  0  0  0  0   0        0       0   0       0  0  0
  0  0  0  0  0  0   0        0       0   0       0  0  0
  0  0  0  0  0  0   0.0461  -0.0237  0  -0.0224  0  0  0
  0  0  0  0  0  0  -0.0237   0.0257  0  -0.0021  0  0  0
  0  0  0  0  0  0   0        0       0   0       0  0  0
  0  0  0  0  0  0  -0.0224  -0.0021  0   0.0245  0  0  0
  0  0  0  0  0  0   0        0       0   0       0  0  0
  0  0  0  0  0  0   0        0       0   0       0  0  0
  0  0  0  0  0  0   0        0       0   0       0  0  0
```

The intent of this demonstration is to show the effects of the element coherence matrix $\mathbf{C}^{(r)}$ that "puts" the entries of the element stiffness matrix $\mathbf{K}^{(r)}$ (element load vector $\mathbf{f}^{(r)}$, respectively) to the right positions of the system stiffness matrix $\mathbf{K}$ (or load vector $\mathbf{f}$). The entries of the load vector are all 0 because no sources have been defined. Of course, the above result demonstrates the state after the first iteration through all finite elements.

5.5.2.5. Integration of boundary conditions

After the element stiffness matrices and element load vectors of all finite elements have been computed, the stiffness matrix $\mathbf{K}$ and load vector $\mathbf{f}$ of the FEM system of equations exist as if there where neither Dirichlet nor Neumann boundary conditions defined.

The integration of boundary conditions is done as depicted in section 4.4.4:

for e = 1 to B **do** {B = number of Neumann/Cauchy boundary conditions}
  get $\mathbf{C}^{(e)}$
  compute $\mathbf{f}^{(e)}$
  compute $\mathbf{K}^{(e)}$

  $\mathbf{K} \leftarrow \mathbf{K} + \mathbf{C}^{(e)T} \cdot \mathbf{K}^{(e)} \cdot \mathbf{C}^{(e)}$
end for

for i = 1 to n **do** {n non-Dirichlet nodes}
  correctionSum $\leftarrow 0$
  for j = n+1 to n+m **do** {m Dirichlet nodes}
    correctionSum $\leftarrow \mathbf{K}(i,j) \cdot a_j$
  end for
  $\mathbf{f}(i) \leftarrow \mathbf{f}(i)-$ correctionSum
end for

The computation of $\mathbf{K}^{(e)}$ and $\mathbf{f}^{(e)}$ is done analogously to the computation of $\mathbf{K}^{(r)}$ in 5.5.2.3 and $\mathbf{f}^{(r)}$ in 5.5.2.4 but using the one-dimensional integration terms of the edge reference element $\hat{E}$:

for α = 1 to 2 **do**
  for β = 1 to 2 **do**
    $\mathbf{K}^{(e)}(\alpha,\beta) \leftarrow$ `QuadInt(`$u_T(x^{(e)}(\xi), y^{(e)}(\xi)) \cdot \hat{f}_\beta(\xi) \cdot \hat{f}_\alpha(\xi) \cdot \sqrt{\mathbf{J}^{(e)T} \cdot \mathbf{J}^{(e)}}$`,1)`
  end for
end for

and

for α = 1 to 2 **do**
  if Neumann boundary condition **then**
    $\mathbf{f}^{(e)}(\alpha) \leftarrow$ `QuadInt(`$u_N(x^{(e)}(\xi), y^{(e)}(\xi)) \cdot \hat{f}_\alpha(\xi) \cdot \sqrt{\mathbf{J}^{(e)T} \cdot \mathbf{J}^{(e)}}$
  else
    if Cauchy boundary condition **then**
      $\mathbf{f}^{(e)}(\alpha) \leftarrow$ `QuadInt(`$u_T(x(\xi), y(\xi)) \cdot u_N(x(\xi), y(\xi)) \cdot \hat{f}_\alpha(\xi) \cdot \sqrt{\mathbf{J}^{(e)T} \cdot \mathbf{J}^{(e)}}$`,1)`
    end if
  end if
end for

with u_N being the boundary derivative and u_T being the transition coefficient of the boundary condition. Neumann and Cauchy boundary conditions are treated the same because the Neumann boundary condition is a special case of the Cauchy one. The term $\sqrt{\mathbf{J}^{(e)T} \cdot \mathbf{J}^{(e)}}$ is called the *pseudo determinant* of $\mathbf{J}^{(e)}$ because for finite edge elements $\mathbf{J}^{(e)}$ is a (2×1) matrix and therefor no inverse is defined.

The assembly of the FEM system of equations is finished at this point with all boundary conditions built in.

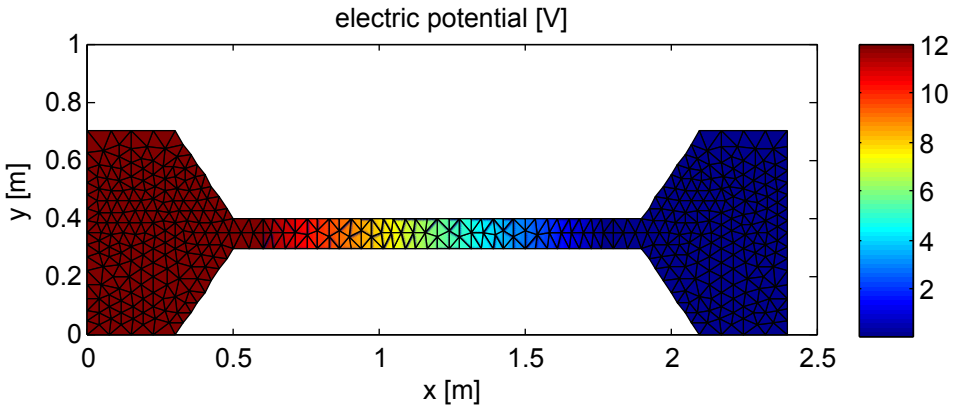

Figure 5.19.: Solution of the potential of the hardening device

5.5.3. Solving – FEM::solve()

As decided earlier, the solution of the FEM system of equations – derived by earlier steps – is not topic of this thesis. Without considering any efficiency weaknesses it is completely abandoned to MATLAB and will be topic of future work.

Hence, the function is a ”one-liner”:

Listing 5.29: Solve the FEM system of equations

```
function this = solve(this)
  u = this.StiffnessMatrix \ this.loadVector;
end
```

Note: Of course additional state control and validation is done in the actual code.

Figure 5.19 shows the solution of the hardening device example.

Example After the assembly of the FEM system of equations of the hardening device example, the MATLAB state of the FEM object might be of interest:

Listing 5.30: Final state of the FEM module

```
FEM =

FEMpack.FEM handle
package: FEMpack

properties:
                        Nodes: {1x478 cell}
                     numNodes: 478
                FiniteElements: {1x803 cell}
             numFiniteElements: 803
                 isDiscretized: 1
                   isAssembled: 1
                      isSolved: 1
                CoherenceTable: [803x3 double]
                DirichletNodes: {1x49 cell}
           BoundaryConditions: []
              StiffnessMatrix: [429x429 double]
                   loadVector: [429x1 double]
                         type: 'currentflow'
    numNodesPerFiniteElement: 3
```

Due to the chosen mesh refinement the geometry has been discretized with a triangulation of 478 nodes that are connected to 803 finite elements. The next three state variables `isDiscretized`, `isAssembled` and `isSolved` inform that the three basic steps of the finite element method – `discretize()`, `assemble()` and `solve()` – have been carried out successfully.
The `CoherenceTable` matrix contains the linkage between the 803 global nodes and the 3 local nodes of each finite element. The electrodes of the left and right side of the example are modeled by 49 Dirichlet nodes. No Neumann or Cauchy `BoundaryConditions` have been defined. The stiffness matrix has dimensions (429×429). This, of course, comes from the number of all nodes minus those Dirichlet nodes whose value is known $(478 - 49 = 429)$. The load vector also has 429 entries, naturally. The last value `numNodesPerFiniteElement` comes from using finite elements of type T3. This would be 6 using T6 elements as described earlier or could also be 4 when using quadrilateral finite elements.

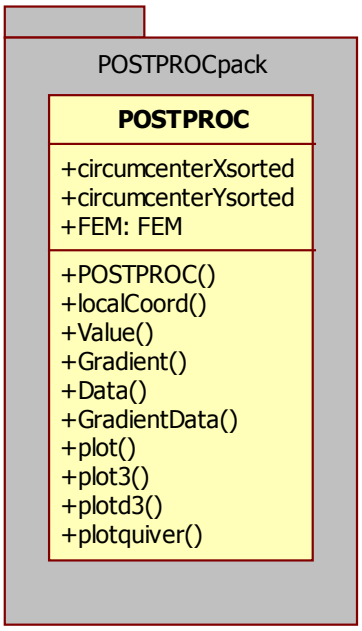

Figure 5.20.: POSTPROCpack UML diagram

5.6. Postprocessing – POSTPROCpack

The postprocessor implemented by the `POSTPROCpack` module finally helps the user interpreting and dressing the solution. It is currently arranged by one single class within the module but also uses class methods from the `FEMpack`. Especially these are the `FiniteElement` class and its inheritances that provide form functions and their gradients. Furthermore the solution potential values are saved to the `Node` objects.

Most important functions from the `POSTPROCESSOR` class are:

- `Value()`: This function calculates the potential value at any **continuous** coordinate of the solution using the finite element basis functions to interpolate.
- `Gradient()`: Evaluates the gradient of the solution at any continuous coordinate.
- `Data()`: Gives a sequence of solution data along a straight line which can be used for diagrams.
- `GradientData()`: Gives a sequence of the solution gradient along a straight line.
- `plotGeometry(),plotValue(),plotGradient(),plotMaterialGradient()`: These methods plot the geometry setup in a 2D graph, the solution in a 3D graph by using the z-axis to plot the value, as well as plotting the gradient.

Further methods can easily be implemented because any physical quantity at any point of the problem domain is available.

Before introducing specific functions of the postprocessor, a description is given of the transformation between the global problem space and the local reference frame. This is an important mechanism because the problem is defined in global space but most computation takes place in local space.

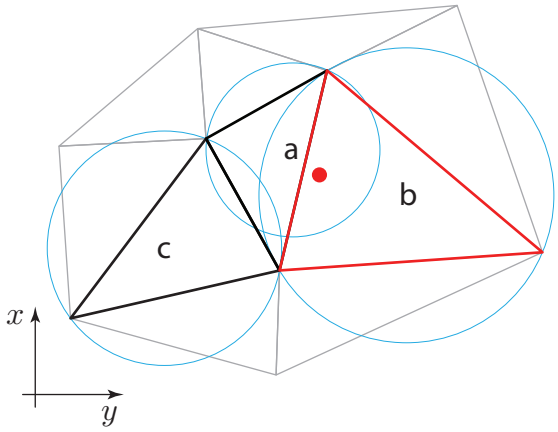

Figure 5.21.: Finite elements and their circumcircles

5.6.1. Local-to-global transformation

This transformation means providing a local coordinate (ξ, η) of the reference frame and obtaining a global coordinate (x, y) of the problem domain.

The local-to-global transformation is given by equation (4.49) on page 29. It is implemented as function `globalCoord()` within the `FiniteElement` superclass.

The Jacobian matrix $\mathbf{J}^{(r)}$ for the finite element is computed at its initialization. It is static for the currently used T3 type of finite elements and therefor needs to be computed once. Using finite elements of higher order would require a computation at every call of the function because the Jacobian matrix will be dependent on the local coordinates that are subject of being transformed to global coordinates. Figure 4.3 recalls the transformation.

5.6.2. Global-to-local transformation

For example, one wants to know the potential value u of the solution at global coordinates (x, y). This involves the global-to-local transformation because the value is interpolated using the basis functions which are implemented in the reference frame of the finite element that is subjacent to (x, y).

The inverse transformation cannot be done by just singling out local coordinates (ξ, η) of equation (4.49) in general. This has two reasons:

- When providing global coordinates (x, y) only, it is unknown which finite element r is subjacent.
- Furthermore the inverse transformation becomes non-linear for higher order finite elements. This is because the Jacobian matrix is dependent on the local coordinates: $\mathbf{J}^{(\mathbf{r})}(\xi, \eta)$. Consequently, also the inverse Jacobian matrix is dependent on the local coordinates: $\mathbf{J}^{-1}_{(r)}(\xi, \eta)$; but these are subject to being calculated.

The first reason rises the concept of circumcircles. The second problem is easily solved for T3 finite elements. For higher order elements a gradient descent method was implemented.

Figure 5.21 shows a sample triangulation with three finite elements highlighted and their circumcircles. The red point demonstrates global coordinates (x, y) that are subject to being brought to a

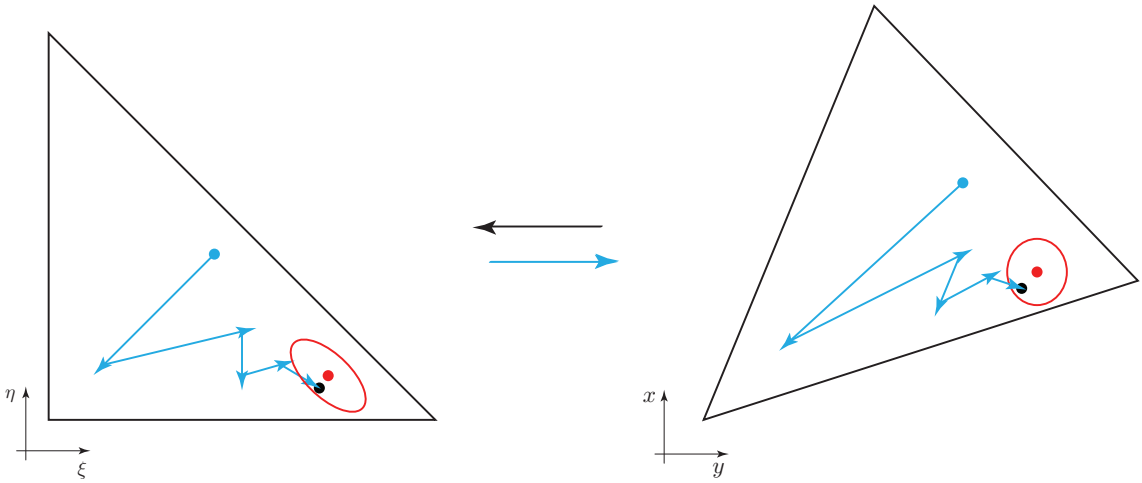

Figure 5.22.: Newton-Raphson method used to find local coordinate

local frame.

Before finding the correct inverse transformation, the correct local frame has to be found. This is demonstrated with the next pseudocode:

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**
(ξ, η) ← `localCoord(x, y)` {global-to-local transformation}

if (ξ, η) is inside of reference element **then**
{found (ξ, η)}
else
{continue search}
k ← k + 1
end if
end if
end while

The circumcenter and the radius of the circumcircle is computed for each finite element r at its initialization. When a global coordinate (x, y) needs to be transformed to the local frame of a finite element, a k-nearest-neighbor-search is performed and yields the "k-nearest" circumcenter to (x, y). In illustration 5.21 this is the circumcenter of finite element a. Doing a global-to-local transformation of (x, y) is not forbidden but delivers local coordinates (ξ, η) that are outside of the reference triangle of finite element a. The search for the correct local frame is continued with the next nearest neighbor which is finite element b. There, a valid local coordinate results.

This algorithm can be found in function `POSTPROCpack::POSTPROC::localCoord()`. It is the main function for all global-to-local computations of coordinates (x, y).

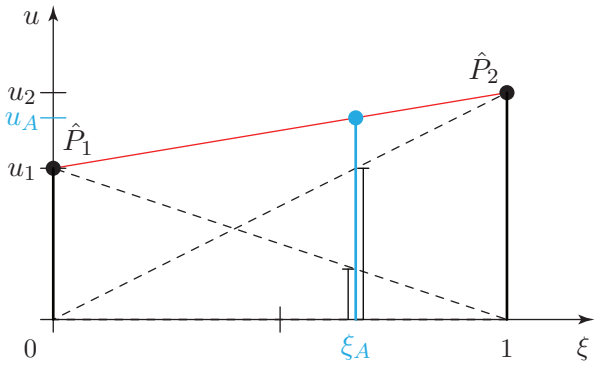

Figure 5.23.: Interpolating the solution

The actual global-to-local transformation is illustrated in figure 5.22. Coordinates (x, y) (red dot) in global space (right) are given. Because they cannot easily be transformed to the local frame (left), the corresponding local coordinates are approximated using a Newton-Raphson method. Starting at local coordinates $(0.5, 0.5)$, they are transformed to global space and tested against the given coordinates (x, y). A certain ϵ-neighborhood must be accepted as a stopping criterion. Depending on the test against the given coordinates a new iteration of local coordinates is computed until the stopping criterion matches. The result are the local coordinates (ξ, η) of the black dot. It is within the ϵ-neighborhood in global space.

The implementation currently uses $\epsilon = 1 \cdot 10^{-6} m$. This brings good results without remarkable efficiency losses.

The implementation `localCoord()` can be found within the generalized finite element class `FEM::FiniteElement`. In the special case of T3 finite elements the transformation can be done by simply inverting equation (4.49) to (5.4) because of the static Jacobi matrix. The implementation can be found within `FEMpack::FEM::FiniteElementT3::localCoord()`. It overloads the generalization.

$$\begin{pmatrix} \xi \\ \eta \end{pmatrix} = \mathbf{J}_{(r)}^{-1} \left[\begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} x_1^{(r)} \\ y_1^{(r)} \end{pmatrix} \right] \tag{5.4}$$

5.6.3. Determining a value of the solution – PP::Value()

Figure 5.23 shows an illustration of a one-dimensional finite element in local space and how the solution is interpolated. The value of the solution is asked for a global (one-dimensional) coordinate x. It is transformed to the local coordinate ξ_A of the respective local reference frame. The (dashed) basis functions of local nodes $\hat{P}_1$ with computed solution value u_1 and $\hat{P}_2$ with u_2 "span" the interpolated (red) solution between the nodes. Summing up the values of the two basis functions at coordinate ξ_A gives the solution u_A.

$$u_A(x) = \sum_{\alpha=1}^{2} u_\alpha \cdot \hat{f}_\alpha(\xi_A) = u_1 \cdot \hat{f}_1(\xi_A) + u_2 \cdot \hat{f}_2(\xi_A) \tag{5.5}$$

The same interpolation can be used for two dimensions with T3 finite elements:

$$u(x,y) = \sum_{\alpha=1}^{3} u_\alpha \cdot \hat{f}_\alpha(\xi(x,y), \eta(x,y)) \tag{5.6}$$

Figure 5.28 on page 84 shows a T3 finite element with its basis functions and the "spanned" solution in global space.

The implementation of T3 form functions was shown in listing 5.23 on page 68. These are:

$$\hat{f}_1(\xi,\eta) = 1 - \xi - \eta \quad \hat{f}_2(\xi,\eta) = \xi \quad \hat{f}_3(\xi,\eta) = \eta \tag{5.7}$$

The implementation of the `Value()` function is as simple as:

Listing 5.31: Computing the solution value at continuous coordinates

```
function [z, rho, sigma, valid] = Value(this, xi)
  z = 0;
  for alpha = 1:numLocalNodes
    z = z + localFormFunc(this, alpha, xi)
          * this.Nodes{alpha}.value;
  end

  % ... %
end
```

Note: Input argument `xi` is a two-dimensional vector. The output value `z` – the electric potential u for electrostatic and current flow, the magnetic scalar potential Θ for magnetostatic and the temperature T for thermal problems – is saved within the node objects and is the same within the local and global frame. It does not need to be transformed, therefor.

The following table lists the interpretation of the solution depending on the type of problem:

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

Table 5.4.: Interpreting the solution

For the sake of completeness, the basis functions of T6 finite elements are mentioned. These are implemented within `FiniteElementT6::localFormFunc()`:

$$\begin{aligned}
\hat{f}_1(\xi,\eta) &= 2\xi^2 + 2\eta^2 + 4\xi\eta - 3\xi - 3\eta + 1 & \hat{f}_2(\xi,\eta) &= 2\xi^2 - \xi \\
\hat{f}_3(\xi,\eta) &= 2\eta^2 - \eta & \hat{f}_4(\xi,\eta) &= -4\xi^2 - 4\xi\eta + 4\xi \\
\hat{f}_5(\xi,\eta) &= 4\xi\eta & \hat{f}_6(\xi,\eta) &= -4\eta^2 - 4\xi\eta + 4\eta
\end{aligned} \tag{5.8}$$

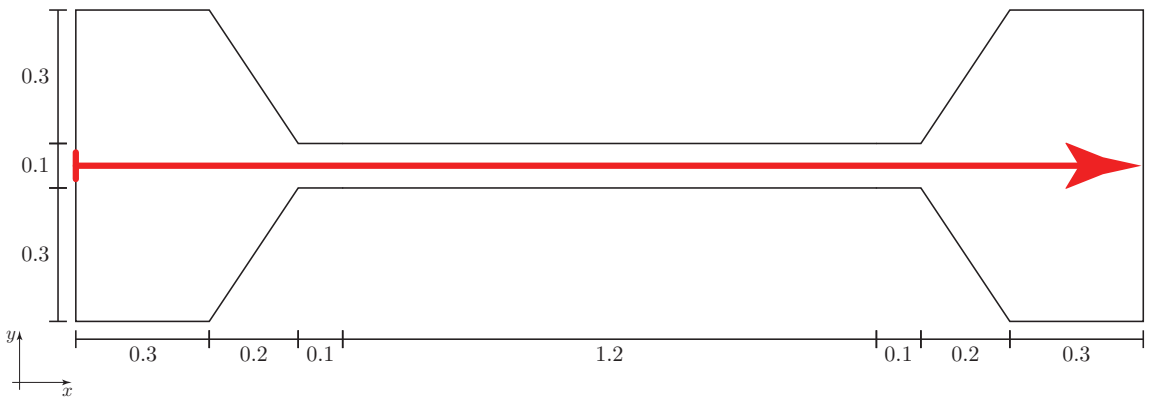

Figure 5.24.: Evaluating the solution along a straight line – drawing a diagram

5.6.4. 2D diagram of the solution – PP::Data()

The `Data()` function returns a series of solution values computed by `Value()` along a straight line. Its header is:

Listing 5.32: Header of the Data() function

```
function d = Data(this, from, to, step)
  % from: from coordinate
  % to:   to coordinate
  % step: step size

  % ... %
end
```

The `from` argument defines the starting point, the `to` argument the end point of a straight line. The third argument `step` is the distance between two evaluation points. An example using the solution of the hardening device:

Listing 5.33: Example of gathering the solution along a straight line

```
d = Data(POSTPROC, [0; .35], [2.4; .35], .01);
```

This gives all solution values from $(0, 0.35)$ to $(2.4, 0.35)$ with a step distance of 0.01. Figure 5.24 shows the evaluation along a straight line with the red arrow. The geometry was defined in figure 5.7 on page 47. A diagram of the above call shows figure 5.25.

5.6.5. Determining the gradient of the solution – PP::Gradient()

This function computes the gradient $\text{grad}\, u(x, y)$ at global coordinates (x, y) of the solution $u(x, y)$.

$$\text{grad}\, u(x, y) = \begin{pmatrix} \frac{\partial u(x,y)}{\partial x} \\ \frac{\partial u(x,y)}{\partial y} \end{pmatrix} \tag{5.9}$$

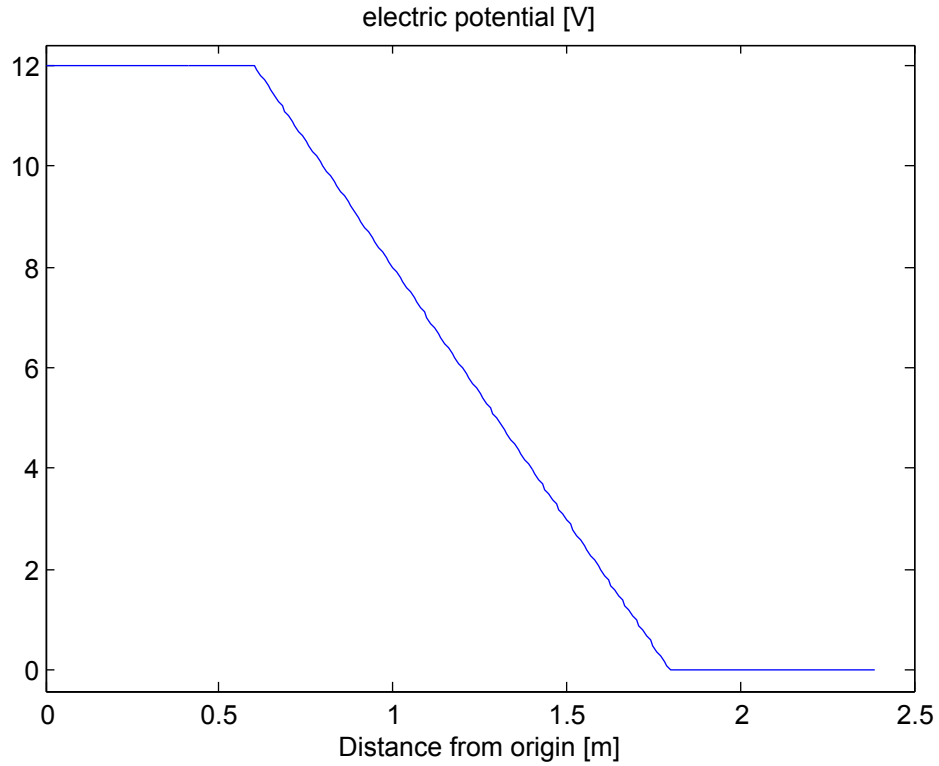

Figure 5.25.: Electric potential u along a straight line

It is also computed within the local reference frame at the corresponding local coordinates (ξ, η) using the basis functions. This is the "local" gradient, however, and must be transformed to the global space.

The relation between the gradient at local space and global space is

$$\text{grad}\, u(x,y) = \mathbf{J}_{(r)}^{-T}\, \text{grad}\, u(\xi^{(r)}(x,y), \eta^{(r)}(x,y)) \tag{5.10}$$

The gradients of the basis functions are implemented as class methods of the `FiniteElementT3` class for T3 finite elements. Listing 5.24 on page 68 demonstrated the implementation.

The basis functions of local nodes $\hat{P}_1$, $\hat{P}_2$ and $\hat{P}_3$ are:

$$\hat{f}_1(\xi,\eta) = \begin{pmatrix} -1 \\ -1 \end{pmatrix}, \quad \hat{f}_2(\xi,\eta) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \hat{f}_3(\xi,\eta) = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{5.11}$$

The transformation to global space is

$$\sum_{\alpha=1}^{3} u_\alpha \cdot \mathbf{J}_{(r)}^{-T} \cdot \hat{f}_\alpha(\xi^{(r)}(x,y), \eta^{(r)}(x,y)) \tag{5.12}$$

and is implemented in function `Gradient()` of the superclass `FiniteElement`:

```
function g = Gradient(this, xi)
  g = zeros(2, 1);
  for alpha = 1:3
    g = g + InverseJacobian(this, xi)'
            * localGradientFormFunc(this, alpha, xi)
            * this.Nodes{alpha}.value;
  end
end
```

Basis function gradients for finite elements of a different type must be implemented within their designated class. For T6 finite elements, the basis function gradients are implemented in `FiniteElementT6::lo` as:

$$\begin{aligned}
\operatorname{grad}\hat{f}_1 &= \begin{pmatrix} 4\xi + 4\eta - 3 \\ 4\xi + 4\eta - 3 \end{pmatrix} & \operatorname{grad}\hat{f}_1 &= \begin{pmatrix} 4\xi - 1 \\ 0 \end{pmatrix} \\
\operatorname{grad}\hat{f}_1 &= \begin{pmatrix} 0 \\ 4\eta - 1 \end{pmatrix} & \operatorname{grad}\hat{f}_1 &= \begin{pmatrix} -8\xi - 4\eta + 4\eta \\ -4\xi \end{pmatrix} \\
\operatorname{grad}\hat{f}_1 &= \begin{pmatrix} 4\eta \\ 4\xi \end{pmatrix} & \operatorname{grad}\hat{f}_1 &= \begin{pmatrix} -4\eta \\ -4\xi - 8\eta + 4 \end{pmatrix}
\end{aligned} \tag{5.13}$$

The interpretation of the solution gradient is listed in this table:

problem type	derivation	interpretation of the gradient
electrostatic	$\mathbf{E} = -\operatorname{grad} u$	electric field $\mathbf{E}$ $[V/m]$
current flow	$\mathbf{E} = -\operatorname{grad} u$	electric field $\mathbf{E}$ $[V/m]$
magnetostatic	$\mathbf{H} = -\operatorname{grad} \Theta$	magnetizing field intensity $\mathbf{H}$ $[A/m]$
thermal		no interpretation

Table 5.5.: Interpreting the gradient of the solution

5.6.6. Gradient considering material properties – PP::MaterialGradient()

This function is an extension to the last one. Its implementation is the same but considers defined material properties. The interpretation of the gradient depends on the type of problem:

problem type	derivation	interpretation of the gradient
electrostatic	$\mathbf{D} = \epsilon\mathbf{E}$	electric displacement $\mathbf{D}$ $[C/m^2]$
current flow	$\mathbf{J} = \sigma\mathbf{E}$	current density $\mathbf{J}$ $[A/m^2]$
magnetostatic	$\mathbf{B} = \mu\mathbf{H}$	magnetic induction $\mathbf{B}$ $[T]$
thermal	$\ddot{\mathbf{q}} = -k \operatorname{grad} T$	heat flux $\ddot{\mathbf{q}}$ $[W/m^2]$

Table 5.6.: Interpreting the gradient of the solution considering material properties

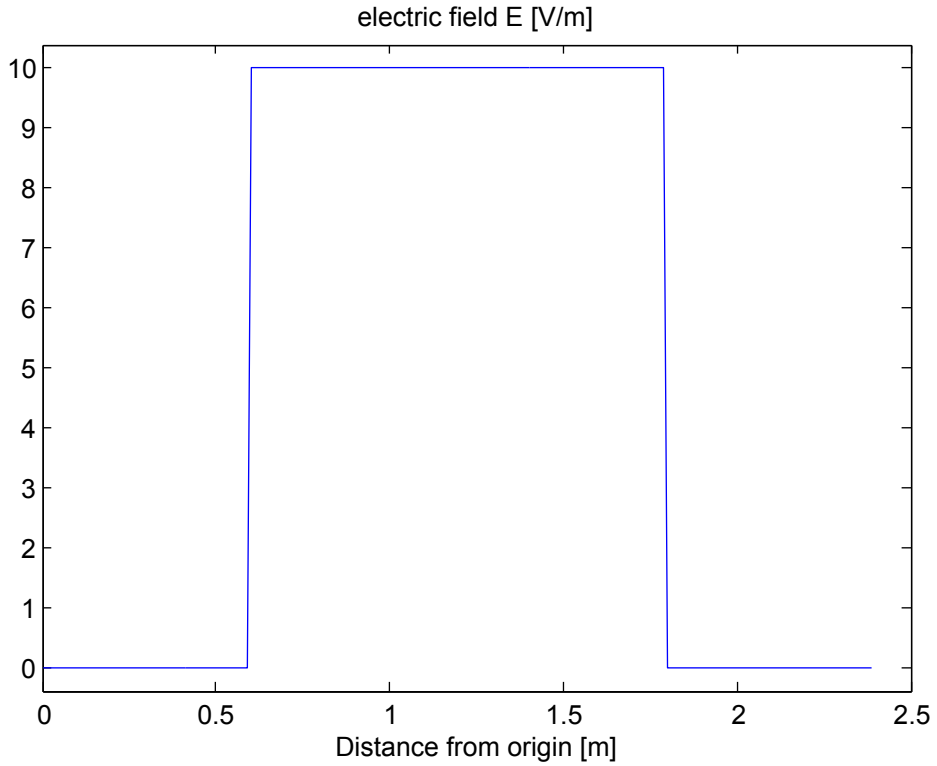

Figure 5.26.: Electric field $\mathbf{E}_x$ along a straight line

5.6.7. 2D diagram of the gradient – PP::GradientData()

This is an analogous function to `Data()` and returns a two-dimensional series of gradients as shown in equation (5.10).

Figure 5.26 shows the x-component of the electric field $\mathbf{E}$ of the hardening device along the same straight line as to demonstrate the electric potential in 5.6.4.

$$\mathbf{E} = -\operatorname{grad} u \tag{5.14}$$

Note: Because of the discretization, the gradient of the computed FEM solution is not continuous. The ”hackly” look of the diagram is a consequence of this fact.

5.6.8. 2D diagram of the gradient considering material – PP::MaterialGradientData()

Also this class is implemented analogously to the last one. It outputs a gradient series considering defined material properties. Figure 5.27 shows the x-component of the current flow $\mathbf{J}$ of the hardening device along the straight line as introduced by figure 5.24.

$$\mathbf{J} = \sigma\mathbf{E} = -\sigma \operatorname{grad} u \tag{5.15}$$

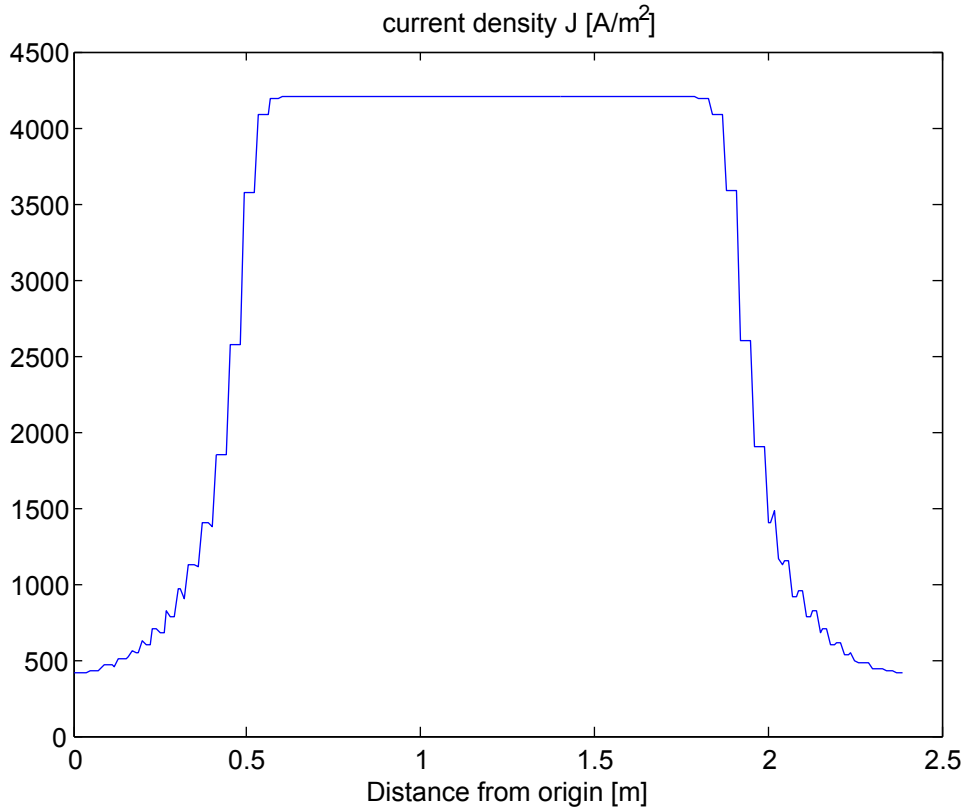

Figure 5.27.: Current flow $\mathbf{J}_x$ along a straight line

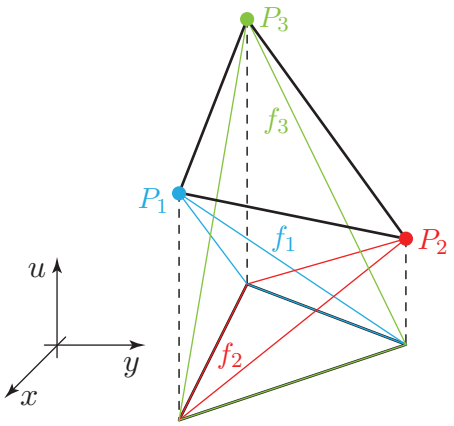

Figure 5.28.: T3 finite element in global space with basis functions

5.6.9. 2D geometry plot – PP::plotGeometry()

The postprocessor is also capable of plotting two- and three-dimensional graphs. The first of them is implemented with the `plotGeometry()` function. Its output of the geometry of the hardening device example was already shown in figure 5.14 on page 58. The output consists of all nodes of the triangulation and all edges of the finite elements. Furthermore, defined boundary conditions and sources are highlighted. Differing material properties are currently not considered. Options considering colors, line widths, text marking and so on are currently hard-coded within the postprocessor and can be adapted there.

5.6.10. 3D solution plot – PP::plotValue()

The `plotValue()` function of the postprocessor produces a 3D plot of the solution. The x- and y-coordinates are in accordance to the geometry of the given problem. The value of the solution is plotted along the z-coordinate using a color palette between the minimum and maximum value.

The solution shape over a finite element of T3 type is a triangle defined by the values of the finite element nodes. This is illustrated in figure 5.28. Consequently, the `plotValue()` function plots the solution by using one triangle `patch`. The graph, as seen by the user, equates exactly the solution of the discretized problem model.

Other finite element types are plotted by partitioning the reference element and transforming each partition into global space. Figure 5.29 shows this technique for a T6 finite element. The fineness of the partitioning is hard-coded within `FiniteElement::plot3()` and can be adapted there.

The respective function is implemented as `plot3()` within `FiniteElementT3` for T3 type finite elements and as `plot3()` within superclass `FiniteElement` for all other types.

Figure 5.30 shows the solution – the electric potential u – of the hardening device in a 3D plot.

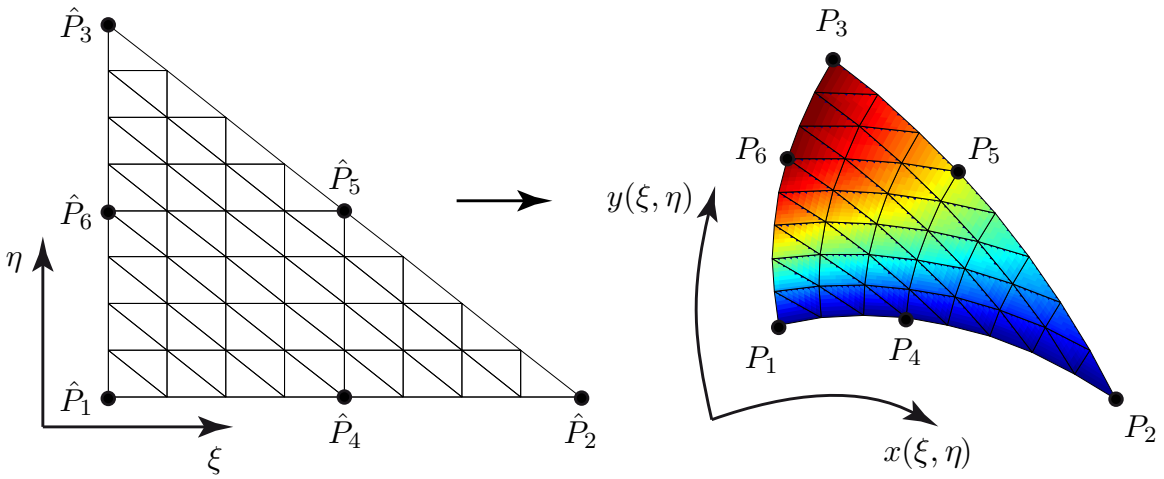

Figure 5.29.: Approximated shape of a T6 finite element

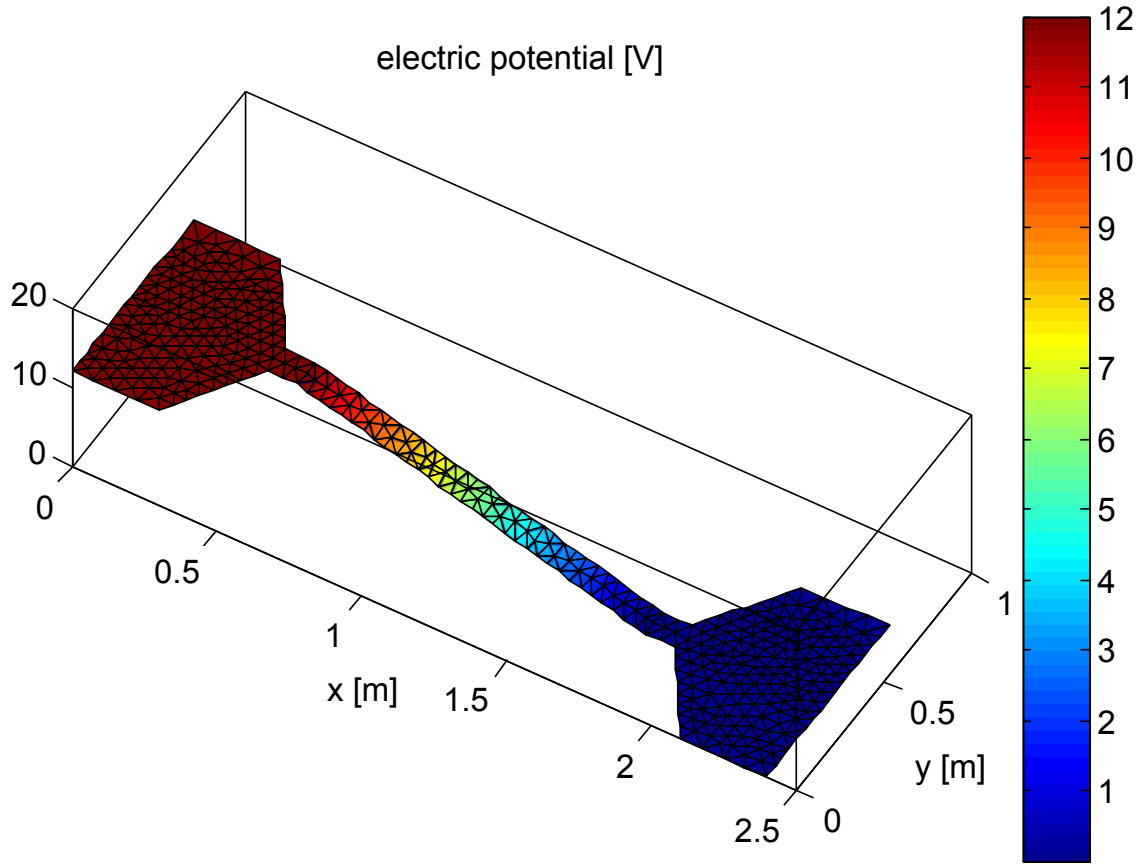

Figure 5.30.: Solution of the hardening device example

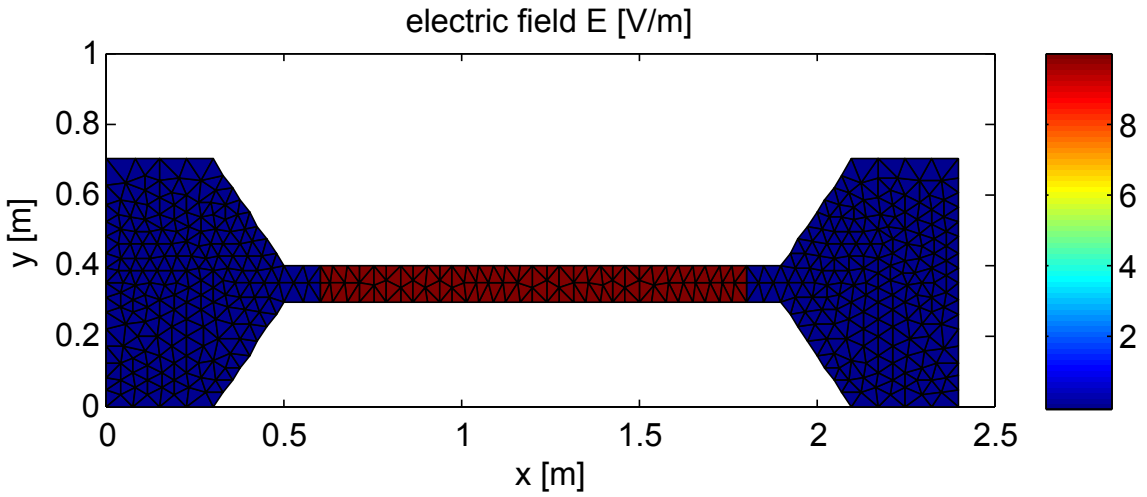

Figure 5.31.: Electric field $\mathbf{E}$ of the hardening device

5.6.11. 3D gradient plot – PP::plotGradient()

The `plotGradient()` function gives a 3D illustration of the gradient of the solution as introduced in 5.6.5 and 5.6.7. The interpretation can be found in table 5.5 on page 81.

Computing the gradient was shown in 5.6.5, the plotting mechanism is analogous to 5.6.10. Figure 5.31 shows the XY-view of the gradient of the hardening device solution which equals the electrical field $\mathbf{E}$.

5.6.12. 3D gradient plot considering material – PP::plotMaterialGradient()

This function is – concerning the implementation – almost the same as the last one but considers the material properties. The interpretation of the output depends on the problem type and is listed in table 5.6 on page 81.

Figure 5.32 shows the material considering gradient which results to the current density $\mathbf{J}$ of the hardening device. The next illustration 5.33 shows the same graph with a different view. There, the discontinuity of the gradient of the solution can be observed that leads to the ”hackly” diagrams 5.26 and 5.27.

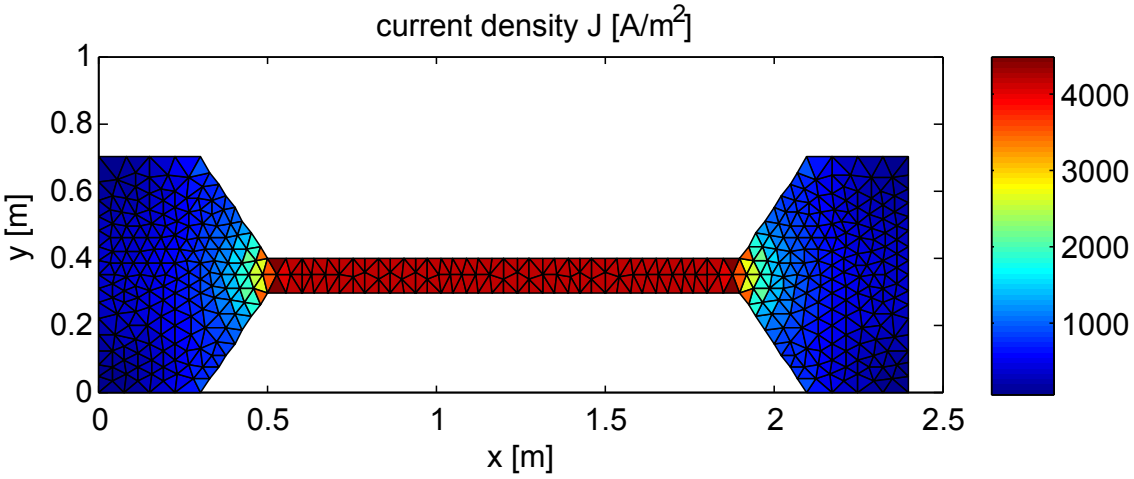

Figure 5.32.: Current density **J** within the hardening device

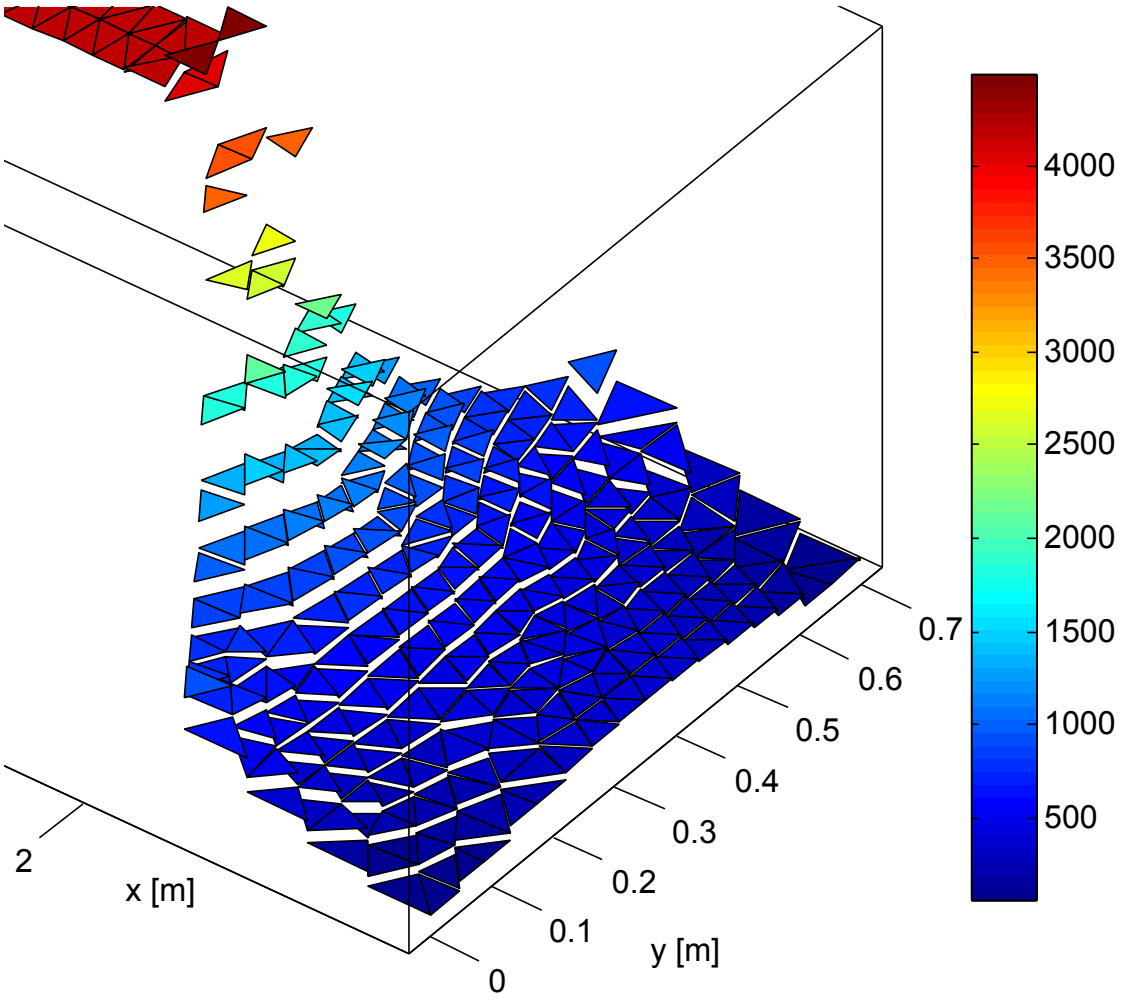

Figure 5.33.: A different view of the current density **J**

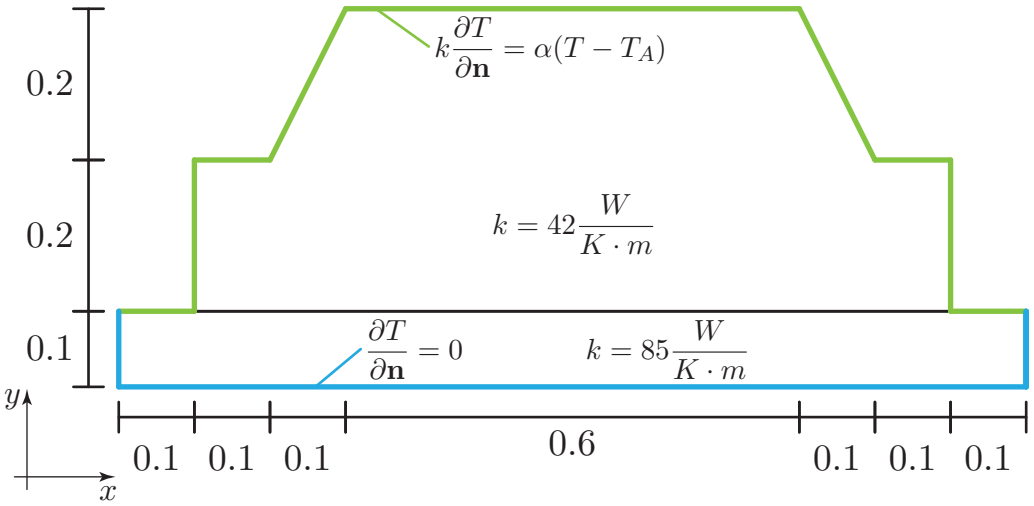

Figure 5.34.: The thermal model

5.7. Multiphysics

The first subtask for the heating device example as defined on page 47 has been shown in detail, so far. The intermediate result is the distribution of the current flow **J**.
In order to gain the final result – this is the temperature distribution or the scalar temperature field T of the steel workpiece – a second model is to be computed. This approach of having two *coupled models* in two different physical domains is called **multiphysics**.
Due to the current flow within the heating plate, power losses produce heat emissions. In the second thermal model, these power losses will act as energy sources for heat.

Figure 5.34 defines the thermal subtask. Only the heating plate and the steel workpiece are modeled. Thermal effects within the electrodes are neglected. The heating plate alloy is assumed with a thermal conductivity of $k = 85WK^{-1}m^{-1}$, the steel workpiece with $k = 42WK^{-1}m^{-1}$. The bottom of the heating plate is thermally isolated and also the connections to the electrodes are assumed to have no temperature flow. This is mathematically expressed by Neumann boundary conditions illustrated in blue color. Through the surface of the steel workpiece and the loose area of the heating plate, a thermal exchange with the ambient environment occurs. This occurrence can be modeled using a Cauchy boundary condition as shown with green color. The parameter α is the *transport coefficient* and describes the thermal flow between the system and the ambient environment with ambient temperature T_A. These are $\alpha = 5.6WK^{-1}m^{-2}$ and $T_A = 300K$.

FEMtastic supports coupled problems with the `import()` function implemented within the `FEM` class of the `FEMpack` module. It takes the results of a previously initialized `POSTPROCESSOR` object and includes them right before the *assembly* step.

Note: The only precondition is that identical parts of both models must be at the same place. Herein, this means that the heating plate has the same coordinates in both the current flow and the thermal model. Different setups of macroelements or a different meshing is of no concern.

Listing 5.34: Multiphysics

```
1  % THERMAL MODEL
2  PREthm = PREPROCpack.PREPROC();
3  FEMthm = FEMpack.FEM();
4
5  % preprocessing thermal model
6  PREthm = ProcessXMLFile(PREthm, xml_file_thermal);
7
8  % setting up FEM kernel
9  FEMthm = discretize(FEMthm, PREthm);
10   % importing the previous model
11 FEMthm = import(FEMthm, POSTPROC, 'current-thermal');
12 FEMthm = assemble(FEMthm);
13 FEMthm = solve(FEMthm);
14
15 % postprocessing the thermal model
16 POSTthm = POSTPROCpack.POSTPROC();
17 POSTthm = init(POSTthm, FEMthm);
```

Listing 5.34 shows how the results from the current flow model are brought into the thermal model. The `POSTPROC` object is the postprocessor from the first model and holds its results – namely the current flow $\mathbf{J}$. In line 9 – right before the FEM system of equations of the thermal computation is assembled – the previous results are being incorporated. The additional parameter *'current-thermal'* defines the conversion of the previous results. It is:

$$p = \frac{1}{\sigma}\mathbf{J}^T\mathbf{J} \tag{5.16}$$

This equation states the context between the current flow $\mathbf{J}$ and the power loss p in $\frac{W}{m^3}$. Latter ones are the sources of the thermal problem.

The respective files are:

- *hardening_device.m*
- *hardening_device-thermal.xml*

Listing 5.35: Importing data from a previous model

```
1  function this = import(this, POSTPROC, op)
2    % where finite elements are stored
3    FE = this.FiniteElements;
4
5    if strcmp(op, 'current-thermal');
6      % current - thermal coupling
7      for i = 1:length(FE)
8        [u, rho, sigma, valid] =
9            Value(POSTPROC, globalCoord(FE{i}, [0.33; 0.33]));
10       if valid
11         J = MaterialGradient(POSTPROC, globalCoord(FE{i},
12                                                    [0.33; 0.33]));
13         FE{i}.loadFunc = 1/sigma * J' * J;
14       end
15     end
16     return;
17   end
18
19   % ... %
20 end
```

This listing shows the `import()` function for T3 finite elements. The power loss is computed for all finite elements of the current flow model (line 6) using the results from the previous model (line 7). The `Value()` function (see 5.6.3 on page 77) returns a *valid* flag that states whether the geometry exists in both coupled models.

The remaining part of the simulation of the thermal problem is the same as with the current flow problem. Hence, presenting the results is the missing part. Figure 5.35 shows the geometry and the triangulation of the thermal model, 5.36 shows the temperature distribution within the heating plate and the steel workpiece.

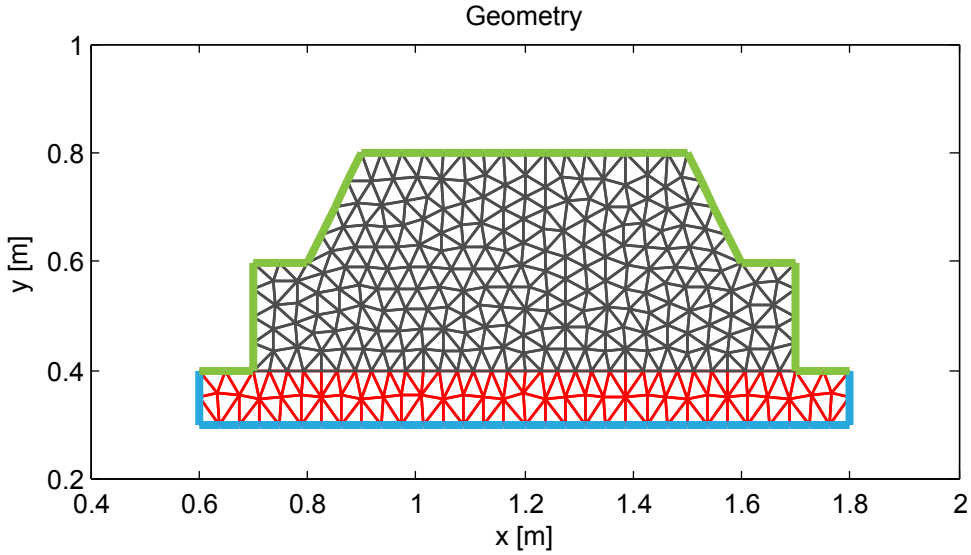

Figure 5.35.: Geometry and mesh of the thermal problem

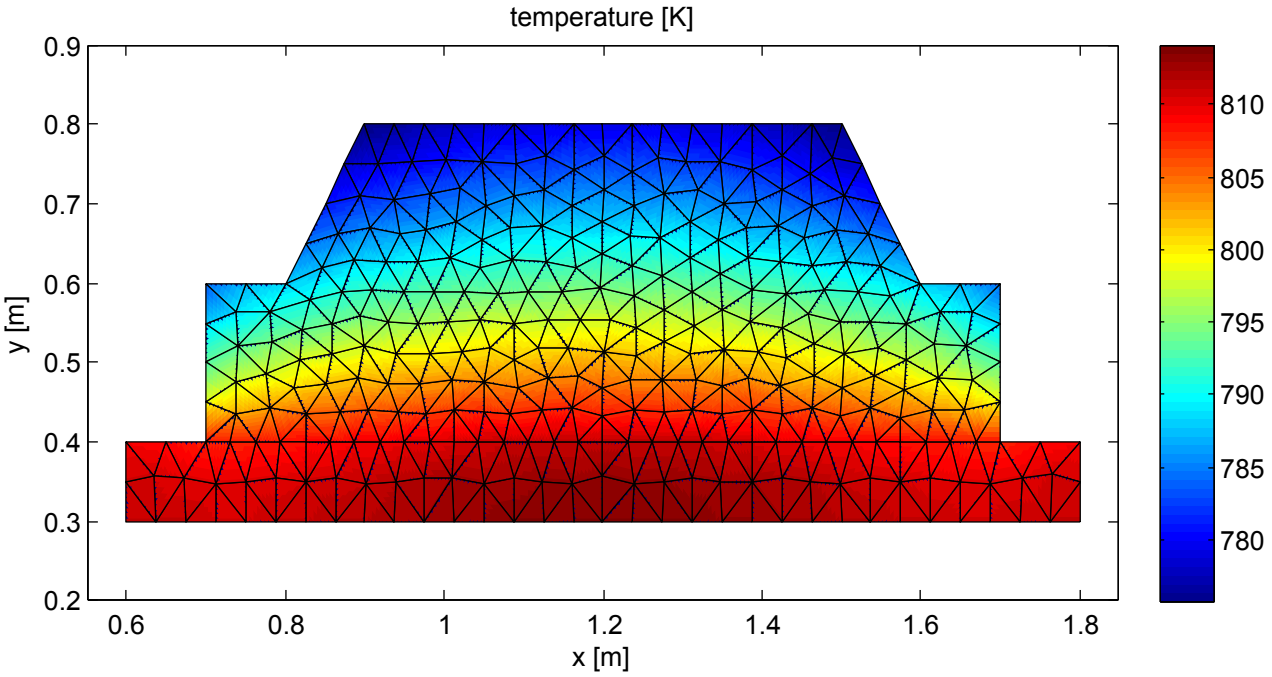

Figure 5.36.: The thermal distribution of the heating plate and the steel workpiece

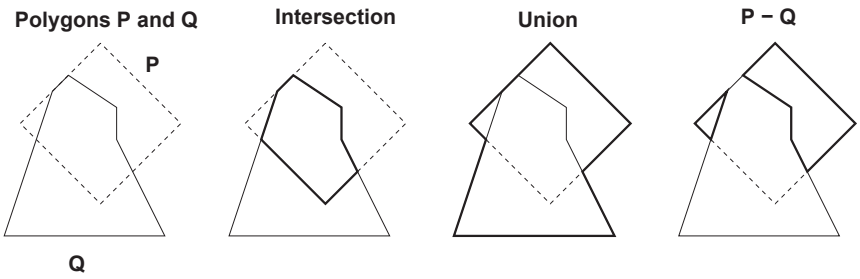

Figure 5.37.: Boolean operations on polygons – source: [17]

5.8. Boolean operations on polygons – Martinez algorithm

In order to be able to do reasonable finite element analysis it is, of course, necessary to endow the user with mechansisms to formulate arbitrary assignment definitions. FEMtastic is aimed to apply the finite element method to any two-dimensional geometry setup. This has been realized by a MATLAB implementation of the algorithm presented in *A new algorithm for computing Boolean operations on polygons* [17] by Francisco Martínez, Antonio Jesús Rueda and Francisco Ramón Feito. The algorithm, herein, is referred to as the **Martinez algorithm** and was implemented independently from FEMtastic, so it can also be used in other applications.

The implementation can be found in the *+MARTINEZ* MATLAB package folder and is well documented.

The algorithm applies a boolean operation to two polygons – the *subject* polygon P and the *clipping* polygon Q. These operations are

- Union,
- Difference and
- Intersection.

The outstanding advantages of this algorithm are:

- Fast computation in time $O((n+k)\log(n))$ with n being the number of total edges of both polygons and k the total amount of intersections between the edges. It might therefor also be used interacively in a future graphical user interface.
- The Martinez algorithm is not limited to a certain type of polygons. Concave and even self-intersecting polygons are supported.
- Polygons with ”holes” are possible.
- The output is not degenerated. Degeneration means that the output is only an approximation of the real result due to limitations of the algorithm. Figure 5.38 shows the correct result of a boolean operation on the right side. The result on the left is degenerated because the two areas of the real result are connected through small bands.
- Regions composed of polygon sets are possible. The result of the difference operation in figure 5.37 shows two regions. These are treated as one polygon data structure, however. This kind of polygons might also be used as input subject or clipping polygon.

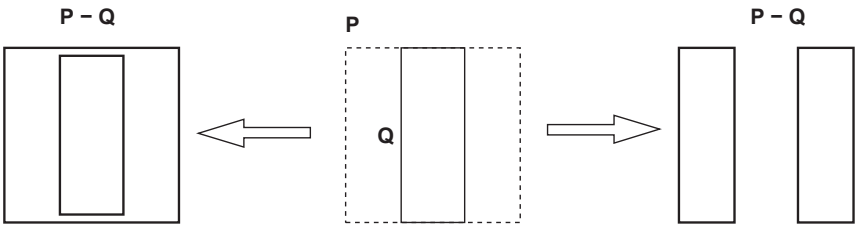

Figure 5.38.: Degeneracy – source: [17]

The Martinez algorithm, thus, seems to provide optimal functionality in order to build up complex geometry setups by defining simple polygon primitives and applying boolean operations to them. Setting up an iron core with air gap was shown as an example in figure 5.3 on page 40. Furthermore, the preconfiguration of FEMtastic with different geometry primitives will be done as a future work. These might be special structures like circles, ovals, rounded rectangles and others as also provided by EleFAnT.

5.8.1. Algorithm

The algorithm is based on the sweep line paradigm and does three tasks [17, sec. 2]:

1. Subdivide the edges of the polygons at their intersection points.
2. Select those subdivided edges that either lie inside the other polygon or that do not, depending on the boolean operation.
3. Join the edges selected in step 2 to form the result polygon.

The algorithm was also implemented using object oriented programming. The left and right endpoints of each segment are *left events* and *right events*, respectively. These events get organized into an *event queue* which is sorted by the x-coordinates of the events.
The vertical *sweep line* moves from left to right, starting by the left-most event. Whenever a left event is reached, the corresponding segment is added to a *sweep queue*. The sweep queue holds those segments – represented by their left event objects – that are currently intersected by the vertical sweep line, sorted in y-direction at every time. This means, a left event is inserted into the sweep queue so that the previous event in the queue belongs to the segment *below* it and the next event in the queue to the segment *above* (assuming there are segments below or above).
After a new segment was inserted into the sweep queue, tests are performed, whether the segment intersects with the segment below or above. It can be proved that all segment intersections are found with this technique. Details are given in [17].

If an intersection between two segments is detected and these two segments do not only intersect at their endpoints, they are properly divided at the point of intersection and the new segments are added to the event queue – still sorted by their x-coordinates, of course.

Whenever the sweep line reaches a right event, the corresponding segment is removed from the sweep queue. Depending on the boolean operation (union, difference or intersection) the segment is added (or not) to the resulting polygon. This test is described in the next section 5.8.2.

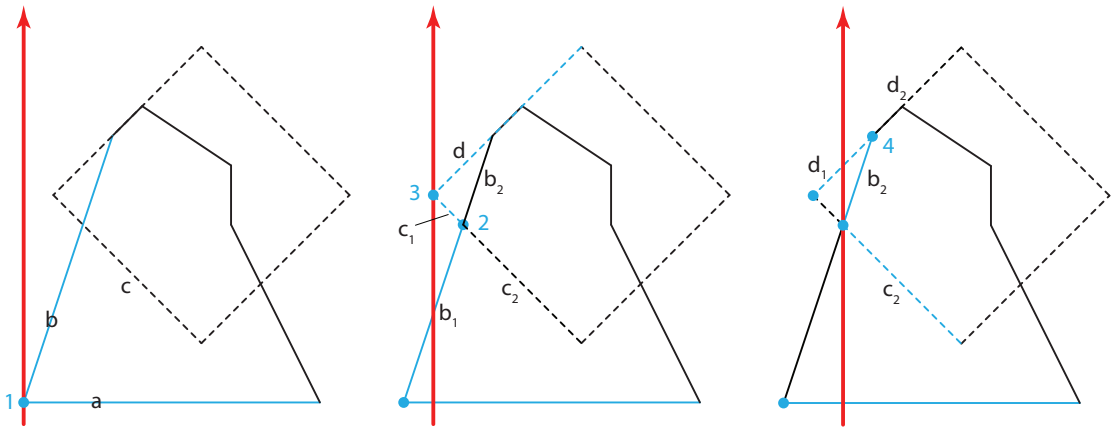

Figure 5.39.: Finding intersection points and subdividing edges

5.8.2. Example

Figure 5.39 uses the subject and clipping polygons from figure 5.37 to demonstrate the advance of the sweep line. The subject polygon is dashed, the clipping polygon is the solid one. The segments in blue color are the ones that are currently in the sweep queue.

- The sweep line starts at the left-most events. These are the left events of segments a and b. The two segments are added to the sweep queue one after another. Testing them for intersection identifies their common endpoint 1. Because it is an endpoint "only", no further processing needs to be done.
- The sweep line advances to the next events. These are two left events located at the left corner of the subject polygon. The lower segment c is added first and tested for intersection with its below segment b. The intersection point 2 is detected. Segment b is divided to b_1 and b_2, segment c to c_1 and c_2. Note that b_2 and c_2 are self-contained segments now that have not been added to the sweep queue so far. Four additional events are added to the event queue. These are the right endpoint events of segments b_1 and c_1 and the left endpoint events of segments b_2 and c_2.
- The next event to be processed is the left endpoint event of segment d. The common endpoint 3 with c_1 is found and no further processing is necessary.
- The right endpoints of segments b_1 and c_1 are the events to be processed next. The segments are deleted from the sweep queue. Whether they are added to the resulting polygon is checked at this point. How this decision is made is described in section 5.8.3 on the next page.
- At the same x-coordinate the two left events from segments b_2 and c_2 are located. Segment c_2 is added first, because it is the lower segment. No intersection with another segment can be found.
- Inserting b_2 brings up the intersection point 4 with d which is divided into d_1 and d_2. The right endpoint event of d_1 and the left endpoint event of d_2 are added to the event queue.
- The algorithm continues in this manner and terminates after processing the last two right endpoint events which are located at the right corner of the subject polygon.

5.8.3. Adding edges to the result polygon

In order to determine whether a segment belongs to the result polygon, three extra flags are set for each segment. These are [17]:

- `polygon`: states whether the edge belongs to the subject or clipping polygon,
- `inside`: indicates if the edge is inside the other polygon, and
- `inOut`: indicates if the edge determines an *inside-outside transition* into the polygon, to which the edge belongs, for a vertical semi-line that goes up and intersects the edge.

Determining the flags is done whenever a left endpoint event is reached and the corresponding segment is added to the sweep queue. This routine with *le* being a left event (segment) and *ple* the *previous (lower)* left event of the sweep queue shows the determination of the flags:

if ple == 0 **then** {Case 1: no segment below}
  le.inside ← false
  le.inOut ← false
else
  if le.polygon == ple.polygon **then** {Case 2: same polygon}
    le.inside ← ple.inside
    le.inOut ← ! ple.inOut
  else {Case 3: different polygon}
    le.inside ← ! ple.inOut
    le.inOut ← ple.inside
  end if
end if

Figure 5.40 shows the determination of the `inside` and `inOut` flags of the segments that are currently within the sweep queue. The illustrated status occurs at the time when d_1 (respectively its right endpoint event) is removed from the sweep queue.

Segment a is the lowest one and is initialized with `inside = inOut = false`. The next segment above a is c_2 and a belongs to the clipping polygon, c_2 to the other subject polygon. Case three matches. Also testing b_2 and d_1 results in case three because the segments belong to the other polygon, each.

Whether a segment belongs to the result polygon, of course, depends on the boolean operation. This is tested whenever a segment is removed from the sweep queue because its right endpoint event was reached. The decision is apparent – *re* means the right endpoint event and has the same flags stored as the left endpoint event of the same segment:

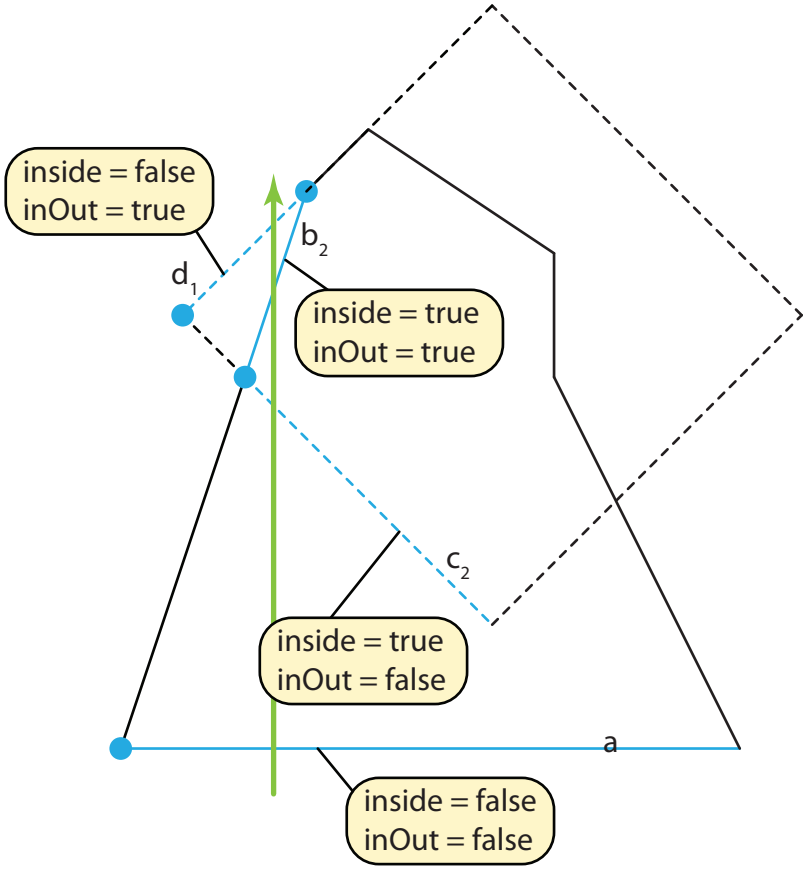

Figure 5.40.: Determining `inside` and `inOut` flags of the edges

if re.inside == true **then**
  Segment belongs to the *INTERSECTION* result polygon.
end if
if re.inside == false **then**
  Segment belongs to the *UNION* result polygon.
end if
if (re.polygon == 'subject' **and** re.inside == false)
  or (re.polygon == 'clipping' **and** re.inside == true) **then**
  Segment belongs to the *DIFFERENCE* result polygon.
end if

5.8.4. Special cases

The algorithm also treats special cases in an elegant way. These are vertical segments, because their endpoints both have the same x-coordinate, and overlapping edges. The treatment is described in [17] and omitted herein.

Chapter 6

Conclusions and outlook

This chapter is aimed to overlook FEMtastic once more and highlight its benefits as well as identifying some weaknesses and suggesting ideas for future work.

6.1. Proof of concept

The correctness of the results gathered with FEMtastic and presented herein, is demonstrated by a comparison of the same problems modeled and computed with EleFAnTs.

6.1.1. Current flow problem

Figure 6.1 shows the setup and mesh of the current flow subtask using EleFAnTs. All properties and boundary conditions are, of course, the same. These are the left electrode with an electric potential of $u = 12V$, the right one with $u = 0V$, the material conductivity of the electrodes with $\sigma = 60{\cdot}10^6 S/m$ and the conductivity of the heating plate with $\sigma = 420 S/m$.

Figures 6.2 to 6.7 demonstrate that the results of EleFAnTs and FEMtastic are the same and therefor considered as correct.

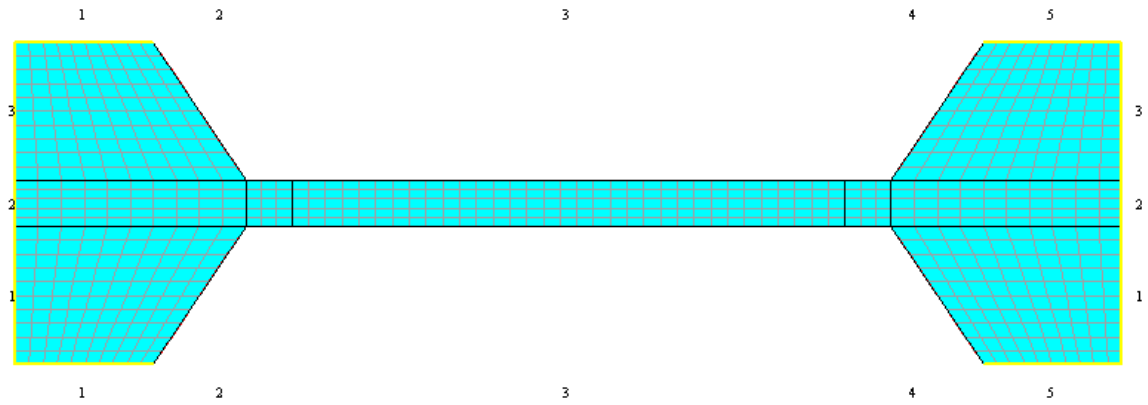

Figure 6.1.: EleFAnTs: current flow model

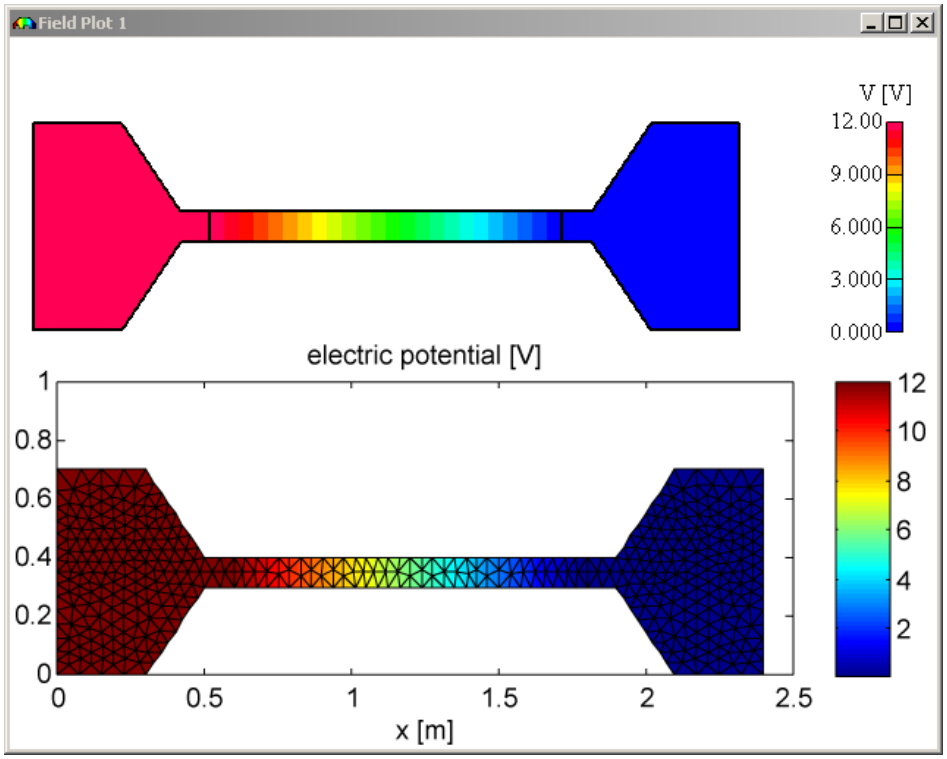

Figure 6.2.: Comparison of electric potential u results (top: EleFAnTs, bottom: FEMtastic)

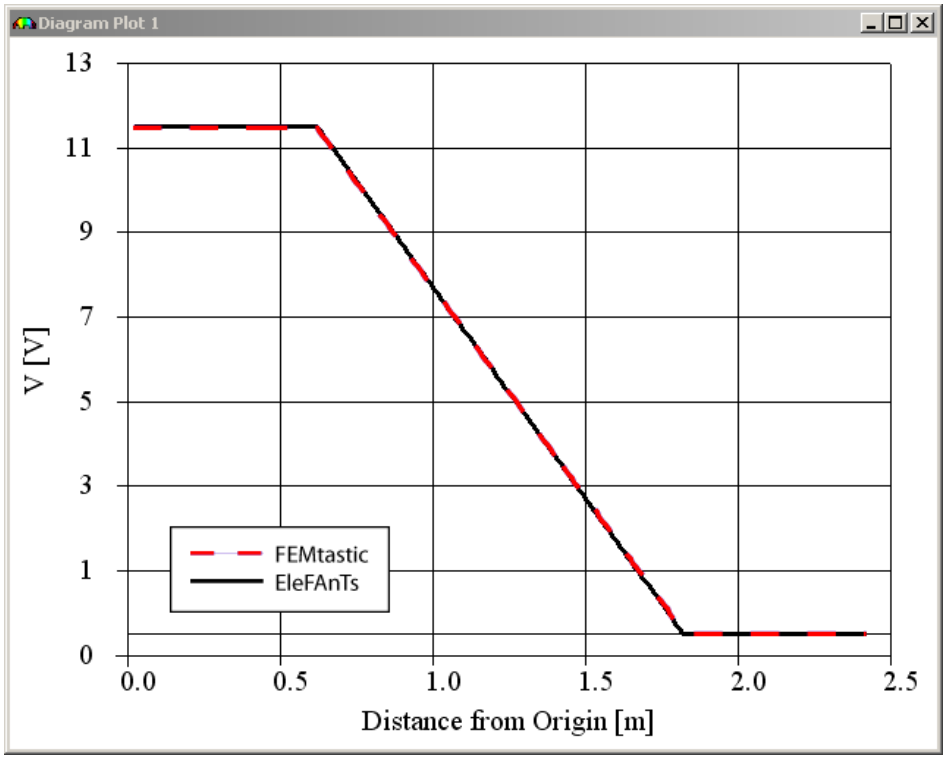

Figure 6.3.: Comparison of electric potential u along a straight line

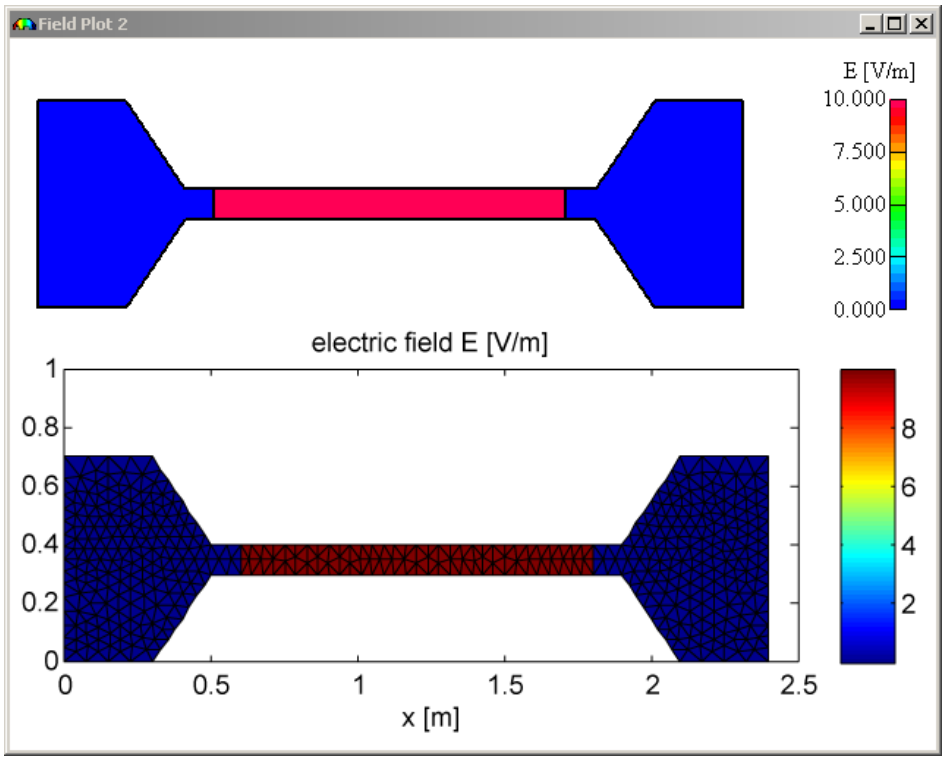

Figure 6.4.: Comparison of electric field $\mathbf{E}_x$ results (top: EleFAnTs, bottom: FEMtastic)

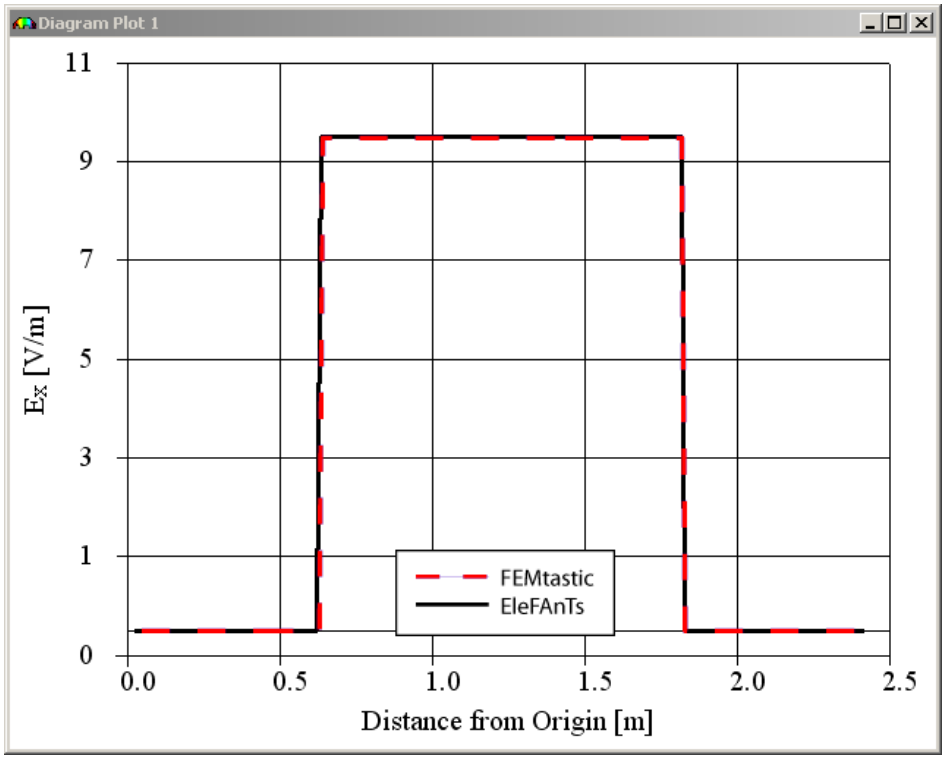

Figure 6.5.: Comparison of electric field $\mathbf{E}_x$ along a straight line

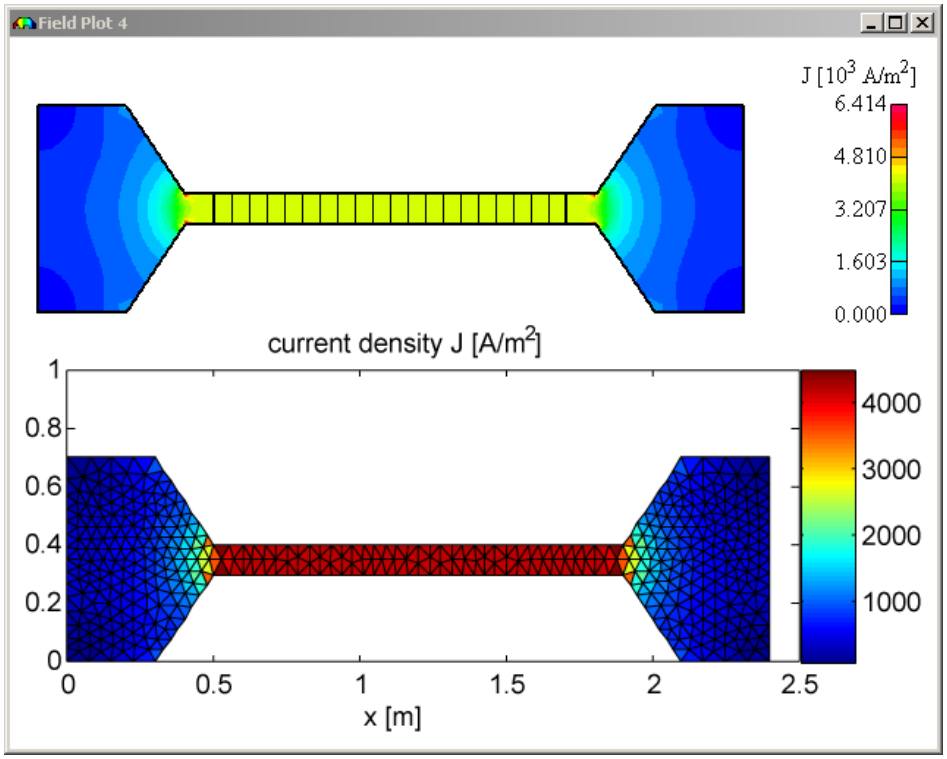

Figure 6.6.: Comparison of electric current flow $\mathbf{J}_x$ results (top: EleFAnTs, bottom: FEMtastic)

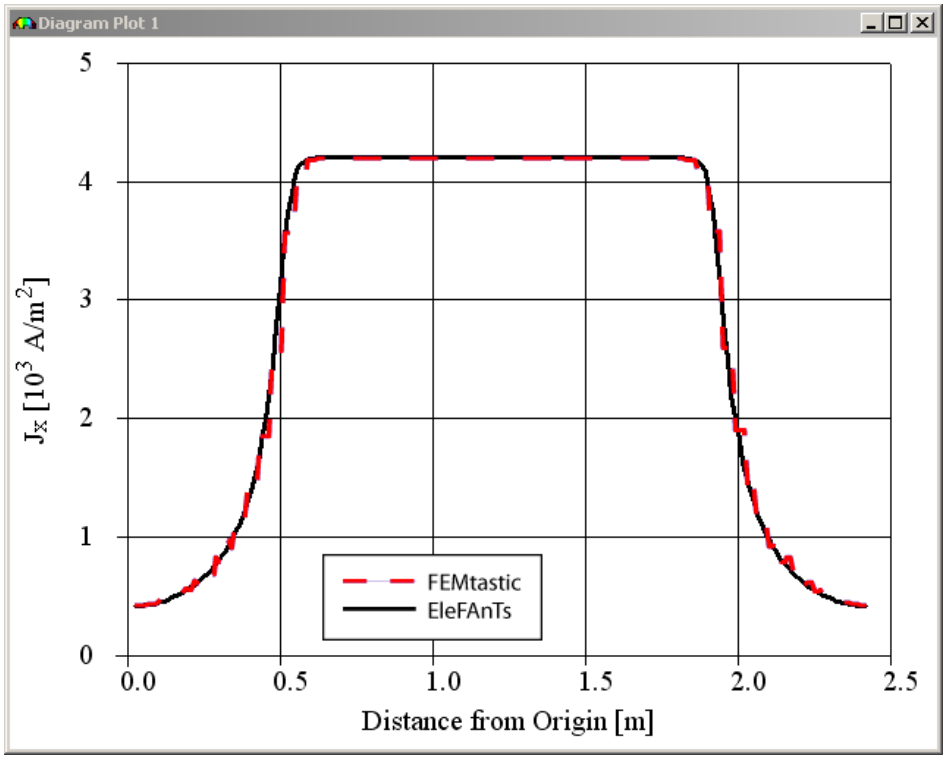

Figure 6.7.: Comparison of electric current flow $\mathbf{J}_x$ along a straight line

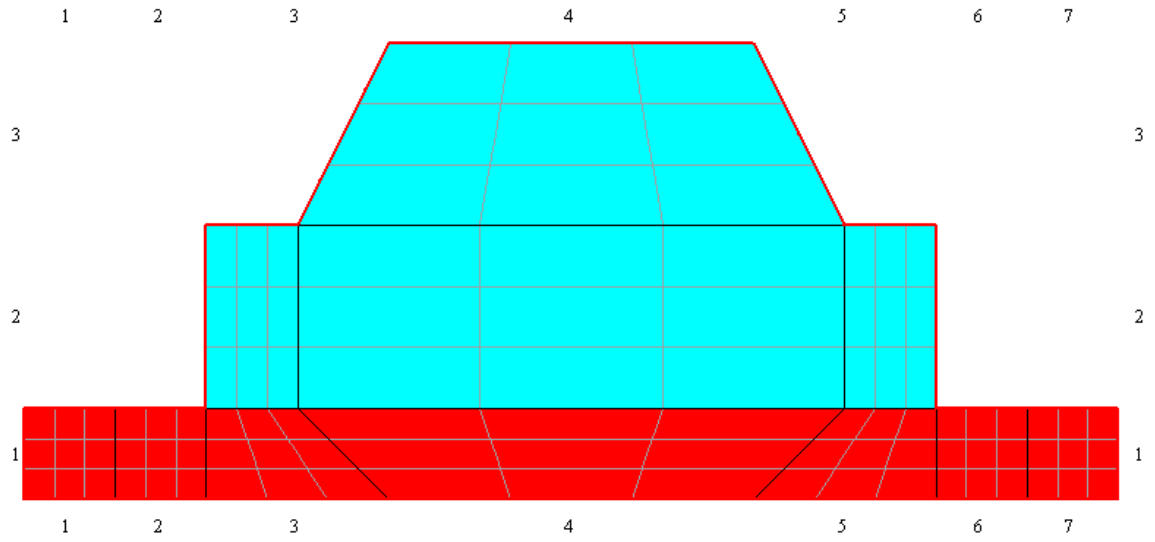

Figure 6.8.: EleFAnTs: thermal model

6.1.2. Thermal problem

Because EleFAnTs-2D does not support a thermal model to be coupled with a current flow model, the power loss within the heating plate of the current flow model was computed and used with the heating plate of the thermal model as a source with a total power of $5039.97W$ which was the result gathered from the current flow model. Because the current flow is fairly homogeneous within the heating plate this can be accepted.

As expected, the results with both EleFAnTs and FEMtastic are the same. Figure 6.8 shows the problem modeling with EleFAnTs, 6.9 compares the results.

6.2. MATLAB

MATLAB has been chosen as the development environment for some reasons. First, the object-orientation which was introduced with version 7.6 (R2008a) in March, 2008, was an unknown terrain and therefor interesting to discover.
A second reason is the claim that, with MATLAB programming language, engineering problems could be solved faster than with the likewise programming languages C++ and Java [32]. This statement has to be distinguished. It might be true for the overall process of developing, proof of concept and testing a new application. On the other hand, the integration and implementation of object-orientation into the MATLAB language not at all can be compared to typical OOP languages.
MATLAB language is still **interpreted** and not **compiled**, as are C++ and Java, and is therefor considerably slower. Due to its novelty, no recent benchmarks could be found comparing ”new” MATLAB with other OOP languages. Though, many benchmarking sites across the world wide web – e.g. [1] – conclude that there is a factor of 50 to 100 between, in general, slow interpreted and fast compiled programming code.
A third reason for the choice of MATLAB is its availability in plenty of educational facilities and an affordable student version.

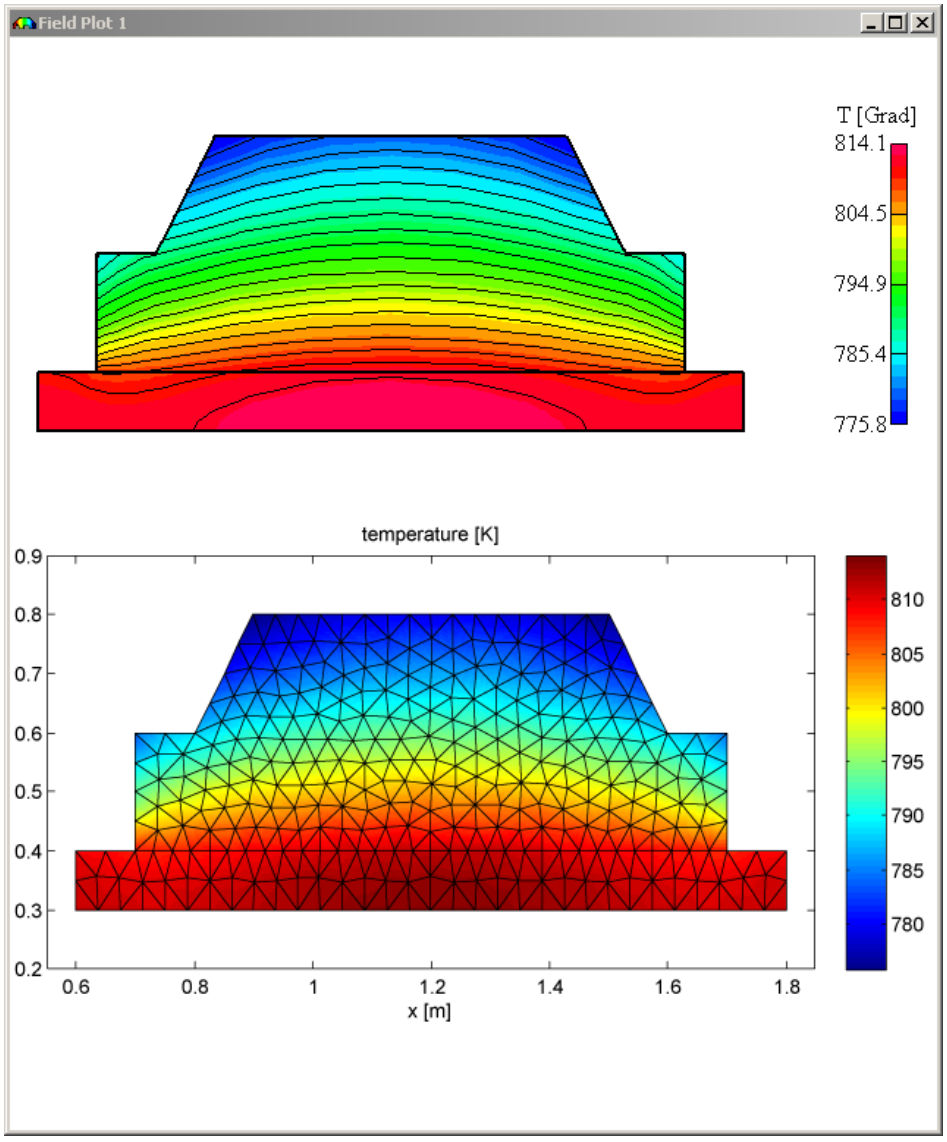

Figure 6.9.: Comparison of temperature T (top: EleFAnTs, bottom: FEMtastic)

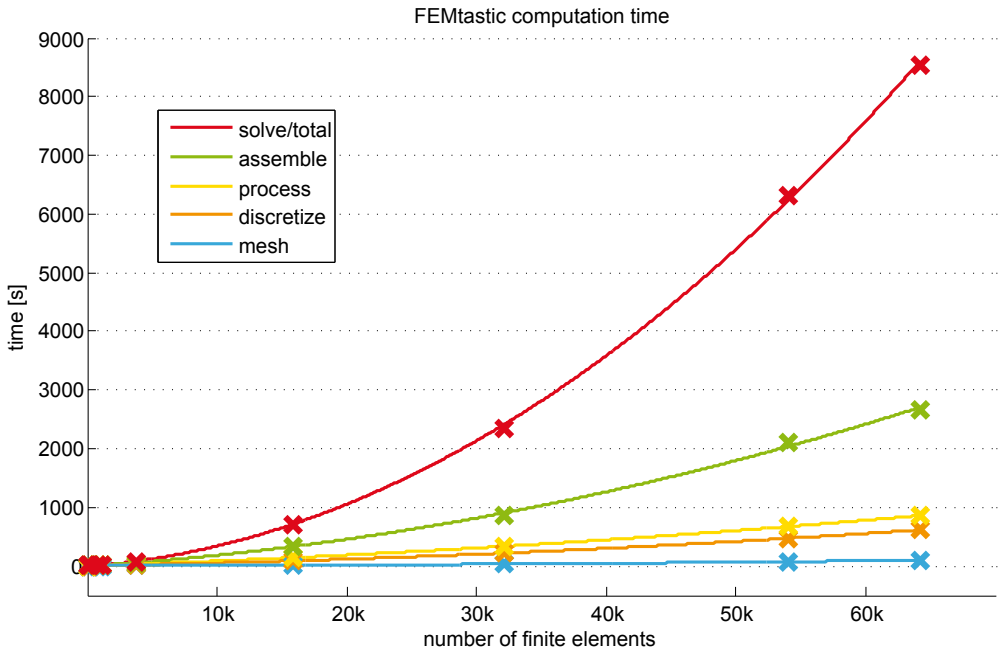

Figure 6.10.: Results gained with FEMtastic

Furthermore, it is an easy-to-learn language that does not require in-depth software engineers for understanding the nature of FEMtastic.

However, taking a look to the results concludes that the term *productivity* does not describe FEMtastic in its current state. Unfortunately, the choice for MATLAB must be blamed for this.

6.3. CPU time

For obtaining significant results, the current flow model from chapter 5 was meshed with different numbers of finite elements from 91 to 64188. The computation times of every step are cumulated to the total time (red top line) in diagram 6.10. The x-axis depicts the number of finite elements, actual measurements are illustrated with markers. The lines are interpolation polygons of second order. Latter fact is an advice for an efficiently implemented algorithm, because of the stiffness matrix that quadratically increases with the number of finite elements and therefor, also the computation time has to increase quadratically.
Nonetheless, porting the implementation to a different programming language must be one of the next steps in future developments.

Diagram 6.11 shows the percentage rate of the computation steps. It clarifies that the bottleneck of the application is the MATLAB inbuilt solver which, of course, is not optimized for the FEM system of equations and therefor is also subject of being replaced. For a stiffness matrix greater than $20k \times 20k$ it does already consume more than 50% of total CPU time.

A great amount of time within the system assembly is consumed by homogenizing the Dirichlet

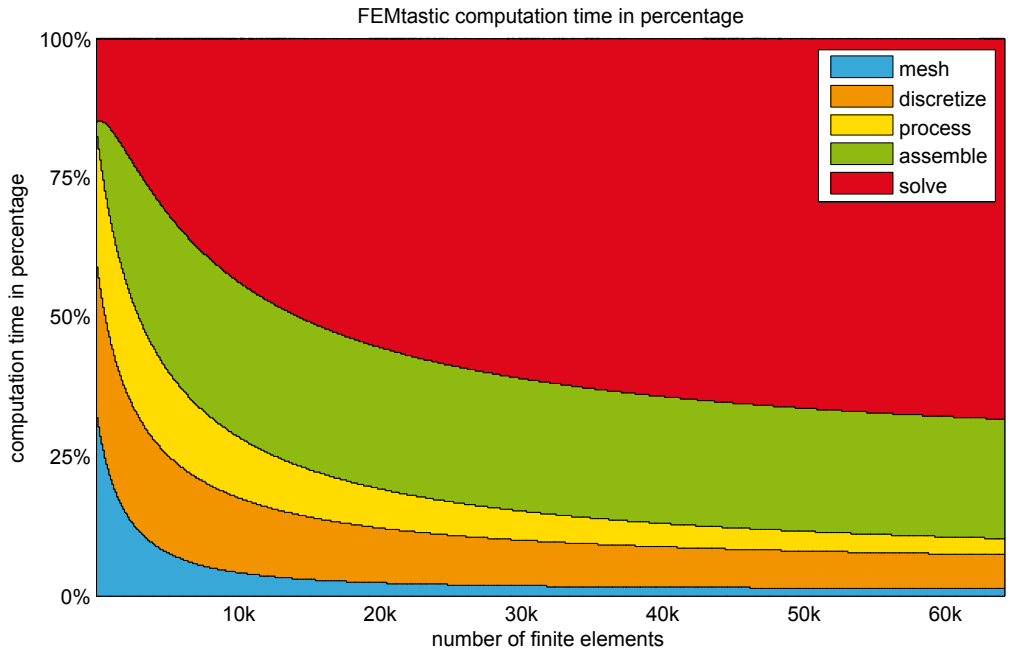

Figure 6.11.: FEMtastic computation steps percentage

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.: Simulation results (times in seconds)

boundary conditions (see 4.4.4 on page 32). Removing 514 rows and columns from the stiffness matrix with 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 6.11 can be seen as the preprocessing steps. The green and red areas together form the actual FEM computation.

The yellow, and surprisingly small, band in figure 6.11 represents the processing of all finite elements within a mesh in order to find the correct edges for Neumann and Cauchy boundary conditions (described in 5.5.1 on page 60). This was originally expected to be very inefficiently. Nevertheless, one could get rid of it by integrating these tests directly to the meshing algorithm.

Table 6.1 shows the detailed results of the measurements. These have been computed using an Intel T7250 CPU with 2 GHz clock speed, 2 MB L2 cache and 2 GB of working memory.

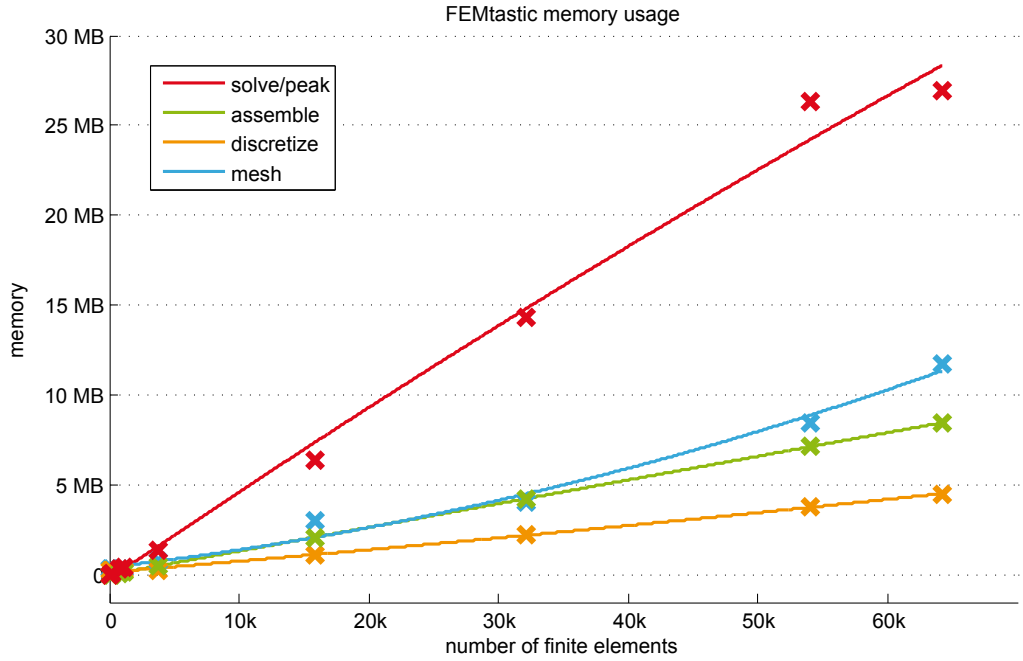

Figure 6.12.: FEMtastic memory usage

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	15822
peak	376	376	376	434	1366	6422	14355	26375	26933

Table 6.2.: Memory usage (values in KB)

6.4. Memory usage

Data is mainly stored as the finite element objects, the node objects, the stiffness matrix and the load vector. Additionally data is stored in some status and helper variables, of course. The stiffness matrix is a sparse matrix and only those values different from zero are actually stored to memory. Thus, the algorithm's memory consumption is linearly dependent to the number of finite elements. This is approved by figure 6.12 that shows the results from table 6.2.
The table summarizes the peak values of memory consumption of the subsequent tasks of the computation. Each task's memory is cleared after it has finished. Of course, the solver consumes the maximum amount of memory, though it is an interesting fact that the usage is also linear.

Figure 6.13 again shows the percentage of memory consumption amongst the tasks.

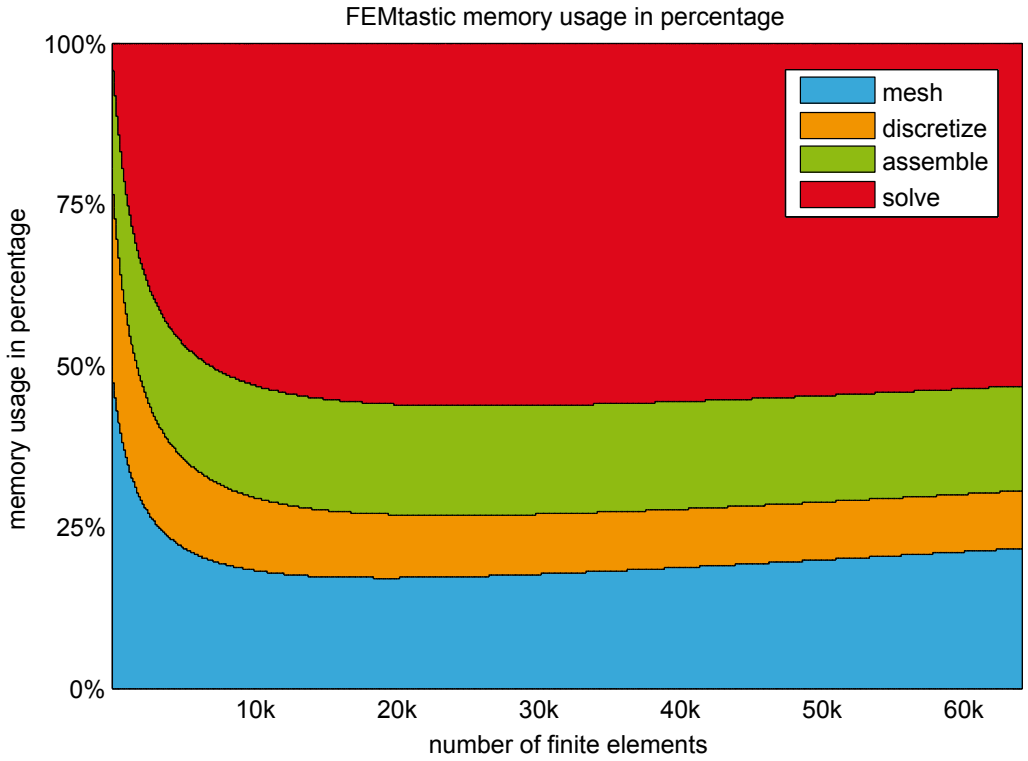

Figure 6.13.: FEMtastic memory usage percentage

6.5. Design

The design has proved to be very flexible in general. It makes a clear distinction between the modules and uses the object-oriented principles of encapsulation and inheritance very well. It was attempted to gather a high factor of cohesion between the classes and therefor reduce the amount of encapsulated data. This lead to a design which, of course, cannot make the demand of being the perfect design, but exposed to be enormously adaptive in making changes along the process of implementation, when some design tweaks were disclosed. Furthermore it also has already proved its capability of extendability. This was ascertained when the second T6 type of finite elements was brought into the system, which took only a few hours of implementation time.

As already mentioned, it is inevitable to port the implementation to a different programming language than MATLAB. However, an outstanding characteristic of MATLAB is that vectors and matrices are basic types which makes it different from any other programming language. These flexibilities, primarily need to be reviewed and also ported in a certain way to a different programming environment. As this will probably change the future design slightly, some additional hints are stated here which came up during the process of evaluation.

The actual modeling – namely bringing a real world problem into computer space – is done with three steps in FEMtastic:

- Formulating a real world problem in XML language.
- Preprocessing the XML file.
- Making objects out of the preprocessor status within the FEM module.

Within the first step it was discovered that the *XML* module and the *PREPROCpack* module are very balanced to each other. The preprocessor rebuilds the hierarchy of the XML tree. This is not generally poor but will make it a little bit difficult to construct further input interfaces. IGES (*Initial Graphics Exchange Specification*) [26] should be mentioned as one of possible interfaces here. It is a free, manufacturer independent data format that is widely used for interchange among computer-aided-design products.
Within the *FEMpack* module all finite elements and their coherences with the nodes are saved. Unfortunately the information of macroelements is lost within this module and consequently also for the postprocessing module. This incorrect decision was made due to the fact that for the actual FEM solution, macroelements are of no concern and only the scope of finite elements is of importance.
Thus, the suggestion is given to move the hierarchy of macroelements, finite elements, edges and nodes to an independent data structure that can exist auxiliary to other modules and implement respective interfaces as shown in [16].

6.6. Output interface

No words have been lost within this thesis concerning the output interface. The exterior optimization strategy and its implementation is still subject of initial development and no interface specifications have been done, so far. Thus, all data diagrams and functions from the postprocessor are considered to be the current output interface.

Appendix A

Appendix

A.1. Complete XML schema definition graph

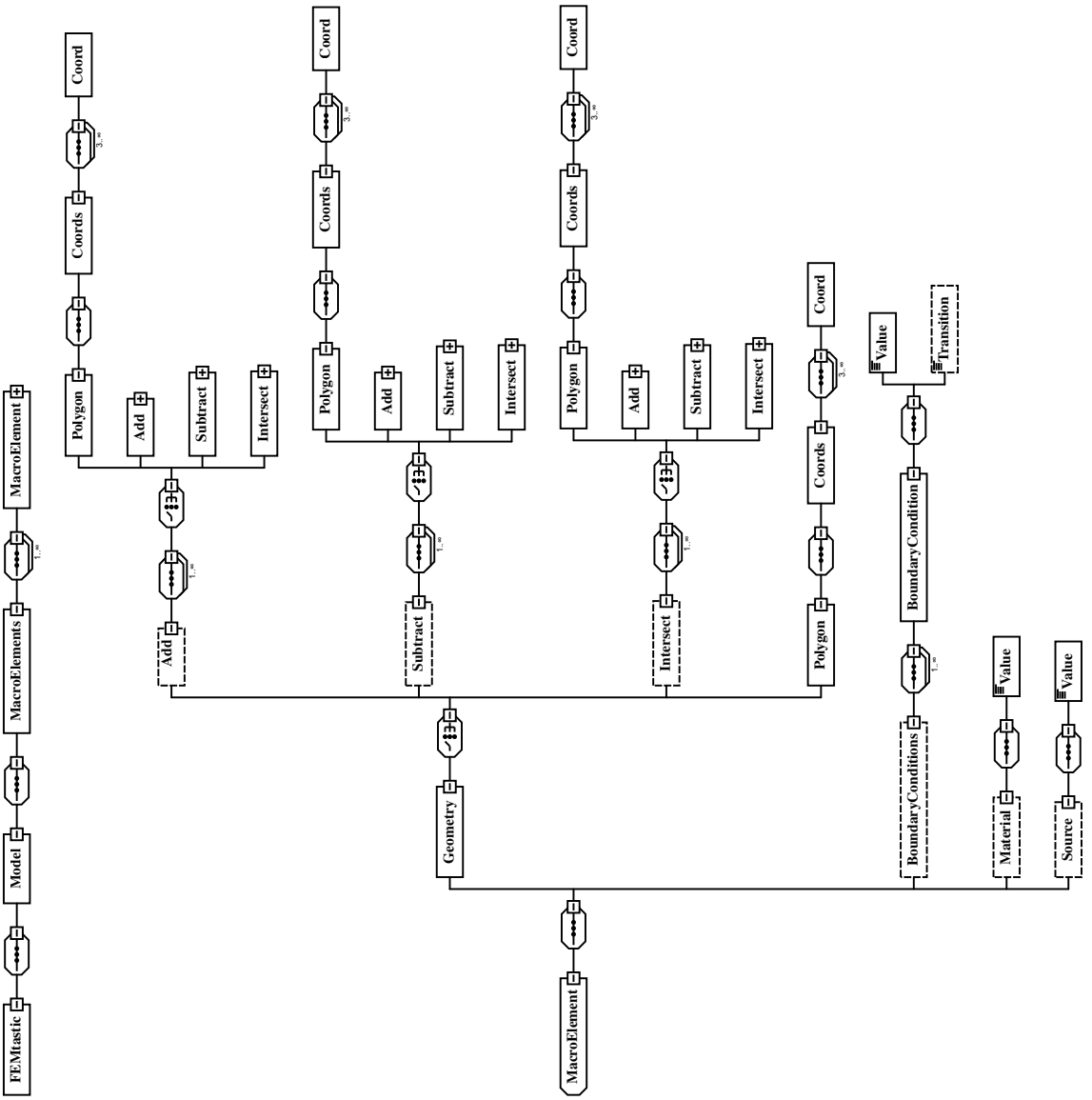

Figure A.1.: Complete XML schema definition graph

A.2. Geometry setup of an iron core with air gap

Listing A.1: XML-setup: iron core with air gap

```
<?xml version="1.0" encoding="UTF-8"?>
<FEMtastic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="FEMtastic.xsd">
  <Model type="electrostatics" refinement=".8">
    <MacroElements>
      <MacroElement>
        <Geometry>
          <Subtract>
            <Intersect>
              <Polygon>
                <Coords>
                  <Coord x="-5" y="-5" />
                  <Coord x="-5" y="5" />
                  <Coord x="5" y="5" />
                  <Coord x="5" y="-5" />
                </Coords>
              </Polygon>
              <Polygon>
                <Coords>
                  <Coord x="-9" y="0" />
                  <Coord x="0" y="9" />
                  <Coord x="9" y="0" />
                  <Coord x="0" y="-9" />
                </Coords>
              </Polygon>
            </Intersect>
            <Polygon>
              <Coords>
                <Coord x="-3" y="-3" />
                <Coord x="-3" y="3" />
                <Coord x="3" y="3" />
                <Coord x="3" y="-3" />
              </Coords>
            </Polygon>
            <Polygon>
              <Coords>
                <Coord x="-6" y="-1" />
                <Coord x="-6" y="1" />
                <Coord x="-2" y="1" />
                <Coord x="-2" y="-1" />
              </Coords>
            </Polygon>
          </Subtract>
        </Geometry>
      </MacroElement>
    </MacroElements>
  </Model>
</FEMtastic>
```

A.3. Setup of the hardening device current flow model

Listing A.2: XML-setup: Hardening device, current flow

```
<?xml version="1.0" encoding="UTF-8"?>
<FEMtastic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="FEMtasticDesign.xsd">
  <Model type="currentflow" refinement="1.5">
    <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>60E6</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</Value>
          </BoundaryCondition>
        </BoundaryConditions>
        <Material><Value>60E6</Value></Material>
      </MacroElement>
    </MacroElements>
  </Model>
</FEMtastic>
```

A.4. Setup of the hardening device thermal model

Listing A.3: XML-setup: Hardening device, thermal model

```
<?xml version="1.0" encoding="UTF-8"?>
<FEMtastic xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="FEMtastic.xsd">
  <Model type="thermal" refinement="1.5">
    <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" from="3" to="3"      >
            <Value>300</Value>
            <Transition>5.6</Transition>
          </BoundaryCondition>
          <BoundaryCondition type="Cauchy" from="5" to="5">
            <Value>300</Value>
            <Transition>5.6</Transition>
          </BoundaryCondition>
          <BoundaryCondition type="Neumann" from="1" to="2">
            <Value>0</Value>
          </BoundaryCondition>
          <BoundaryCondition type="Neumann" from="6" to="6">
            <Value>0</Value>
          </BoundaryCondition>
        </BoundaryConditions>
        <Material>
          <Value>85</Value>
        </Material>
      </MacroElement>
      <MacroElement>
        <Geometry>
          <Polygon>
            <Coords>
              <Coord x="0.7" y="0.4" />
              <Coord x="0.7" y="0.6" />
              <Coord x="0.8" y="0.6" />
              <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" />
            </Coords>
          </Polygon>
        </Geometry>
        <BoundaryConditions>
          <BoundaryCondition type="Cauchy" from="1" to="7">
            <Value>300</Value>
            <Transition>5.6</Transition>
          </BoundaryCondition>
        </BoundaryConditions>
        <Material>
          <Value>42</Value>
        </Material>
      </MacroElement>
    </MacroElements>
  </Model>
</FEMtastic>
```

Bibliography

[1] Dan Ballard. Primes results for x86 vs. ppc vs. arm. URL http://www.mindstab.net/wordpress/archives/184. [Online; accessed, 23-October 2010]. 103

[2] Oszkár Bíró. *Simulation statischer Felder - Vorlesungsskript*. Institut für Grundlagen und Theorie der Elektrotechnik, 2009. 24

[3] P. Castillo, R. Rieben, and D. White. Femster: an object oriented class library of discrete differential forms. volume 2, pages 972 – 975 vol.2, jun. 2003. doi: 10.1109/APS.2003.1219397. 8

[4] M.C. Costa, J.-L. Coulomb, and Y. Marechal. An object-oriented optimization library for finite element method software. *Magnetics, IEEE Transactions on*, 36(4):1057 –1060, jul. 2000. ISSN 0018-9464. doi: 10.1109/20.877623. xvii, 7

[5] Darren Engwirda. Unstructured Mesh Methods for the Navier-Stokes Equations. School of Aerospace Engineering, The University of Sydney, 2005. Undergraduate Thesis. 35

[6] Darren Engwirda. MESH2D - Automatic Mesh Generation, 2009. URL http://www.mathworks.de/matlabcentral/fileexchange/25555-mesh2d-automatic-mesh-generation. [Online; accessed 20-July-2010]. 35

[7] B.W.R. Forde, R.O. Foschi, and S.F. Stiemer. Object-oriented finite element analysis. *Computers & Structures*, 34(3):355–374, 1990. 6

[8] H. Fujio. The feelfem system: a repository system for the finite element method. page 8 pp., apr. 2003. doi: 10.1109/IPDPS.2003.1213461. 7

[9] E. Gomez, J.A. Fuentes, A. Gabaldon, and J. Roger-Folch. Fem applied to electromagnetism: effective object-oriented software design. page AU9, 2002. doi: 10.1109/INTMAG.2002.1000754. 5, 7, 8

[10] Guillaume Flandin. XMLTree: an XML toolbox for MATLAB, 2008. URL http://www.artefact.tk/software/matlab/xml/. [Online; accessed 30-August-2010]. 49

[11] S.R.H. Hoole and T. Arudchelvam. Reverse engineering as a means of improving and adapting legacy finite element code. pages 227 –232, dec. 2009. doi: 10.1109/ICIINFS.2009.5429859. 5

[12] IGTE. Institute for Fundamentals and Theory in Electrical Engineering, 2010. URL http://www.igte.tugraz.at. [Online; accessed 15-October-2010]. 5

[13] Jung, Michael; Langer, Ulrich. *Methode der finiten Elemente für Ingenieure*. Teubner, 2001. ISBN 3-519-02973-1. 23, 26, 27, 29, 31, 32, 34

[14] Kenneth H. Huebner. *The Finite Element Method for Engineers*. Wiley, 4 edition, 2001. ISBN 0471370789. 5, 13, 14

[15] R. I. Mackie. Object oriented methods–finite element programming and engineering. pages 133 – 138, 1995. 6

[16] W. Mai and G. Henneberger. Object-oriented design of finite element calculations with respect to coupled problems. *Magnetics, IEEE Transactions on*, 36(4):1677 –1681, jul. 2000. ISSN 0018-9464. doi: 10.1109/20.877765. xvii, 7, 8, 109

[17] Francisco Martínez, Antonio Jesús Rueda, and Francisco Ramón Feito. A new algorithm for computing boolean operations on polygons. *Computers and Geosciences*, 35(6):1177–1185, 2009. ISSN 0098-3004. doi: http://dx.doi.org/10.1016/j.cageo.2008.08.009. xviii, 35, 92, 93, 95, 97

[18] Stuart McGarrity. Object-oriented programming in matlab. URL `http://www.mathworks.de/mason/tag/proxy.html?dataid=11440&fileid=54059`. [Online; accessed, 16-October-2010]. 9

[19] Olgierd Cecil Zienkiewicz, Y.K. Cheung. *The Finite Element Method in Structural and Continuum Mechanics*. McGraw-Hill, 1 edition, 1967. 13

[20] Per-Olof Persson and Gilbert Strang. A Simple Mesh Generator in MATLAB. *SIAM Review*, 46:2004, 2004. 35

[21] Peter P. Silvester, Ronald L. Ferrari. *Finite Elements for Electrical Engineers*. Cambridge University Press, 3 edition, 1996. 14

[22] M. Popescu, I. Munteanu, C.-G. Constantin, and D. Ioan. An object oriented data structure for field analysis. *Magnetics, IEEE Transactions on*, 34(5):3403 –3406, sep. 1998. ISSN 0018-9464. doi: 10.1109/20.717801. 7, 8

[23] K. Preis, I. Bardi, O. Biro, R. Hoschek, M. Mayr, and I. Ticar. A virtual electromagnetic laboratory for the classroom and the www. *Magnetics, IEEE Transactions on*, 33(2):1990 – 1993, mar. 1997. ISSN 0018-9464. doi: 10.1109/20.582690. 6

[24] K. Preis, O. Biro, T. Ebner, and I. Ticar. An electromagnetic field analysis tool in education. *Magnetics, IEEE Transactions on*, 38(2):1317 –1320, mar. 2002. ISSN 0018-9464. doi: 10.1109/20.996336. 6

[25] Kurt Preis. *Simulation mechatronischer Systeme - Vorlesungsskript*. Institut für Grundlagen und Theorie der Elektrotechnik, 2007. 24

[26] The U.S. Product Data Association (US PRO). http://www.uspro.org/documents/iges5-3_fordownload.pdf. URL `http://www.uspro.org/documents/IGES5-3_forDownload.pdf`. [Online; accessed, 26-October 2010]. 109

[27] R. W. Clough. The Finite Element Method in Plane Stress Analysis. In *Proceedings of 2nd ASCE Conference on Electronic Computation*, Pittsburgh, PA, September 1960. 13

[28] V. Reinauer, T. Wendland, C. Scheiblich, R. Banucu, and W.M. Rucker. Object-oriented development and runtime investigation of 3-d electrostatic fem problems in pure java. pages 1 –1, may. 2010. doi: 10.1109/CEFC.2010.5481836. 8

[29] E.J. Silva, R.C. Mesquita, R.R. Saldanha, and P.F.M. Palmeira. An object-oriented finite-element program for electromagnetic field computation. *Magnetics, IEEE Transactions on*, 30(5):3618 –3621, sep. 1994. ISSN 0018-9464. doi: 10.1109/20.312724. 7, 8

[30] Strang, Gilbert; Fix, George. *An Analysis of The Finite Element Method*. Prentice Hall, 1973. ISBN 0130329460. 14

[31] Inc. The MathWorks. Latest features in matlab. URL http://www.mathworks.de/products/matlab/whatsnew.html. [Online; accessed, 16-October-2010]. 9

[32] Inc The Mathworks. Introduction to object-oriented programming in matlab. URL http://www.mathworks.de/mason/tag/proxy.html?dataid=4364&fileid=18842. [Online; accessed, 23-October-2010]. 103

[33] The StarUML development community. StarUML. URL http://staruml.sourceforge.net/. [Online; accessed, 02-October-2009]. 8

[34] M. J. Turner, R. W. Clough, H. C. Martin, and L. P. Topp. Stiffness and deflection analysis of complex structures. *J. Aeronautical Society*, 23, 1956. 5, 13

[35] Yang Xiang and Wanlei Zhou. The design and development of an integrated system for object-oriented finite element computing. pages 201 – 204, 2002. doi: 10.1109/ICAPP.2002.1173574. 6

[36] D. Yergeau, R.W. Dutton, and R.J.G. Goossens. An oo-pde solver for tcad apps. *Potentials, IEEE*, 21(2):25 –29, apr. 2002. ISSN 0278-6648. doi: 10.1109/45.997973. 7