DISSERTATION

On the Specification and the Assembly of Discretized Differential Equations

Ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften
eingereicht an der Technischen Universität Wien,
Fakultät für Elektrotechnik und Informationstechnik,
Institut für Mikroelektronik

von

Dipl.-Ing. Michael Spevak

[redacted]

Tulln, im Januar 2008

Kurzfassung

Die Lösung partieller Differentialgleichungen mittels Diskretisierungsschemen und algebraischen Methoden bildet einen breiten Forschungszweig im Bereich des Scientific Computing. Gemeinsame Schnittstellen für die Beschreibung einzelner Gleichungen sind jedoch in ihrer Allgemeinheit ebenso wenig angedacht wie generelle Methoden für die Assemblierung von Gleichungssystemen. Trotz zahlreicher Versuche eine Allzweck-Simulationsumgebung zu schaffen, gibt es keine einheitliche Methodik, mit der verschiedene Diskretisierungsschemen austauschbar verwendet werden können. Weiters verlangt das Aufstellen von Gleichungssystemen viele untergeordnete Schritte, die im einzelnen schwierig und mühsam durchzuführen sind und eine hohe Fehlerrate nach sich ziehen. In vielen Fällen sind die daraus resultierenden Fehler schwer zu entdecken und zu beheben. Das wesentliche Ziel dieser Arbeit ist es, die eingangs genannten Probleme zu lösen und eine Methodik zu entwickeln, mit welcher die Spezifikation diskretisierter Differentialgleichungen durchgeführt werden kann, sowie Strukturen zur Behandlung von Gleichungen und für das Aufstellen von Gleichungssystemen bereitzustellen.

Ein allgemeines topologiebasiertes Konzept zur Beschreibung verschiedener Diskretisierungsschemen sowie eine gemeinsame Beschreibungssprache für die funktionale Beschreibung von diskreten Zusammenhängen wird entwickelt. Verschiedene Differentialgleichungen können mittels unterschiedlicher Diskretisierungsschemen mit demselben Formalismus beschrieben werden, welcher von der funktionalen Programmbibliothek Phoenix2 abgeleitet und im funktionalen Teil von GSSE implementiert ist. Darüber hinaus werden funktionale Operatoren zur Durchführung der Akkumulation von Summanden bereitgestellt, welche die Summanden in benachbarten topologischen Elementen auswerten. Die Gesamtheit dieser Funktionen kann in beliebiger Kombination mittels einer in C++ eingebetten Sprache abgerufen und verwendet werden.

Für das Aufstellen von Gleichungen wird das Konzept von linearisierten Ausdrücken verwendet, welches einfach in den oben beschriebenen Formalismus eingebettet werden kann. Es werden Datenstrukturen verwendet, welche neben einem numerischen Wert die linearen Abhängigkeiten eines entsprechenden Ausdrucks nach einzelnen Variablen angeben. Dies ist sowohl für lineare als auch für nichtlineare Verfahren von Vorteil, wobei die Berechnung von Ableitungen, die bei der Linearisierung zwangsläufig ist, automatisch durchgeführt wird. Bei der Lösung von nichtlinearen Gleichungen können residuenbasierte und gradientenbasierte Verfahren verwendet werden. Eine weitere Schnittstelle für Eigenwertgleichungen kann analog verwendet werden.

Abstract

The solution of partial differential equations using discretization schemes and algebraic methods is a very important topic of scientific computing, which lacks common interfaces for the description and the assembly of discretized differential equations. Even though many approaches towards a general purpose simulation environment have been implemented, an environment which enables the use of different discretization schemes is still lacking. Furthermore, the assembly of such equations requires many subsequent tasks which are error prone and cumbersome. Such errors are often hard to detect and only occur under certain circumstances.

The main aims of this thesis is to overcome the described difficulties and provide a method for the specification of discretized differential equations as well as structures for the assembly of algebraic equation systems.

A common topologically based framework for the specification of different discretization schemes is developed and a description of the functional specification layer for discrete and discretized equations is shown. Commonly used differential equations can be discretized using the provided framework. The basic features of the discrete formalism are derived from a given functional library Phoenix2 as well as the GSSE topological layer GDL. Furthermore, accumulation operations are introduced which enable the developer to calculate sums over, e.g. neighboring elements which fulfill a topological property. In contrast to available functional frameworks, non-local formulae can be evaluated. Therefore traversal operations can be used freely by the implementer and are not implicitly assumed by the functional framework. Due to a `C++` domain specific embedded language, the formulation is concise and short.

For the assembly, a framework of linearized expressions is provided, that can be easily combined with the specification framework mentioned above. These data structures store

linear dependences of the residual expressions specified for the discretization schemes. Therefore, the specification of the residuum with its linear dependences offers the possibility to assemble an equation system automatically, without calculating the derivatives of the residuum with respect to the single solution variables by hand. For the solution of nonlinear equation systems, gradient based methods can be employed easily and the linearization for each step is performed automatically. An interface for the treatment of eigenvalue equations can be used analogously.

Danksagung

Zuerst möchte ich Herrn Professor Siegfried Selberherr für seine Unterstützung beim Schreiben dieser Arbeit danken. Er hat mich während meiner gesamten wissenschaftlichen Tätigkeit am Institut für Mikroelektronik wie auch beim Abfassen dieser Arbeit stets unterstützt und motiviert. Ich bin ihm dankbar für seine Geduld und die Motivation, die er mir gerade im letzten Jahr zukommen ließ.

Ich möchte auch meinen beiden ehemaligen Kollegen René Heinzl und Philipp Schwaha für wertvolle Gespräche und sehr unkonventionelle Ideen bedanken, die das Gedeihen dieser Arbeit erst ermöglichten.

Ich danke Herrn Professor Tibor Grasser für zahlreiche Anregungen, Hinweise und Diskussionen.

Weiters möchte ich noch Herrn Professor Erasmus Langer für die Gelegenheit danken, mein Doktorstudium am Institut für Mikroelektronik durchzuführen.

Contents

List of Symbols

General Naming Conventions

Symbols which refer to vectors are given in bold font ($\mathbf{a}$). Symbols which refer to matrices are given in capitalized bold font ($\mathbf{A}$). Symbols which refer to geometric entities are given in calligraphic font ($\mathcal{P}$). Symbols which refer to function spaces are given as calligraphic font ($\mathcal{F}$). Symbols which refer to sets and spaces are given in bold letters ($\mathbf{e}$). Symbols which refer to topological elements of a cell complex are given in bold font ($\mathbf{e}$). Symbols which refer to named variables are underlined ($\underline{e}$).

Discrete Problems and Discretization

Numerical

$\mathcal{R}$	set of real numbers
$\mathcal{E}$	set of numerical values within a numerical data type
a_i, b	elements of the numerical set $\mathcal{E}$
g	granularity of a data type
$\mathbf{A}$	system matrix of a linear equation system
$\mathbf{x}$	solution vector of a linear equation system
$\mathbf{b}$	right hand side vector of a linear equation system
κ	condition number of a linear equation system
N	dimension of a function space
$\mathcal{F}$	function space of cell-wise linear shape functions
f_i	i-th basis function of $\mathcal{F}$
$\mathbf{v}$	vector of weights, e.g. for the basis functions of $\mathcal{F}$
v_i	i-th element of $\mathbf{v}$
$\mathcal{F}'$	subspace of continuous functions within $\mathcal{F}$
f_i'	i-th basis function of $\mathcal{F}'$
$\mathcal{G}$	second/third order function space
g_i	i-th basis function of $\mathcal{G}$
$\mathcal{G}'$	subspace of continuous/smooth functions within $\mathcal{G}$
g_i'	i-th basis function of $\mathcal{G}'$
f_i^{l}	local shape functions of $\mathcal{F}$
$\mathcal{Q}_i,$	collocation point of a function
$\mathcal{I}$	triangle
x, y, z	coordinates
$\mathcal{P}$	geometric points

n, p	carrier concentration for the semiconductor equations
ψ	electrostatic potential for the semiconductor equations
$\mathcal{D}$	simulation domain
$\mathcal{T}$	topological space
t_i	elements of the topology
$\mathcal{H}$	basis of the topology
$\mathcal{S}$	sub-basis of the topology
h_i	elements of the basis of the topology
s_i	elements of the sub-basis of the topology
$\mathcal{C}$	cell complex
n	dimension of the cell complex $\mathcal{C}$
$\sim$	binary incidence relation
$\mathcal{O}(\bullet, \bullet)$	orientation function
q	quantity defined on the cell complex $\mathcal{C}$
$\mathbf{x}_1$, $\mathbf{x}_2$	coordinates of points
c	curve function c
r	parameter for the curve function c
a, b	free parameters
$\oplus, \ominus, \otimes$	second order binary relations for arithmetic operations
$\mathbf{2}$	second order constant function returning a function returning 2
Sin	second order sine function
$\mathbf{u} = \mathbf{u}_1$	unnamed function returning the first argument
$\mathbf{u}_2, \mathbf{u}_3, \ldots$	unnamed function returning the 2nd, 3rd, ... argument
$\underline{v}, \underline{w}, \ldots$	named functions for vertices
$\underline{e} \ldots$	named functions for edges
$\underline{c} \ldots$	named functions for cells
$\bigoplus$	count of elements
$\bigsqcup$	vectorization symbol
$\delta_{i,j}$	Kronecker symbol

Differential Equations

$\mathcal{L}$	linear or linearized differential operator
$\mathcal{M}$	boundary functional
Δ	Laplace operator
grad	gradient
div	divergence
u	solution function
w	finite element weighting function
R	residual expression
C	net doping concentration
ϕ	flux quantity for finite volumes
G	Green's function
δ	Dirac impulse function
$\mathcal{T}$	triangle
x_i, y_i	coordinates of the triangle $\mathcal{T}$
$[\partial]$	vector of derivatives for finite differences
J	Jacobian of the finite element transformation
K, K'	coupling coefficient function
A	surface size of the dual surfaces
l	quantity of edge length
m	cell volume of the dual cells of a vertex
$\Delta\psi$	potential difference for the finite difference method
$\mathcal{G}$	geometry matrix for the finite difference method
$[\partial]$	vector of derivatives for the finite difference method
$\mathbf{u}$	function value vector for neighboring vertices for the finite difference method
opp	edge opposing a vertex within a triangular cell
dual	associated element of the dual Delauney graph
$\mathcal{B}$	Bernoulli function
D	derivative vector for the finite difference method
$\mathcal{O}(\bullet)$	Landau symbol

Algebraic Equations

x_i	unknown variables
$\mathbf{x}$	vector of unknown variables
r_0	right hand side of a linear expression
$\mathbf{b}$	right hand side vector
$\mathbf{E}$	algebraic equation system
N	number of equations and unknown variables
$\mathbf{x}_0$	linearization vector
λ	eigenvalue
$\lambda(\bullet)$	discrete eigenvalue function
$\mathbf{K}$	local stencil matrix of coupling coefficients
$\hom(\bullet)$	homogeneous linear function
R	residuum
q	solution quantity
Q	linear expression of the solution quantity
K	coupling coefficients
q_B	Dirichlet boundary value
α, β, γ	boundary coefficients
k, k'	boundary coupling coefficients
ϕ_B	boundary flux
ϕ	flux along an edge
ε	permittivity
κ	thermal conductivity
u	solution function
ϱ	thermal capacitance
V	potential
$\mathbf{v}_B$	Boundary vertex
BI	function that returns an inner, neighboring vertex of a given boundary vertex
VF	function that returns the first boundary vertex
$\mathbf{B}F$	function that returns the first of several boundary vertices

List of Figures

Chapter 1

Introduction

In the field of scientific computing which includes a large variety of different disciplines, it has turned out to be of utmost importance to find a concise and broadly accepted formalism which provides means for different, commonly used structures.

This is essential in order to obtain reproducible results, because in some cases the simulation process as well as the results are provided in a manner that the external observer is not able to redo the respective simulation experiments. The steps which lead to the solution of the problem are not sufficiently disclosed. As a first step, a classification scheme for scientific computing simulation processes is introduced with which simulation tools can be categorized.

Many implementations and software packages comprise algorithms for the solution of problems from many different disciplines. Each software project is focused on a very special method which is implemented in a highly optimized manner, whereas the implementation of the remaining tasks is often neglected. A large number of scientists is involved in the development of new algorithms, data structures, algebraic methods, or discretization schemes used for the solution of partial differential equations. In many cases these efforts lead to an improvement regarding one of these topics, whereas for the other fields methods and implementations of state-of-the-art approaches or even approaches of worse quality are used, often due to ignorance of the neighboring fields.

Even though there are several approaches of supporting large software packages [1], common interfaces are not available that would assist a scientist skilled in one special field to use the power of libraries in order to fulfill high standards in all of the mentioned fields.

Currently, each kind of software introduces new interfaces, which are designed especially for a certain purpose and can not be used for comparable methods due to interfacing problems, so that the implementation of similar tasks often requires a complete re-design of the software and interfaces. This of course worsens the situation of a scientist, who is trying to implement a special method that inevitably depends on the use of third party software. In order to circumvent these difficulties it is common practice to use scripting languages to write the "gluing code" between the single parts of code, which quickly solves the problem at first glance.

However, there is a lot of software already glued together and even though scripting languages of high performance are used [2] to manage the bottleneck of bringing data from one application to another, these scripts become more and more cumbersome to manage. The implementation is highly specific gluework which is more difficult to handle and therefore often software has to be re-designed, when more external software components are required. Especially problematic is the change of external software components by newly developed software of higher performance, because neither the developers of the software, nor the developers of the used respective library have designed interfaces independent from the actual data structures. Therefore the lack of interoperability and substitutability of libraries causes suboptimal simulation software.

In order to identify missing features, a classification scheme is introduced in Section 1.1, which shows the general steps of a scientific computing application. This classification also gives a practical orientation on how to separate software into parts and how to categorize existing parts of code. Interfaces are addressed and the main problems regarding inappropriate use of these interfaces are described. In Section 1.2 known frameworks commonly used for the solution of scientific problems are compared and analyzed with respect to this classification. Finally it will be shown, that most parts of scientific computing are well solved as standalone problems such as the solution of algebraic equation systems. However, the specification of discretized differential equations as well as the interfacing still lacks proper specification methods. Based on this analysis, Section 1.3 formulates the aims of this thesis, namely a proper specification layer for differential equations as well as discretized algebraic equations.

1.1 Classification of Simulation Processes

There exists a vast number of computer programs as well as programming frameworks which can be used in order to perform computer simulation. Programs use different kinds of approaches, solve different physical problems, various types of mathematical processes, and use a large variety of available algorithms and data structures.

The first step towards a rigorous specification and classification for the different kinds of programs used for computer aided engineering is to provide a layer model of abstract steps which are performed in a computer simulation. It is not necessary to make use of all of the layers provided in this layer model for all situations. In this section an approach towards a layered model is presented, which provides a clear separation of different steps which have to be performed, when designing a simulation program.

Another aspect of this model is to share workload within a group of scientists who have different expertise in various kinds of scientific computing. The reason for this separation using clearly defined interfaces is to establish a concise method of interaction between different scientists. Even though the mutual understanding is often very poor, the use of such interfaces enables scientists to establish proper computational methods.

For one person - in contrast - scientific computing is hardly overlookable and the development of methods in all different fields involved in the simulation is time consuming and does, therefore, not lead to optimal results. For this reason a sensible separation of competences has to be established in order to balance the workload.

The fields covered by scientific computing are modeling, discretization, algebraic methods, and program design, where program design can be considered as necessary for the implementation of methods of all other fields, which form the theoretical background. Program and data structural design is primarily required to define clear and concise interfaces between the single theoretical fields. For this reason a data structural interface is found for modeling, discretization and algebraic methods.

Modeling is required to find a mathematical model for a physical phenomenon of interest. It is typically carried out by physicists and engineers, who have a concrete question in mind. As an example for modeling the Schrödinger equation can be considered, which

perfectly describes the behavior of particle systems with very in small scales under the influence of a potential.

Discretization is a general term for methods which make continuous problems solvable for the computer. Typical methods are the finite element method or the finite volume method. The solution of a linear or a nonlinear equation system as well as the solution of an eigenvalue problem is a typical representative of an algebraic method.

Even though the formulation of the problem in a programming language and the design of data structures which represent the mathematical structures used are relevant for the implementation of software, the layers show the abstract steps which have to be performed, but not how they have to be implemented.

Once these fields of scientific computing are separated, the layers can be easily defined. The original question is posed to the engineer or physicist who uses modeling in order to provide a continuously formulated problem which is discretized and solved by a computer aided mechanism. After the solution is obtained by the computer it is often checked for validity in a mathematical manner. This means that the solution fulfills the given mathematical equation. Furthermore the physical accuracy has to be checked, which means that the physical model is used within its boundaries of validity.

Most of the available models inherently have restrictions which have to be considered before the solution can be validated. In the following the main focus is put on the abstract methods rather than the implementation. Even though the use of computers is an essential aid when solving large equation systems and software is an essential means for making the problems concise and easy to formulate, the mathematical and physical problems can be also performed in an abstract manner or - in some cases - even by hand. The main focus is put on the separation of the single stages, where each of the stages can be simplified by the computer.

It has to be considered that all these mental steps can be carried out independently from each other. While some steps naturally induce a special treatment in the underlying layers, a premature restriction on some methods unnecessarily restricts the variety of possible solution approaches. Fig. 1.1 shows the single steps of the simulation as well as the layers of the classification.

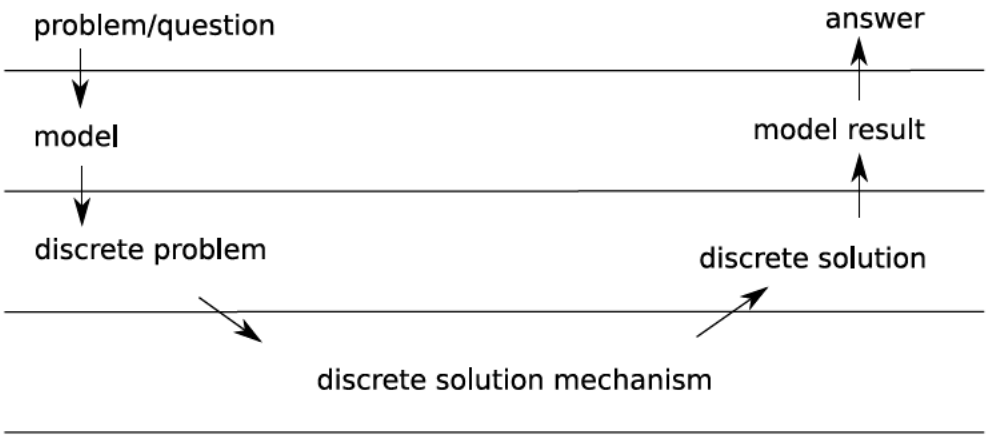

Figure 1.1: Layer model for the classification of simulation processes.

1.1.1 Engineering Problems

At the start of the considerations one faces an engineering or even a management problem which has to be solved. Such a problem can be posed in a rather abstract and non-physical manner. In general, at this level a rigorous consideration does not play an important role at all, whereas strategic deliberations are more ostensible. The kinds of questions which are posed may be related to the effects which are caused by the configurations and situations in question.

The most common reason for raising physical questions is to obtain an optimized method, device, or process so that a specific, non-scientific goal can be reached. In most cases the focus is not even put on the immanent physical effect of the device but on a more abstract, mostly human related effect. Typical engineering questions which are the starting point of a computer simulation are:

- *How can the power consumption of a microchip be minimized?*
- *How can CO_2 emissions be reduced?*
- *What is the optimum traffic regulation means for a city?*
- *What is the optimum workload-balance system for a facility?*
- *What is the optimum shape of a high voltage insulator?*

- *How do I earn most money with it?*

It is clear that such an engineering problem cannot be answered by a state-of-the-art computer program, because the computer lacks an intrinsic in-depth knowledge of the specific scientific field. Moreover, automated answering of these questions is only possible for a very small range of specified problems.

1.1.2 Modeling

The solution of each engineering problem requires knowledge in a field such as applied physics, chemistry, process technology, civil engineering, construction, medicine, or any other. Modeling is required in order to describe the physical effects relevant for answering the engineering problem. All effects which have to be taken into consideration are comprised by the model chosen. A typical result of the modeling process is the mathematical formulation of the desired physical phenomena, for instance as partial differential equation.

In general, many different models are available for the solution of a problem which differ in fidelity and complexity, each of them offering a tradeoff between complexity and fidelity of the solution obtained by the model. For this reason different levels of detail are employed depending on the phenomena that are interesting to the preceding engineering problem.

The implications of mathematical considerations on the model have to be kept to a minimum and should only affect aspects such as overall solvability and perhaps uniqueness of the problem. At this stage it is not necessary to define the discretization scheme as well as the required algebraic solution methods. A premature restriction to a discretization and an algebraic method unnecessarily leads to limitations of usability and applicability, and other mathematical methods which are developed later on cannot be incorporated into the original program.

Example: *It can be shown that (non-relativistic) electrostatic problems (engineering) in general result in diagonally dominant symmetric matrix problems. In fact two different problems have been solved in one step, namely the modeling of the physical behavior of a given configuration using electrostatic equations and the subsequent discretization of*

the resulting differential equations (elliptic self-adjoint equations). Between these steps an interface is used, namely the continuous problem formulation. At this interface a separation between modeling and the further processing of the model equation(s) can be established. The advantage of this separation is that the details of the model and the further processing steps can be considered as black-boxes.

1.1.3 Continuous Problems

A possible interface for the specification of the model obtained is given by continuous physics. Even though there are other approaches possible, most engineering problems can be transformed into continuous problems. In continuous problems a domain $\mathcal{D}$ is defined and some physical quantities such as velocity or temperature are given as field on the domain $\mathcal{D}$. Even though unknown, the solution quantity is assumed to be a field over the continuous domain $\mathcal{D}$.

In order to obtain the solution from given quantities some governing relation is given. This can be a simple point-wise operation, a differential equation, or an integral operation such as a convolution.

Continuous problems as they appear in modeling form an interface between modeling and mathematical treatment. In many cases these continuous formulations are used as an interface between two scientists or an interface between a human and a computer, e.g. for the case when mathematical software is used. In each case clearly defined and easily understandable formalisms have to be provided in order to avoid unneccessary overhead and insufficient information at the interface.

Another aspect of interfaces is that information which is passed through (e.g. between an engineer and a mathematician) inevitably changes its semantics. While modelers have a clear idea of the physical behavior of the investigated system, this usually does not hold to a mathematician and is perhaps not even relevant. Moreover, it is necessary to find a discretization scheme which works for the posed equation and yields correct solutions.

1.1.4 Discretization

Discretization is required for obtaining an appropriate solution of a mathematical problem. It is used to transform the initially continuous problem which has an infinite number of degrees of freedom (e.g. eigenfunctions, Green's functions) into a discrete problem where the degree of freedom is inevitably limited. This limitation is necessary due to the finite nature of the subsequent calculation process.

Function spaces, required to form all possible solutions when applying arbitrary initial as well as boundary conditions, have an infinite dimension. Accordingly, the representation of a solution vector is not possible within the computer. The step of discretization is used to find a function space with a reasonable finite number of base functions, which comprises a proper approximation of the analytical solution.

In many cases base functions or shape functions with a local support are used. In this case the rather abstract task of finding shape functions can be reduced to the simpler task of finding a finite tesselation of the simulation domain.

At this point the physical origin of the mathematical formulation is neglected. The following steps do only rely on the posed mathematical model of the physical problem. This avoids post-implementation fittings which are not based on the continuous mathematical formulation but on the intuition of the engineer.

The following examples show alterations of mathematical and physical methods which are valid within the layer structure, because they origin from the mathematical formulation only.

Example 1: *The use of a functional expression, which characterizes the physical behavior, for instance the mobility carrier of a material, is a purely physical adaption and results in changes of the posed differential equation and is a modeling approach.*

Example 2: *The Scharfetter Gummel Discretization [3] is a typical mathematical adaption of a discretization scheme with respect to the mathematical character of the functions n, p and ψ. Even though the intentions initially did have physical nature, the method itself is formulated only by relying on the underlying differential equations. Further physical effects are not introduced.*

All changes of the final discretization formulae due to physical reasons, which cannot be directly derived from the underlying physical model and its mathematical formulation are doubtful in their physical as well as mathematical meaning, because this is neither implied by a concise modeling approach, nor do such methods use a proper mathematical solution procedure. Of course, there is a certain chance to obtain acceptable results for special cases, but the overall credibility of such a system is corrupted due to the fact, that neither the mathematical methods nor the model can be benchmarked for validity and reproducibility of the results is not given.

It has to be stated that many approaches have combined the discretization (including assembly) of differential equations and the solution of the resulting algebraic problems. Even though at first glance it seems to be straight-forward to combine these two steps, one must remark that the implementation of all combinations, for instance for testing the optimum method, results in a growing implementation overhead.

1.1.5 Discrete Problems

After the discretization of a differential equation has been performed, the equation is reduced to a system of algebraic equations. In general, there are three main classes of discrete problems which are commonly known in scientific computing:

- Linear equation systems
- Nonlinear equation systems
- Eigenvalue equation systems

Apart from finiteness, discrete problems which are the result of discretization in general have the same algebraic structure of their associated problems. This means that in most cases a linear problem (e.g. a linear differential equation) results in a linear equation system and a continuous eigenvalue problem becomes discrete but remains an eigenvalue problem.

The discrete problem formulation is an interface between the discretization scheme and the algebraic method used for further processing. The layer model avoids to pass parameters

from the discretization methods to the algebraic methods. If a discretization scheme provides solution parameters to a certain kind of algebraic solution method, the software products for discretization and solution can be only used in this very special combination. Once a new algebraic method is more apt to deal with the discrete problem, the specially determined parameters become obsolete or other parameters are required which implies additional implementation overhead. If a number of discretization schemes is thus bound to one solver interface, the use of another solver interface or even the use of an update of the solver interface with new interfaces might require an updating of all discretization methods.

1.1.6 Algebraic Methods

In order to solve algebraic problems, for instance a linear algebraic equation system or an eigenvalue problem, special algebraic methods have to be used. For the solution of discrete problems, a large variety of such solution methods are available. One such software package is Trilinos [4], which covers many different methods and offers high flexibility of solution mechanisms as long as Trilinos internal methods are used. Perhaps the most annoying fact related to these environments is that all algebraic environments use different software interfaces and a large effort has to be spent on making the single environments compatible with each other.

For this reason it is impossible to separate the discrete algebraic problem from the algebraic methods used. As some simulation environments even combine discretization schemes with algebraic methods, it is nearly impossible to separate the discretization, the data storage of the discrete problem, and the algebraic methods.

In the layered model the origin of the mathematical problem does not have any impact on the method used. Instead, only the discrete problem has to be taken into account.

1.1.7 Discrete Solution

With the aid of algebraic methods the discrete solution of a problem is obtained from the discrete problem. At this layer the solution consists of a vector of numbers which has to fulfill the requirements of the given algebraic equation system. When eigenvalue

equations are solved, a number of different eigenvalues and the associated eigenvectors are provided. This vector is returned to the discretization layer where the solution vector is considered as weighting vector for basis functions.

If discrete modeling is used, the discrete solution itself becomes relevant. In this special case of a discrete model, which is often used in industrial simulation or electrical circuit simulation, for instance for finding an optimum workload balance for a facility, the model does only contain discretized data and does not have any functional information on distributed quantities.

1.1.8 Mathematical Solution

The coefficients of this solution vector are multiplied with the basis functions and the sum of the weighted basis functions is the appropriate solution of the discretized problem. If functions with local support with a maximum value 1 are used, the stored numbers can be interpreted as function values in the points where the respective shape function has its maximum. However, many methods such as the boundary element method show that this is not the case in general. Many methods exist which do not use function spaces with this special property and therefore, cannot be interpreted in this way.

It has to be stated that once the function space is determined in the process of functional discretization, there is no way of changing the shape of the base functions due to e.g. physical or mathematical reasons after the calculation is performed.

Even though this seems to be clear from the mathematical point of view, many post-processing tools, which are not directly involved in the discretization process, provide different means of obtaining the original function by different interpolation methods. In order to obtain the correct function the interpolation method has to be used which also was used in the discretization step. This is of special relevance, if differential terms such as gradients have to be evaluated in “non-grid” points.

1.1.9 Mathematical Error Estimation

The step of mathematical error estimation has two main purposes: First and most straight-forward, the error in the total simulation result shall be estimated and – if pos-

sible – reduced. Second and more relevant, the error can be localized and assigned to separate shape functions. This implies that these functions are inappropriate for forming the solution of a given problem and in general have to be altered. In terms of functions with local support the regions of support are divided into subregions where new functions are generated, which are used for a subsequent discretization and solution of the problem. The measures which are taken for measuring the error are so-called a-posteriori [5, 6, 7, 8, 9] error estimation methods. These methods are proven to work on a class of linear problems whereas their correctness cannot be proven in general. Despite this fact these methods are commonly used in order to have a heuristic method for refining and fitting shape functions.

1.1.10 Physical Interpretation

After the mathematical solution is obtained and proven or assumed to be adequate for the posed problem, the mathematical solution may be interpreted in a physical manner. Here, all purely mathematical values gain a physical meaning. As an example, vector functions or vector fields [10] may be interpreted as velocity fields. The physical solution can be compared to an experiment which is carried out, e.g. to verify or refine the simulation.

Mathematical solutions are typically free of physical units and perhaps scaled with respect to a given system of units, for instance the SI. Constants which occur in physical problems have to be scaled to a special system of units, e.g. the vacuum dielectricity constant ε_0 has to be scaled with respect to the scaling of the occurring voltages, distances, time steps, and currents.

For this reason, simulations results where no such special constants occur explicitly, e.g. the Laplace equation (in its mathematical interpretation), may be valid for various simulation problems. However, it has to be remarked that derived quantities, e.g. field strength, depend on the scaling used. The physical interpretation can result in scaling the complete simulation result due to a re-scaling of all stored values.

1.1.11 Physical Model Validation

All physical models have a range of validity where models are designed to obtain a simplified description of some physical mechanism under certain circumstances. If these assumptions are incorrect, the solution of the model might be wrong. Typical examples for physical assumptions are listed below.

- Low masses or energies for non-relativistic models
- Lattice temperature equals carrier temperature for isothermal models. Otherwise higher order models [11] must be used.

Several measures might be taken in order to avoid problems with physical models becoming invalid. In most cases the model is replaced by a refined and perhaps more elaborate or detailed model with extended range of validity. Sometimes it is possible to use refined models only in regions where extended parameter ranges are required.

1.1.12 Answering the Initial Engineering Question

As a last step the initial engineering question has to be answered. In many cases the visualization of the mathematical or physical solution is sufficient to give a correct answer. Some questions, e.g. the detection of thermal breakdown, can be answered by looking at the heating curve. In some cases the answer is not trivial to find or more simulations have to be performed, which makes answering the question cumbersome or even impossible.
In many cases engineering questions seek to find some optimum configurations of devices or methods. In this case, many simulations have to be performed and their results have to be post-processed. From these results further simulations have to be started and an optimization loop is run. After a certain criterion for accuracy is reached, the loop is terminated and the final solution is retrieved.
There is a large variety of optimization processes available which are partly integrated into simulation environments [12] or available as stand-alone optimization framework [13], which can be used for arbitrary underlying simulation tools.

1.2 Related Work

As the main objective of this thesis is the convergence of scientific computing and simulation mechanisms, this work aims to combine many different topics.

First, functional programming and general purpose lambda implementations are treated. Available implementations have several difficulties and disadvantages with respect to topological data structures.

Second, topological implementations are considered. In many cases the topological implementation relates to one particular incidence relation and can, therefore, be seen as implementation of one type of discretization.

Algebraic solution mechanisms are relevant for this thesis as they are immanently necessary for the implementation of scientific simulation software. Even though the actually used methods can be seen as a black-box method from the discretization point of view (except for optimization), the interfaces for accessing the matrices are considered.

As related topic highly automatic solution frameworks such as FEMLab [14] can be considered. Even though these frameworks have a high flexibility at the modeling level and a large variety of differential equations can be formulated, the discretization is performed by one scheme only, so that many solution methods for special equations can not be investigated.

`MATLAB` is very a powerful general purpose environment with a large number of implemented functions. It can be used for a large variety of mathematical problems and provides comfortable features [15]. Additionally, many software packages or libraries that are programmed using MATLAB are available to tackle single specific tasks such as the solution of different kinds of differential equations based on physical phenomena, for instance hydrodynamics or thermodynamics. Furthermore, discrete simulation models can be used, for instance neural networks [16], electrical circuit analysis [17] (e.g. for signal processing), or image processing tools [18].

However, it has to be stated that the `MATLAB`-language itself only provides imperative programming methods and rudimentary object related methods. The specification of the formulae is directly written into the program code. Compared to the generic programming

approach [19] which is typically used for `C++` and `STL` [20] applications the flexibility is rather limited. The functions used require special memory layouts of the passed arguments and further development for data types with higher performance are prohibited.

The last section comprises the framework GSSE [21, 22] which is an implementation of the mathematical concepts presented in this thesis. Moreover, GSSE forms a software basis for the considerations of Chapter 3 and Chapter 4 as it allows an abstract topological treatment of arbitrary cell complexes.

1.2.1 Functional Languages for Formula Specification

Functional programming is a technique which enables the formulation of functions which can be combined in order to form compound functions. Finally, one of these functions is applied to the desired data structure. This function contains all relevant information required to determine, e.g., a discretized equation. A formalism can be obtained with which discretization operations can be carried out.

In C++ [23] different libraries have been developed which use object-oriented and generic programming [24] methods in order to emulate functional programming. The basic concept which makes functional programming possible in C++ is the function object [23] or functor. The standard template library [25] which is an integral part of C++ offers very basic methods for the implementation of function objects, such as binders and higher order functions. The lack of operator overloading makes the use of the functional parts of the STL rather cumbersome and inconcise for the specification.

The Boost Lambda library [26, 27] offers basic functionality like operator overloading, partial function evaluation, higher order functions, and unnamed variables. The FC++ library [28] offers the treatment of functions of arbitrary order.

The Boost Phoenix 2 library [29] which is used for the implementation of frameworks offers explicitly named variables which extremely eases the implementation of accumulation operations required for the discretization and assembly of differential equations. Furthermore, it offers access to arbitrary containers, the implicit use of C++ loops, and conditional expressions.

1.2.2 Structures Handling Topological Data

First, the frameworks are discussed which deal with discretized differential equations using distinct discretization schemes on a given topological framework.

One of the main inspirations for the development is the GrAL framework [30, 31]. In this work the main ideas of topological traversal as well as treating the underlying cell complex with topological methods are given. Implementations exist for many different problems regarding scientific computing.

SGFramework [32] is a mathematical framework especially designed for the solution of partial differential equations using the finite volume method. Various differential equations can be specified whose discretization can be formulated using finite volume schemes as described in Section 2.3. A similar framework which is intended and designed for the solution of drift-diffusion [33] semiconductor equations is Prophet [34].

Roxie [12] is a design environment for the electromagnetic optimization of accelerator magnets. Many optimization features are treated which allow to solve a (quite restricted) engineering problem, namely how to obtain a magnetic field which is possibly identical with that of a given multipole in a large field of an accelerator magnet.

A topological framework for treating triangular and tetrahedral meshes is the wafer-state server (WSS) [35]. This framework offers the possibility of treating quantities which are related to vertices as well as a very generalized approach towards segmentation and the use of subsets of the given cell complex. The topology treatment is reduced to cells and vertices and the only incidence relation available is to obtain all vertices which are incident with a common cell. For these reasons the application to methods which do not use cell based element matrix assembly is complicated and requires various workarounds.

The smart analysis programs (SAP) [36] are implemented which provide a high performance solution of a small number of equations such as the Laplace equation or diffusion equations.

There are several special purpose environments for the solution of very special kinds of differential equations, especially the Navier Stokes equations for fluid mechanics [37, 38]. These environments typically do not offer many different configuration features and are

specially optimized for this single purpose. The topological cell complex implementation is often directly integrated into the assembly process which makes these tools apt for this special purpose. For general studies on discretization schemes such environments are not intended.

Other tools which also involve geometrical features of the respective cell complex are mesh generators and computer geometry algorithms [39]. In contrast to the generic scientific simulation environment (GSSE) [40], which is designed for flexible topological operations and quantity treatment, these environments offer flexible methods for changing the geometry as well as re-meshing given domains.

The simulation environment `dealII` [41, 42, 43] is intended as a rapid prototyping tool for finite element simulation and offers methods for mesh refinement and error estimation. The specification of discretization schemes is usually done by writing the local element matrices for finite elements of arbitrary order. `dealII` provides own mesh refinement strategies, different topological elements, and shape functions of different order.

Especially for the use of graphs the Boost Graph library [44, 45] is developed. This library contains various algorithms for graphs and implements STL concepts of accessing data which are associated with vertices and edges. Functional structures can be defined in order to formulate local expressions on single nodes.

1.2.3 Algebraic Solvers

Software for the solution of algebraic equations is required after the discretization of a given differential equation has been assembled. The typical and mostly explored field is the solution of linear equation systems. For such systems various implementations are available. An established standard for the treatment of linear equations is LAPACK [46], which defines a set of interaction functions at different levels.

PETSC [47, 48] and Trilinos [4] solvers have a general purpose interface for accessing and solving linear equation systems and eigenvalue equation systems. Many different operations for assembly are possible and various solution methods can be used and exchanged without great effort, for instance, by changing a parameter.

Even though the diversity of different methods used for the solution of eigenvalue problems is considerable and special methods are used for special problems, there are approaches for an intelligent and highly automatized approach towards the solution of algebraic problems. First, the large variety of methods of PETSc and Trilinos allows to test and evaluate different solution methods and parameter settings. In addition, special expert systems like EigAdept [49] have been developed to automatically solve algebraic equations using the best available method. Such environments enable to use an algebraic solver as a black box, while the user does not necessarily need to specify the solution method.

1.2.4 Automatic Environments

For the solution of partial differential equations, which often also interacts with modeling of physical and engineering processes highly automatized environments are widely used. For the solution of physical problems only the respective differential equations with the respective parameters have to be given and a geometry has to be specified.

Ansys [50] is a professional approach which offers an unexperienced user approved numerical methods and discretization schemes for the solution of simulation problems at a physical level. Standard simulations can be carried out without deeper knowledge of the process of discretization and solution of the respective equations, whereas the main focus is put on user-interoperability. It can be integrated directly in a computer aided design and engineering process and offers methods which are generally approved and tested under a large number of circumstances.

For the development of new models and discretization methods these methods are generally not appropriate, because many settings which would have to be set are at a low level which requires deep insight into the numerical details of the simulation. This, however, contradicts the initial aim of providing a high level simulation tool.

General purpose computer algebra programs like FEMLab [14] offer different methods of high level access to differential equations. In contrast to Ansys, these frameworks are intended to solve the respective problems at mathematical layers, even though physical interpretations and add-ons exist.

1.2.5 Conclusion

It has been shown that the introduced frameworks have different focuses with respect to the classification shown in Section 1.1. Fig. 1.2 shows the flexibility of related software tools within the single layers introduced in the former section.

It can be seen that none of the investigated tools provides high flexibility in the discretization layer as well as in the interfaces to the discretization layer. One of the main reasons is that the discretization strongly relies on the underlying topological structure which is often restricted to special access methods. The following section shows an appropriate topological framework with which these operations are possible and the discretization can be carried out.

1.2.6 GSSE

This work is based on the implementation of the Generic Scientific Simulation Environment [40, 51, 22], which contains methods for handling the underlying topological data structures as well as storing quantities with respect to these structures. Furthermore, a functional layer is provided with which one is allowed to formulate functional expressions based on the quantities associated with the topological elements as will be shown in Section 2.3.

The implementation of the topology handling is shown in detail in [21] and forms the base of the considerations of Section 2.3. The main advantages of this library over other libraries is that arbitrary topological elements can be used and quantities can be associated with any topological element. Such a framework is necessary in order to provide a functional calculus as will be shown in Section 2.5.

The second major part is a functional programming layer as discussed in Chapter 5. The main idea behind this layer is to provide a mathematical description language to formulate expressions, especially for discretization schemes. Furthermore, an additional indirection allows to formulate residual expressions and to implicitly obtain all derivatives necessary in order to assembly the required system matrix.

Figure 1.2: Flexibility of related software tools within the single layers introduced in Section 1.1.

1.3 Aims

Tools regarding the discrete specification of scientific problems as well as an abstract treatment of interfaces to both, discretization schemes as well as algebraic solution methods (solvers), are required. In special cases all of these tasks can be implemented in an appropriate manner, however, the flexibility of the implemented code is rather low. Even though a high flexibility can be obtained with single solvers and the handling of discrete topological complexes, an appropriate framework for interfacing these features is still missing.

The aim of this work is to provide a mathematical framework for the discretization, linearization, and matrix interfacing, which closes the gap between flexible and highly performant algebra tools and topological tools for the handling of discrete structures with which discretization schemes can be realized.

For the discretization or the formulation of discrete problems, a functional calculus is provided, with the main aim to preserve the topological opportunities which are offered by the topological structure implemented in the GSSE library. Furthermore, one of the design goals is that with the aid of this calculus it has to be possible to implement different kinds of discretization schemes. As a consequence different methods can be compared, while the implementation effort is kept to a minimum.

By storing the expressions for different discretized differential operators in a library it is easy to use pre-formulated expressions in order to solve one's own differential equations. Within such a library different methods regarding modeling, discretization, and algebraic solution of the respective problem can be tested and optimized.

The next aim is to ease the effort for the specification of derivatives with respect to single solution variables for the assembly of a discrete algebraic equation which is, especially for the implementation of finite volumes, most cumbersome and error prone. Additionally, the formulation of the respective formulae shall remain as simple as possible and no separate framework for the specification shall be required. Furthermore, the assembly of matrices should be treated in a manner that the solver used and its respective matrix format can be used in combination of the other methods.

The overall goal is an arbitrary combination of methods explained in the above listed aims. If combined with an abstract solver interface, a large variety of different problems can be approached using such a framework.

Chapter 2

Discrete Problems and Discretization

Methods designed to transform continuous mathematical problems - in most cases partial differential equations – into algebraic problems are called discretization methods. Such a continuous problem is usually given by a model for the governing physical processes which are determined by the physical reality to be investigated. The problem is given on a continuous space which is usually a subset of $\mathcal{R}, \mathcal{R}^2, \mathcal{R}^3, \ldots$. This means that one is faced with at least an ordinary differential equation (ODE) or a partial differential equation (PDE) that optionally contains temporal derivatives.

Discretization considers the fact that analytical methods which are known from functional analysis cannot be directly applied to functions stored in a computer, and an assumption on the functional structure of the single elements has to be made.

One main aspect of this chapter is the development of a formalism which is designed for writing expressions in the context of discretization. It is shown that such a formalism can be used for a concise definition of the topological as well as the functional structure of the respective discretization formulae. This formalism can be used for the specification of computer programs, when such a formalism contains sufficient information to carry out all operations which are required for the evaluation of the formulae.

2.1 Numerical Discretization

The result of floating point numerical operations inevitably leads to a certain error, namely the numerical discretization error or truncation error. This error affects all calculations that are carried out with floating point arithmetic.

A necessary restriction on computational data types is that only a finite memory space can be used in order to represent a numeric value. This implies that only a finite number of numeric values can be represented while most of the numeric values of the solution can only be approximated. After a set $\mathcal{X}$ of real numeric values is defined each real value that has to be treated numerically has to be approximated by members of the set $\mathcal{X}$. This inevitably results in a numerical truncation error which is made when the real values are replaced by members of the set $\mathcal{X}$.

2.1.1 Numerical Discretization Error

In most cases floating point data types are used which have the drawback that each single operation imposes a numerical error. Under some circumstances such a behavior can result in an accumulation or in an extinction of numerical errors. An introduction into the effects of numerical calculations and the difficulties can be found in [52].

There exist data types which try to imitate the behavior of rational numbers [53], they generally represent the numerator and the denominator of a fraction and use adaptive integer numbers. By the determination of the greatest common divisor, for instance by the Euclidian algorithm, a cancellation can be performed and the adaptive integers can be kept small.

Even though these calculations are possible in principle, adaptive integer arithmetic operations are slow compared to the highly optimized floating point operations. Moreover, it has to be stated that the closedness of the operation is limited to the memory consumption. There are some numbers, whose sum or product can not be stored due to memory limitations.

In contrast, computational data types typically used are not dense and do have some given granularity g. The granularity is the size of an interval in which no number of the

given data type is located. When using floating point arithmetic numbers, this interval also depends on the size of the numbers within which the interval is searched.

The numerical error which is made by choosing a discrete representation of a number instead of the "real" number can be estimated by $1/4g$, where the error is equally distributed from $-1/2g \ldots +1/2g$. For each operation which does not exactly lead to a value within the set $\mathcal{X}$ of data type numbers, an approximation within the set $\mathcal{X}$ has to be determiend.

2.1.2 Errors of a Numerically Discrete System

When performing numerical operations on one input variable, the error can be estimated a in straight forward manner. In contrast, the handling of a system of equations is more difficult to estimate. Methods which are used to solve algebraic problems have to be considered with respect to their behavior when dealing with truncated or rounded numerical values. For the following considerations the equation system

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.1}$$

is used, where $\mathbf{A}$ is the matrix, $\mathbf{x}$ is the solution vector and $\mathbf{b}$ is the right-hand-side vector. The typical measure to describe an equation system is the so-called condition number [54].

$$\kappa := \text{cond}(\mathbf{A}) = \|\mathbf{A}\| \, \|\mathbf{A}^{-1}\| \tag{2.2}$$

This condition number states the deviation of the solution vector of a linear equation system measured by its norm with respect to the right hand side vector of the equation system [54].

$$\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x}\|} < \kappa \frac{\|\Delta \mathbf{b}\|}{\|\mathbf{b}\|} \tag{2.3}$$

Such an estimation is useful, if the right hand side is added a numerical error, e.g. noise caused by discretization. If numerical noise is added to the matrix, which is typically the case when material parameters or geometrical coefficients for discretization schemes have

to be determined via floating point computations including all occurring errors, another consideration can be used [54]:

$$\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x}\|} < \frac{\kappa}{1 - \|\Delta \mathbf{A}\| \, \|\mathbf{A}^{-1}\|} \frac{\|\Delta \mathbf{b}\|}{\|\mathbf{b}\|} \tag{2.4}$$

Errors of this kind are considered in the numerical treatment of algebraic methods as shown in detail in [54]. In the following considerations, the numerical error is not further considered, because the main focus of this work is the specification of formulae rather than the numerical treatment or the solution of special problems.

2.2 Functional Discretization

While in the former section the problems with the representation of a number have been discussed, this section considers the treatment of functions in a computer. The main idea of this section is that a function space with a given number of basis functions is used. Each function is defined as linear combination of basis functions. In general, the basis functions fulfill some basic properties such as continuity, smoothness, etc., which is also preserved by the linear combination.

When these functions are stored in the computer, a function space comprising a finite number of basis vectors is introduced. Each function of this function space can be described as weighted sum of basis functions, f_i where the weighting coefficients w_i are stored as a vector-like data structure $\mathbf{w}$, while the basis functions of the function space are assumed implicitly. Therefore, the function space covers all different functions which can be written as a linear combination of the following form

$$f(\mathcal{P}) = \sum_{i=1}^{N} w_i \cdot f_i(\mathcal{P}) \ . \tag{2.5}$$

The function space therefore represents the multitude of all possible functions which can be treated in the computer. From this point of view it is clear, that once the function and a special basis space are defined, only the coefficients need to be stored, whereas the shape of the basis functions is implicitly assumed.

In many cases the function space is defined in the following suggestive manner which is typical for finite difference schemes: Given a set of points $\mathcal{P}_i$, each of the base functions f_j of the function space $\mathcal{F}$ is associated to one of the points $\mathcal{P}_i$ in a way that the following equation holds true, where $\delta_{i,j}$ denotes the Kronecker symbol.

$$f_j(\mathcal{P}_i) = \delta_{i,j} \ , \tag{2.6}$$

For this reason, each weighting factor can be directly associated with the value of the function and the vector of these factors is understood as the set of points and their associated function values. Such an interpretation is very suggestive, because the rather abstract view on shape functions is replaced by direct function values, where the main problem is, however, that the information of the functional behavior between the points is lost. Differentiation, quadrature or the calculation of functionals is not possible from this point of view, because the information on the basis functions is ignored.

Finite element schemes [55] as well as boundary element schemes [56] define the underlying shape functions and then use form functionals, namely integrals, in order to determine the respective discrete dependences between the weighting factors of the functions. Even though the method can be written as an equation of weighting factors, the coupling of weighting coefficients strongly depends on the shape of the basis functions.

In contrast, finite difference schemes [57] and to some extent also finite volume schemes [33, 58] do not explicitly define the underlying function space. They mainly rely on the discrete function values which are defined point-wise. Care has to be taken that for each discretization step the same interpolation scheme is used.

2.2.1 Functions and Discrete Representations

Each function which can be represented in a computer can also be written as linear combination of basis functions of a function space. A weighting vector is given which comprises the single weighting coefficients for the basis functions.

As any other function, such a function has to provide a function value, in general a scalar or a vector value when a point of the simulation domain consisting of coordinates is passed

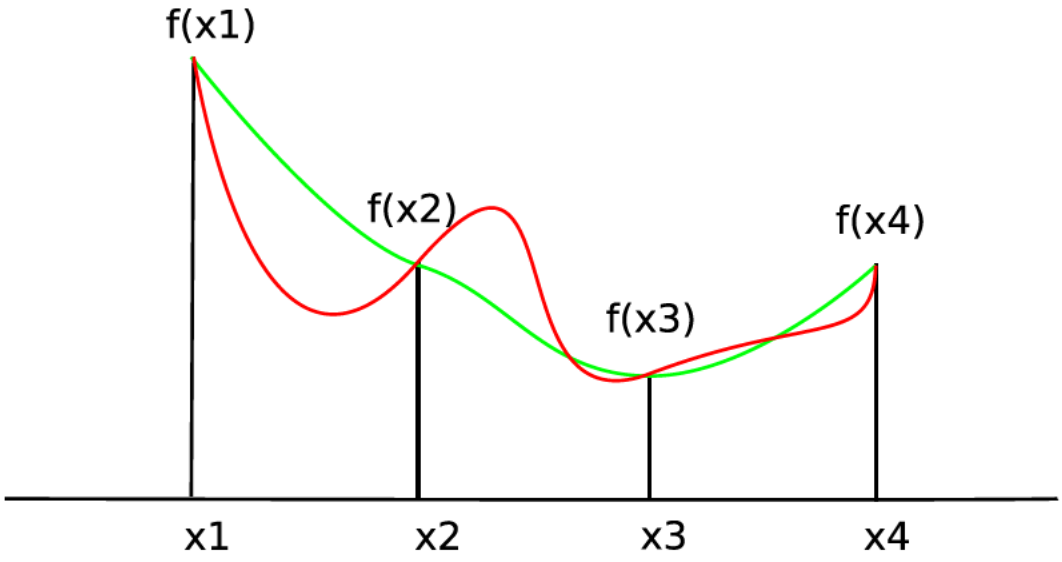

Figure 2.1: A function which is defined by discrete points is ambiguous in the domain between the points.

to the function. This holds true for each point within the domain of the function space and not only for a finite selected set of points.

Finite difference schemes often make use of this pointwise formulation. The interpretation of the method (in one dimension) is often described as follows: *In the neighborhood of a point it is assumed that the function can be formulated using a truncated Taylor series which is based on the values of this point and the neighboring points, for instance three or five points. After the function is determined by these values, some differential operator is defined in the center point.* Figure 2.1 shows that different interpolating curves are possible and therefore a list of point value pairs is not sufficient in order to characterize the function.

The main problem with such a method is that the shape functions are not explicitly given and therefore basic functional evaluations can not be performed in a straight forward manner. In most cases, some interpolation method is found which is used in order to determine the function from the given point-wise values. Mostly, linear interpolation mechanisms are used to recover the function values or the shape of the function so that even differentiation or quadrature can be performed. The determination of distinct function values as a post-processing step is cumbersome and the assumptions of the recovered

function are not always compatible with the initial assumptions about the functions used for the discretization.

In analogy, the two- or three-dimensional extension of the Scharfetter Gummel scheme [3] can be seen for the discretization of the drift-diffusion equations . The Scharfetter Gummel method uses a re-formulation of the functional behavior of the carrier concentration n on an edge in a multi-dimensional space. It can be described as a specialization of the finite difference method which is characterized by a special assumption on the carrier density n related to the potential ψ.

If this method is generalized to two or more dimensions using finite volumes - as it is done in several scientific simulation environments [59] - the values of the function are only defined on the connection lines between two points and it is assumed that the function values do not vary in a direction orthogonal to the connection line. It can be easily shown that such a method does not provide a solution for the complete simulation domain but a function which is defined only on the edges [59].

Finite element schemes typically define shape functions and use these functions in order to evaluate functionals. From this point of view at each stage of the calculation it is clear which basis functions are used and how they have to be evaluated. A set of coefficients in the computer to identify the function is used within the context of a sum of functions weighted by some stored coefficients. It is therefore clear that also for post-processing steps the same basis functions are chosen and the function is defined consistently with the simulation.

In function spaces which use functions with local support the question of finding basis functions can be reduced to finding an appropriate tesselation of the simulation domain. The elements of the tesselation are used as local support for the respective basis functions. Once an appropriate tesselation is found for a given simulation domain, the function space can be defined in a straight forward manner by defining functions locally on single element of the tesselation.

In order to find an element of the tesselation covering a certain point from this given point, "point location" methods [60] using jump and walk algorithms [61] or binary trees like oct-trees and quad-trees [62, 63] have to be used. Such a method is required in order to

determine which basis functions have to be evaluated in order to determine the function value of a certain point within the simulation domain.

2.2.2 Spatial Resolution of a Simulation Domain

In order to find simple basis functions of a simulation domain, the simulation domain is tessellated into elements of a simple geometrical archetype [31], for instance into triangles, where single shape functions can be defined. In this section the tesselation of the simulation domain is considered, while the next section shows the construction of the basis functions on the tesselation.

The process of tesselation of a continuous domain, which is usually a partial set of $\mathcal{R}^2$ or $\mathcal{R}^3$ with a finite volume is usually performed by methods which are referred to as meshing or gridding. The main aim of these methods is to tessellate the given simulation domain into a number of subdomains. In most cases, the subdomains of the tesselation are simplices [64].

Such methods are also used in other fields of computer science such as visualization or computer games. In both fields, the main focus is put on the optimization of the visual appearance. In many cases only the surfaces of the respective objects are required.

It has to be stated that this field suffers from many unsolved problems and the tesselation of some simple domains as well as local refinement [65] turns out to be difficult and impossible in some cases. However, these problems mainly occur in three-dimensional simulation domains, whereas for two-dimensional domains the tesselation is rather straight forward.

2.2.3 Shape Functions and Basis Functions

In scientific computing the main aim of the tesselation is to provide a number of pairwise disjoint sets. Shape functions are transformed to single archetypical elements of the tesselation and basis functions for the function space are obtained. Each of these basis functions is non-zero on the given set (local support) whereas it is zero otherwise. The shape of these functions is typically the same for all sets of the tesselation.

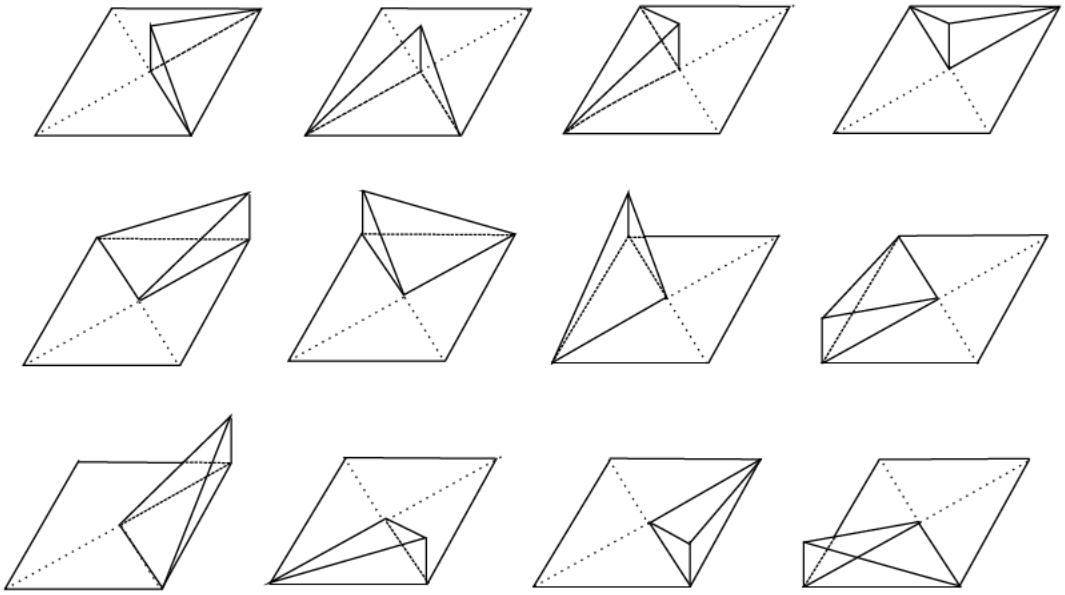

Figure 2.2: Shape of the base functions of $\mathcal{F}$.

Applying this method on all elements of the tesselation, a set of basis functions can be found based on the underlying tesselation of the simulation domain. In the following the construction of base functions is performed in detail for linearly bounded triangular elements. It is assumed that the domain is divided into subdomains of the same geometrical archetype, namely triangles. In this case it is possible to use standard functions which are based on the geometrical archetype of the elements. The single elements are transformed into the the standard element in order to evaluate the function value. For a triangle $\mathcal{I}$ a sensible choice are local linear shape functions f^{l} as basis functions is (Fig. 2.2):

$$\mathcal{I} : 0 \leq x \leq 1,\ 0 \leq y \leq 1 - x. \tag{2.7}$$

$$f_1^{\mathrm{l}}(x, y) = x \tag{2.8}$$

$$f_2^{\mathrm{l}}(x, y) = y \tag{2.9}$$

$$f_3^{\mathrm{l}}(x, y) = 1 - x - y \tag{2.10}$$

It can be easily seen, that each function yields unity in one corner point and zero in all other corner points. This implies that the weighting coefficients of the single functions have the value of the function when evaluated the respective corner point.

These functions are applied to the standard triangle and this triangle is back-transformed in its original position. Of course, such a transformation can be applied on many different archetypes of elements and also the local shape functions can be altered. From this point of view the shape of these functions f_i^{l} can be chosen arbitrarily.

For the triangle example, the three shape functions are transformed to each triangular element. One obtains a function space with linear basis functions defined on local elements of the tesselation. However, it has to be mentioned that for such a function space even very essential properties such as continuity are not given.

In order to guarantee continuity it is necessary to use a subspace of the original function space defined by the local shape functions. The main aim is to search for a subspace in which the base function and therewith all functions are continuous. To form such a function space, some weighting coefficients have to be coupled in a predefined way. In this case it turns out that the collocation of functions and points as described above is convenient, because in the corner points, where several triangles intersect, the function values have to be identical in all triangles. For this reason the function space is restricted so that each shape function which is collocated with the same point has the same weighting factor. Of course, the coupling of the weighting factors restricts the possible functions of the function space and forms a subspace $\mathcal{F}'$ of the initial space $\mathcal{F}$. The basis functions f' of the newly defined subspace $\mathcal{F}'$ of the original base space can be written in the following manner, where the points of the intersection of the triangles are denoted as $\mathcal{Q}_i$.

$$f_i'(\mathcal{P}) = \sum_{j:f_j(\mathcal{Q}_j)\neq 0} \frac{f_j(\mathcal{P})}{f_j(\mathcal{Q}_j)} \tag{2.11}$$

If the values of the shape functions in the collocation points are identically unity, the definition of $\mathcal{F}'$ can be simplified to (see Fig. 2.3)

$$f_i'(\mathcal{P}) = \sum_{j:f_j(\mathcal{Q}_i)\neq 0} f_j(\mathcal{P}) \tag{2.12}$$

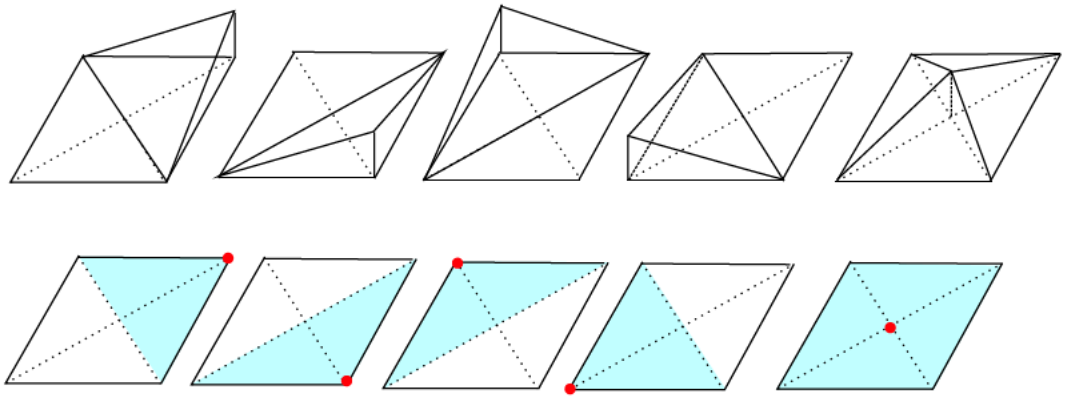

Figure 2.3: Local support and shape of the base functions of $\mathcal{F}'$.

The resulting functions f_i' have a different local support than the base functions of $\mathcal{F}$. These functions are defined within a neighborhood of the respective intersection point as can be seen in Fig. 2.3

In some cases it is favorable to use more shape functions on a base element. An example could be the use of third order shape functions which form the function space $\mathcal{G}$. Such a function space can be based on 10 basis functions, where three are collocated with the corner points of the triangle, 6 are collocated with edge points, and one is collocated with a point the interior of the triangle.

It can be seen that the function space $\mathcal{F}$ is a subspace of the function space $\mathcal{G}$. In analogy to the space $\mathcal{F}$, the property of continuity is not given in this case, because each function is based on a single triangle. In this case, however, the fitting of the functions is not restricted to a single point. While it is clear that continuity is given on the edge between two triangles, where the two end points of the edge are fit to the same value, this does not hold true for higher-order polynomial shape functions. For this reason, it is necessary to guarantee that continuity is preserved on the edge. Moreover, it can be stated that if more shape functions are available more restrictions on continuity as well as smoothness can be imposed by reducing the number of base functions.

Again, the collocation of functions and points on the triangle implies an easy fitting of neighboring triangles. In this case the collocation fitting is not only performed on the corner points of the triangle but also on two points of each edge. From this point of view it is clear that even though the use of collocation of points and shape functions is not

coercive, other local shape functions imply further equations in the final equation system and, therefore, an unnecessary use of memory and computation time.

In analogy to the measures which have been taken to derive the global shape functions of the local linear shape functions the third order local shape functions yield the global shape functions g'_i of the function space $\mathcal{G}'$ by

$$g'_i(\mathcal{P}) = \sum_{j:g_j(\mathcal{Q}_i)\neq 0} g_j(\mathcal{P}) \; . \tag{2.13}$$

It has to be mentioned that in this case the collocation points $\mathcal{Q}_j$ are the corner points, points on the edges and points in the interior of the triangles. The functions which are collocated with the interior of the triangle have their local support only on the respective triangle. The functions which are collocated with an edge point of a triangle have their local support on both triangles which are supersets of the common edge. As already mentioned, the functions which are collocated with the corner points of the triangle have their local support on all triangles which cover the respective point.

It should be noted that the method which has been shown for triangles can also be used on many other archetypes such as tetrahedra, squares, prisms, and pyramides as long as a transformation into a standard element and local element functions are given.

2.3 Topological Structures

In the last section it was shown that different collocation points result in different local support of basis functions of a function space $\mathcal{F}'$ which are constructed via tesselation in archetypical elements and a subsequent attaching of local shape functions.

The question arises if such a behavior can be specified independently from the geometrical properties, because relying on geometrical features is inefficient and also inaccurate. For instance, it is not possible to determine, if a point is on an edge or within a triangle, by methods of floating point comparison. It is inefficient to search all triangles which are within the neighborhood of a point or an edge, because a list or a tree has to be searched and it has to be checked, if the respective elements are incident.

The rule for the collocation of functions and their local support is briefly reviewed in order to show the requirements on the underlying topological structure: *Functions which are collocated with the interior of triangle have their local support only on the respective triangle, functions which are collocated with an edge point of a triangle have their local support on both triangles which are supersets of the common edge, and functions which are collocated with the corner points of the triangle have their local support on all triangles which cover the respective point.*

The properties which are required in all of these rules are of pure topological nature. This means, that only properties of sets such as unions, intersections, subsets, and supersets are relevant, whereas the geometrical properties such as coordinates and distances are irrelevant for the execution of this rule.

In the following a structure is introduced which covers all the topological properties of the initially described geometrical structure without describing its geometrical properties comprising coordinates, distances, and angles. A method is briefly introduced which provides proper means for handling the topological operations, e.g. unions, intersections and operations for finding the local neighborhood of a given element. Furthermore an association of basis functions basis on the topological space is given.

2.3.1 Finite Cell Complexes

For the definition of the underlying topological structure the definition of a topological space is used [64]. A topological space consists of a base set $\mathcal{D}$ as well as a topology $\mathcal{T}$ which is a set of subsets of $\mathcal{D}$. The topology contains the empty set $\emptyset$ and the base set $\mathcal{D}$. Furthermore, the intersection of a finite number of elements of the topology as well as the union of an arbitrary number of elements of the topology is contained in the topology. The definition can be formalized in the following manner [64] and is called the *open set definition of a topological space*:

$$\emptyset \in \mathcal{T} \tag{2.14}$$

$$\mathcal{D} \in \mathcal{T} \tag{2.15}$$

$$\forall_{i=1}^{N} t_i \in \mathcal{T} \Rightarrow \bigcap_{i=1}^{N} t_i \in \mathcal{T} \tag{2.16}$$

$$\forall_{i=1}^{\infty} t_i \in \mathcal{T} \Rightarrow \bigcup_{i=1}^{\infty} t_i \in \mathcal{T} \tag{2.17}$$

First, it has to be stated that the number of possible sets within the topological space has to be finite due to the limitations of the computer when associatively storing coefficients on the elements. Secondly, in most cases the use of sets is cumbersome, because one is actually interested in single elements of the tesselation, whereas sets containing different unconnected elements of the tesselation are rather seldomly used. In analogy with vector spaces, one can find a basis of elements from which the topological space can be composed using union operations.

A set of subsets of a set $\mathcal{H}$ on which a topological space $\mathcal{T}$ is formed is called a basis, if each set contained in the topology can be written as

$$\forall t \in \mathcal{T} \Rightarrow t = \bigcup_{i=1}^{N} h_i \wedge \forall i h_i \in \mathcal{H} \; . \tag{2.18}$$

However, each set which is an element of the tesselation is the result of an intersection of (a finite number of) such elements. If the underlying topological space is derived from a finite cell complex which usually results from a tesselation, the basis of the respective space contains all open cells, edges, faces, and vertices (or points).

If the information about the cells and their mutual intersection and union operations is available, the information of the elements of lower dimension can be obtained by intersection of the (closed) cells. The set of the closed cells is said to form a subbasis, namely a set from which all elements of the topological space $\mathcal{T}$ can be obtained by a union of arbitrarily (but finitely) many sets of the subbasis $\mathcal{S}$.

$$\forall t \in \mathcal{T} \Rightarrow t = \bigcup_i \bigcap_j s_{i,j} \wedge \forall i, j s_{i,j} \in \mathcal{S} \tag{2.19}$$

The main advantage of such a subbasis is that there is only the restriction that $\cup_{i=0}^{N} s_i = \mathbf{D}$ for the subbasis $\mathcal{S}$. (when using a finite number of elements).

In order to specify the construction of the topological space, the notion of a CW complex is introduced [64]. A CW complex is a structure which additionally fulfills the so-called weak-topology property. This means, that the boundary of each element of dimension n is formed by one or more elements of the dimension $n-1$. As a consequence, an n-dimensional complex consists of at least one element for each dimension $0 \ldots n$.

The second property which has to be fulfilled by a CW complex, namely the closure-finiteness, is trivially fulfilled by finite cell complexes which are directly implied by the use of a computer.

2.3.2 Incidence and Traversal

In connection with the construction of a function space on a tesselation, it has to be determined, which cells cover an edge or a vertex or which vertices are covered by an edge. In general, it is not necessary to discriminate which of the sets is the subset and which is the superset but one only defines the property of incidence $\sim$ between two elements of which one is a subset of the other.

$$\mathbf{a} \sim \mathbf{c} \Leftrightarrow \mathbf{a} \subset \mathbf{c} \vee \mathbf{c} \supset \mathbf{a} \tag{2.20}$$

Once the property of incidence is defined, one aims to find all elements which are incident with an initial element. Such a formulation is obtained, if the local support of a basis function is required. As an example, the local support of a shape function which is collocated with an edge point, is the set of all cells which are incident with the edge on which the point is located. Therefore, the following definition of such sets of incident elements turns out to be fruitful.

$$\mathrm{I}_{\sim,k}(\mathbf{a}) = \{\mathbf{x} : \mathbf{c} \sim \mathbf{a} \wedge \dim \mathbf{c} = k\} \tag{2.21}$$

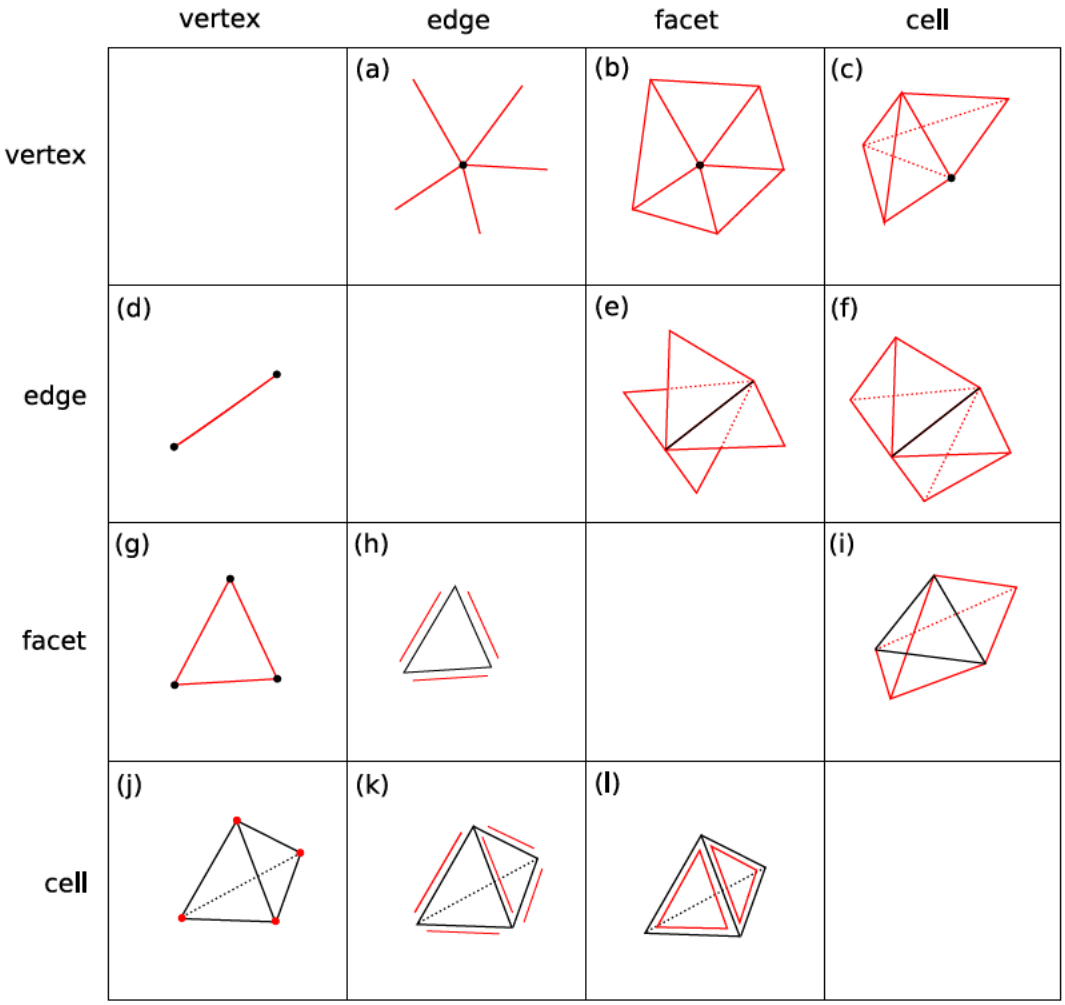

Figure 2.4: Methods of incidence traversal within a three-dimensional cell complex consisting of tetrahedra.

Traversal is defined within a data structural context, because the members of the set of incident sets $\mathrm{I}_{\mathcal{A}_k,\sim}(\mathbf{a})$ have to be iterated or traversed, i.e. the set is examined element by element. Figure 2.4 shows all possible methods of incidence traversal for a three-dimensional cell complex consisting of tetrahedra.

In the following the traversal functions are given names consisting of two capital letters where the first letter denotes the basis element (V, E, F, C) and the second letter denotes the traversed element. For instance, the function VC stands for a function which returns all cells incident to the vertex passed. Additionally, traversal functions may have $\mathcal{C}$ and $\mathcal{B}$ as first letter, where $\mathcal{C}$ denotes the cell complex and $\mathcal{B}$ denotes the union of boundary

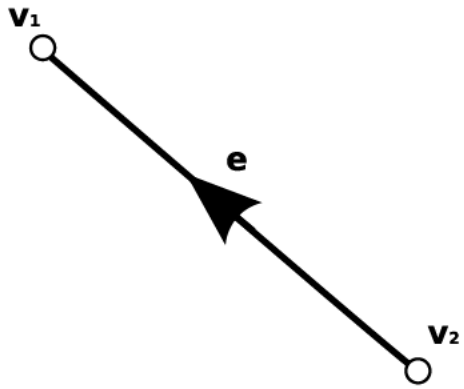

Figure 2.5: An edge and its two incident vertices can be used for the evaluation of the orientation function $\mathcal{O}$.

facets of the respective cell complex. The following identities can be established between the incidence traversal function $\mathrm{I}_{\sim}$ and the specialized incidence traversal functions:

$$VC(\mathbf{v}) = \mathrm{I}_{\sim,N}(\mathbf{v}) \tag{2.22}$$

$$CV(\mathbf{c}) = \mathrm{I}_{\sim,0}(\mathbf{c}) \tag{2.23}$$

$$CV(\mathbf{c}) = \{\mathbf{v} \in \mathcal{C} : \dim(\mathbf{v}) = 0\} \tag{2.24}$$

2.3.3 Orientation

Another feature required by many applications is orientation. Orientation can be defined as a binary function with two arguments $\mathbf{a}$ and $\mathbf{c}$ which are elements of the cell complex and have the dimension n and $n+1$. The binary function returns whether the elements are oriented consistently or not. Usually the values $+1$ and -1 are used for the result of the orientation function $\mathcal{O}$.

For instance, an edge can be oriented in a manner that an oriented path is given which begins on one boundary vertex $\mathbf{v}_1$ and ends in the other boundary vertex $\mathbf{v}_2$. Figure 2.5 shows an edge with an internal orientation and two boundary vertices of the edge. The orientation function is passed the edge and the source vertex and yields -1. In contrast, when passed the edge and its sink vertex, the orientation function yields $+1$.

$$\mathcal{O}(\mathbf{e}, \mathbf{v}_1) = 1 \tag{2.25}$$

$$\mathcal{O}(\mathbf{e}, \mathbf{v}_2) = -1 \tag{2.26}$$

This orientation function can be used in any simulation which makes use of a graph comprising oriented edges. It is important that it can be determined in which direction edge-related quantities, such as voltages in electronic circuits, are measured. It can be observed that the actual direction itself usually does not play any role, i.e. that the methods are independent from the orientation of the edges. However, once an edge orientation is chosen for all edges, the results are only valid with respect to this orientation. Other edge orientations result in another vector of solutions, whereas the obtained result remains unchanged. The orientation function is also available for elements of higher dimension as far as the dimension of two passed elements differs by one.

2.4 Topological Mappings of Shape Functions

In this section the function space introducted in the former sections is founded on the definition of the cell complex $\mathcal{C}$. Once the cell complex is established and available within the framework of the computer, weighting coefficients can be stored in association with the underlying cell complex and a function (which is an element of a predefined function space) is established.

Again, it has to be stated that such an interpretation of a function stored in a computer completely differs from the function by point interpretation, which is used in most methods based on finite differences [57] or finite volumes [58].

A basic data structural requirement for the specification of the function space based on a cell complex is that data can be associated with single elements of the cell complex. In general, one or more mappings between elements of the basis $\mathcal{H}$ of the cell complex $\mathcal{C}$ and some numeric data are used. Such a function might be defined as follows:

$$\forall \mathbf{e} \in \mathcal{H} : f(\mathbf{e}) \text{ is defined} \tag{2.27}$$

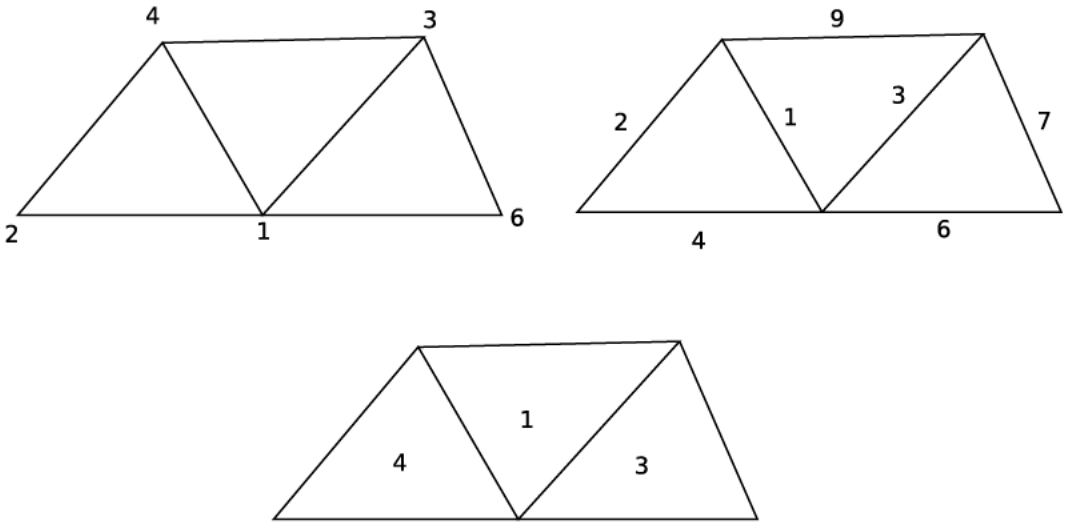

Figure 2.6: Quantities defined on vertices, edges and cells.

A function which assigns an element of the cell complex a numerical value is called a **quantity**. It is also possible that such a function is only defined on a partial set of the cell complex. The domain of definition has to be given explicitly in order not to obtain invalid function values (see Fig. 2.4).

$$\forall \mathbf{e} \in \mathcal{K} \wedge \mathcal{K} \subset \mathcal{H} : f(\mathbf{e}) \text{ is defined} \tag{2.28}$$

2.4.1 Geometrical Properties of the Elements

At this point the topological representation of the cell complex is added to the geometrical representation. Whereas the topological information contains which elements are subsets or supersets of other functions, the geometrical properties represent the actual position of the single element.

A typical geometric property is, for instance, the coordinate information of a point or the information, whether a point is contained within a cell. In order to obtain a homeomorphic mapping from the cell complex into the domain of the function, which is usually a partial space of $\mathcal{R}^2$ or $\mathcal{R}^3$ or in some cases some (preferably differentiable) manifold, it is required that each element of the tesselation can be defined via a continuous representation in the domain.

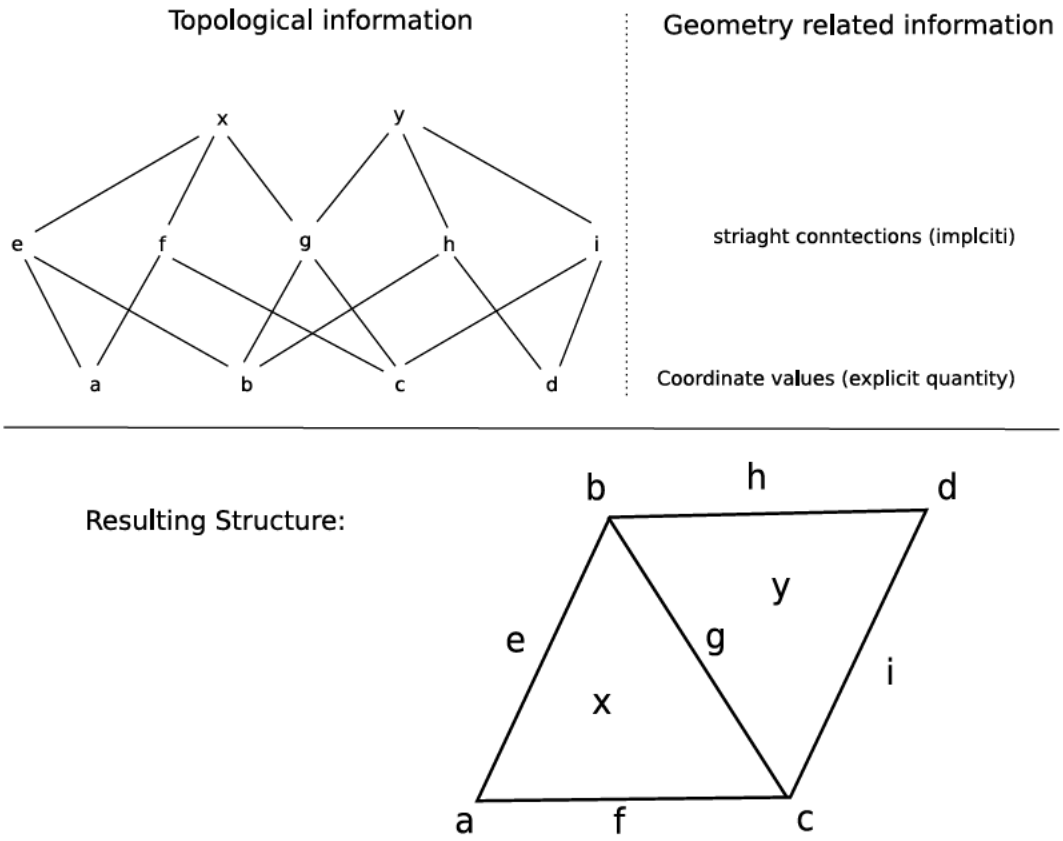

Figure 2.7: Geometrical and topological information stored separately.

As an example, all connection lines between points can be assumed to be linear and all surfaces of an element can be assumed to be plain. In this case it is only necessary to provide the coordinate information of the points in order to explicitly determine the shape of the elements. For instance, a triangle can be defined by giving all its points. Fig. 2.7 shows the separation of topological and geometrical information when using straight edges. The connection line between two given points does not necessarily have to be straight. Any continuous line could be given, which perceives the continuous (or differentiable) mapping from the cell complex to the simulation domain. The same holds true for the specification of surfaces.

It can be arranged that all data related to the geometrical representation of a certain element of the tesselation can be associated with the respective element. As an example, coordinate values of a point are associated to the respective vertex, and curvature information of the edges, such as curvature coefficients, can be associated with the topological edge structures of the cell complex. The main advantage of this method is that the infor-

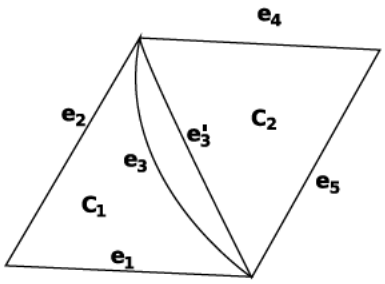

Figure 2.8: Cell related definition of the boundary curvature may lead to a non-unique definition of the bounding surface.

mation is stored in a consistent manner, because for the evaluation of geometric data, e.g. the measuring of the volume of each of the two two neighboring cells, it is possible to use the same curvature information of the intersecting surface. In contrast, the association of the surface curvature with the cells possibly leads to a non-unique definition of the bounding surface (see Fig 2.8).

In analogy to the shape functions, the geometrical structure of the topologically defined cells is related to some assumptions. The more implicit assumptions are made about the geometrical properties, the less information has to be explicitly stored of the single elements and vice versa.

An example is the specification of simplicial cells (topological definition). A possible geometric definition of such a cell is e.g., a simplex which defines all edges to be straight and all surfaces to be plain. In such a case, the only data required are the coordinates of the points, because all further information of the position of higher order elements can be derived. Implicitly, the following properties are assumed:

$$\mathbf{c}(p) = \mathbf{x}_1 + (\mathbf{x}_2 - \mathbf{x}_1) \cdot p \tag{2.29}$$

In contrast, for each element of the cell, a polynomial (or any other) function can be assumed which represents the shape of the respective element when a set of parameters is given. In the following case, a triangle is used, where the geometrical shape of the edges is represented by parameterization and the parameter p is within the unity range, and the points $\mathbf{x}_1$ and $\mathbf{x}_2$ denote the point vectors of the bounding vertices and $\overrightarrow{e_x}$ and $\overrightarrow{e_y}$ denote the unit vectors of a given coordinate system.

$$\mathbf{c}(p) = \mathbf{x}_1 + (\mathbf{x}_2 - \mathbf{x}_1) \cdot p + a \cdot (p^2 - p) \cdot \overrightarrow{e_x} + b \cdot (p^2 - p) \cdot \overrightarrow{e_y} \tag{2.30}$$

The parameters a and b denote the actual shape of the edge. If both are zero, the formulation yields a straight edge between the points $\mathbf{x}_1$ and $\mathbf{x}_2$.

The parameters used in the definition are associated to the lowest-dimensional element for whose definition they are relevant. A set of all incident lower dimensional elements can be obtained via traversal.

The information regarding the position of a point is stored in the respective vertex. If a parametric (geometrical) formulation of an edge, which is incident to the vertex is required, one can obtain the required information from the underlying vertex. However, the geometrical formulation used implicitly has to guarantee, that the topological properties of the elements are preserved under the geometrical transformation.

For this reason it has to be checked, if a higher dimensional element can be based on lower dimensional elements. If, for instance, two points are congruent, the construction of an edge based on these points is obviously not possible. This also implies that the curvature coefficients of curved edges of a triangle must not be too large, in order not to degenerate the triangle. In such a case there is no proper mapping between the topological elements and their geometrical representation.

However, if the information of vertices is given explicitly and the necessary geometrical information to construct a higher-dimensional element from its incident lower dimensional boundary is stored in this element, a consistent geometrical definition of all cells can be given. The information to construct higher-dimensional elements from lower dimensional elements can therefore either be given globally by using straight bounded elements only or by locally defining curvature coefficients.

2.4.2 Association of Functions and Topological Elements

Once the simulation domain is tessellated, the topological structure of a finite cell complex is established, and a geometrical meaning is added to the cell complex, it has to be clarified, if such a structure is useful for the representation of functions. It is considered that all these elements can be handled in the computer and various values can be stored on

different elements of the finite cell complex. In the following a method for the construction of a function space based on an existing cell complex is given.

First, the function is defined by choosing shape functions and what is further defined by the choice of weighting coefficients. It is clear that a tradeoff has to be found, because a too small number of shape functions leads to a bad representation of the solution function within the chosen function space and accordingly to a discretization error. Choosing too many shape functions makes the calculations consume too many resources and too much time.

A typical tradeoff is the choice of linear shape functions. The values which are obtained on the common points of the complex are defined explicitly (by value) whereas the shape of the functions is defined implicitly as linear on the single geometrical elements.

In the following it is assumed that each local shape function is transformed onto each cell which gives $N = l \cdot c$ basis functions of a function space, where l denotes the number of local shape functions and c denotes the number of cells in the cell complex. Each basis function returns the value of the transformed local shape function within its cell of definition whereas it returns zero otherwise. Using this method, one obtains a function space $\mathcal{F}$ spaned by shape functions which are defined on the simulation domain.

In order to exactly address one of the basis functions it is necessary to specify the cell where the basis function is defined. Moreover, the local shape function has to be given in order to uniquely specify a basis function. At this point it is convenient not to number the shape functions but to use the elements covering the collocation points of the basis function, which are called collocation elements and named $\mathbf{e}$.

As an example, the function space $\mathcal{F}$ of cell-wise linear basis functions is considered. Each basis function of the function space $\mathcal{F}$ can be specified by giving a cell as well as the collocation element. Each basis function f of the function space $\mathcal{F}$ can be written as $f(\mathbf{c}, \mathbf{e})$.

Each shape function is directly assigned a weighting function that is associated with the same topological element as the respective shape function. For the function spaces with higher-order polynomial shape functions, the association of functions to topological elements can be performed in analogy. However, it has to be considered that if on single

topological elements more shape functions f' are collocated, quantity vectors have to be used instead of scalar quantities.

In the following the question is discussed how the value of a function given as shown in the former section can be evaluated at a certain position. For storing a function within a computer, the following definitions, assumptions, and stored data have to be available as explained in the former sections.

- Topological cell complex
- Geometrical base shapes (implicit)
- Geometrical coefficients (explicit)
- Shape functions (implicit)
- Weighting coefficients (explicit)

Each function given as a linear combination of basis functions of the a function space $\mathbf{F}$ can be written in the following manner.

$$f(\mathbf{x}) = \sum_{\mathbf{c}\in\mathcal{C}} \sum_{\mathbf{e}\in\mathbf{c}} f(\mathbf{e},\mathbf{c})(\mathbf{x}) w(\mathbf{e}) \tag{2.31}$$

When carrying out the summation by iterating all cells $\mathbf{c}$ and all incident collocation elements $\mathbf{e}$ almost all elements are identically zero whereas only functions of the cell which holds the given point yield non-zero values.

A simplification can be obtained by determining the cell that holds the argument point $\mathbf{x}$. For this purpose several point search or point location algorithms [61] can be used. Both topological as well as geometrical properties have to be considered in order to find the cell $\mathbf{c}$ which holds the given point. After the respective element has been found, the function has to be determined.

$$f(\mathbf{x}) = \sum_{\mathbf{e}\in\mathbf{c_x}} f(\mathbf{e},\mathbf{c}_x)(\mathbf{x}) w(\mathbf{e},\mathbf{c}_x) \tag{2.32}$$

where $\mathbf{c}_x$ is the cell that holds $\mathbf{x}$. The summation is carried out for all collocation elements. The basis function $f(\mathbf{e}, \mathbf{c}_x)$ is evaluated with the given point $\mathbf{x}$ as argument and is weighted with the function weight $w(\mathbf{e}, \mathbf{c}_x)$.

Alternatively, the weight function can be used independently of the cell $\mathbf{c}$ which restricts the function space and couples the weights that are associated with the same collocation element $\mathbf{e}$. It can be seen easily that the resulting function space is identical to the function space $\mathcal{F}$.

Finally, it can be stated that the definition of a function, comprises information about the weighting coefficients and the shape of the base functions. Information about the weighting coefficients requires memory for each single collocation element.

Information about the basis functions comprises the geometrical shape of the elements, the shape functions locally defined on the elements as well as the coefficients for the geometrical representation of the elements such as coordinates and curvature information.

If the number of collocation elements is increased by h-refinement [5], for instance doubled, the required memory for the weighting coefficients as well as the memory for storing the geometrical coefficients is doubled. Each newly added collocation element is assigned a weighting coefficient and, if necessary, geometrical information.

Adding a new function on the same topological structure (p-refinement) the memory required for weighting coefficients is increased by the number of elements, whereas memory usage for the geometrical coefficients remains unchanged.

2.5 Formulation of Discrete Problems

In the former section the topological base operations were defined in order to retrieve elements which are incident with a given base element. As an example, all vertices incident with a given face can be determined. A second feature resulting from topological considerations is defined in this section, namely orientation. For each two elements of dimension n and $n-1$, an orientation function is defined that returns if these elements are consistently oriented.

Furthermore the method of association of topological data and numerical values allows to store data on elements of the underlying cell complex. These numerical values can contain geometrical information such as coordinate as well as curvature information.
In order to perform operations on functions which are based on the construction shown in the former section, a framework of base operations has to be defined, with which all necessary calculations can be carried out.

2.5.1 Basis Operations

In the following, a basic calculus of operations is defined which is required to formulate a discrete problem in a functional manner: The calculation is defined exactly and can be carried out by a computer. At a first glance, such a formalism seems to be trivial due to the fact that each programming language defines basic arithmetic features. However, the handling of the underlying topological structures is not supported at all so that additional features have to be provided.
The most essential element of a description formalism is the access to a function which is defined on the cell complex. Due to data structural considerations, the domain of this function can be restricted to a certain skeleton of the underlying complex. However, this does not affect the concept of the function. In this context the definition of a quantity is employed in order to obtain the respective value for a given topological element. In the formalism such a function can be formulated as $q(\mathbf{e})$, where q and q' denote the quantities and $\mathbf{e}$ denotes the element on which the quantities are evaluated.

In the next step basic arithmetic operations are introduced. Based on the rules which hold for the underlying numerical data types, the operations are introduced in the following manner:

$$q(\mathbf{e}) + q'(\mathbf{e}) =: [q \oplus q'](\mathbf{e}) \tag{2.33}$$

$$q(\mathbf{e}) - q'(\mathbf{e}) =: [q \ominus q'](\mathbf{e}) \tag{2.34}$$

$$q(\mathbf{e}) \cdot q'(\mathbf{e}) =: [q \odot q'](\mathbf{e}) \tag{2.35}$$

$$\frac{q(\mathbf{e})}{q'(\mathbf{e})} =: [q/q'](\mathbf{e}) \tag{2.36}$$

It can be seen that the functional component and the argument can be separated so that larger expressions can be formulated using compact formulations. After the structural difference between, for instance $\oplus$ and $+$, is observed, it can be easily seen that using the same symbol, namely $+$ for both notions does not introduce any ambiguity. In order to discriminate between the first order binary operator which is applied on two real or rational numbers and the second order operation which forms a function out of two given functions, the context of the operator symbol has to be considered. If written between values, writing $q(\mathbf{e}) + q'(\mathbf{e})$ the symbol has the meaning of the well known addition operation. When applied like $[q + q'](\mathbf{e})$ the sign has the meaning of $\oplus$ as introduced in 2.33.
The same holds true for the $\ominus, \ldots$ notation which compromises the readability of the notation, while no additional information is added.

2.5.2 Constants and Second Order Functions

In order to provide the use of simple numerical constants, constant functions are introduced which return a constant value independently from the actual argument.

$$q(\mathbf{e}) + 2 = q(\mathbf{e}) + \mathbf{2}(\mathbf{e}) =: [q \oplus \mathbf{2}](\mathbf{e}) \tag{2.37}$$

It is obvious that the numerical value 2 is structurally different from the constant function $\mathbf{2}$. However, the final formalism has to be defined in a programming language and,

therefore, a discrimination of 2 and **2** would lead to an unnecessary complication of the notation. In analogy to the basis operator symbols, the structure of the given functional entities is clear and depends on the context in which it is used.

Next, function application is introduced in the following way, where the functional part is strictly separated from the argument part.

$$\sin(q(\mathbf{e})) = [\sin \circ q](\mathbf{e}) =: [\mathrm{Sin}[q]](\mathbf{e}) \tag{2.38}$$

The application of the function sin() on the evaluation term $q(\mathbf{e})$ may also be interpreted as the application of the compound function Sin$[q]()$ on the element $\mathbf{e}$. In the following the functions Sin[]() and sin() have a different meaning. Whereas sin() is the typical sine function which is applied on numerical values and results in a numerical value, the second-order function Sin[]() is a function which is applied to a functional expression. Again, it can be determined from the context, if sin() or Sin[]() is meant.

Large expressions can be formed from the base operations. These operations have in common that all quantities which are employed are implicitly evaluated on the common element of evaluation, for instance a vertex, an edge or a cell. In the following section, methods are introduced which perform operations on topologically different elements while using traversal mechanisms as described in Section 2.3.

2.5.3 Accumulation

In many cases it is important to obtain the result of a sum of all values of a certain function (or quantity), which are incident to an element $\mathbf{e}$. For the sake of explicitness, the topological element $\mathbf{v}$ denotes the traversed element and fulfills the condition $\mathbf{e} \sim \mathbf{v}$, where $\sim$ denotes the incidence relation. It is clear that a method for finding all elements of fulfilling the condition $\mathbf{e} \sim \mathbf{d}$ has to be given by the topological framework. Usually the underlying framework provides traversal for incidence and adjacency relations as well as the traversal of all elements in a cell complex.

$$\sum_{\mathbf{v}\in\mathcal{C}:\mathbf{v}\sim\mathbf{e}} q(\mathbf{v}) =: [\sum_{\sim}](\mathbf{e}) \tag{2.39}$$

It has to be stated that the functional character of the $\sum$ symbol is different. On the left hand side of (2.39) the $\sum$ symbol denotes a summation in the commonly used sense, which is passed an expression (here $q(\mathbf{v})$) as well as a summation range, namely all elements for which $\mathbf{e} \sim \mathbf{v}$ holds true. On the right hand side, the $\sum$ symbol denotes a function which is passed the base element of the traversal as well as a function which is applied to all elements to be traversed. The sum of the respective function results is returned as function value of the summation function.
In this notation there is no explicit use of the traversed element $\mathbf{v}$, whose explicit naming is not necessary, because it is clear that all evaluations of the function q are performed for which the relation $\mathbf{v} \sim \mathbf{e}$ holds true. In many cases the summation is not used on the very general incidence relation but on a restricted version which can be written in terms of the traversal function.

$$\sum_{\mathbf{v}\in EV(\mathbf{e})} q(\mathbf{v}) =: [\sum_{EV} q](\mathbf{e}) \tag{2.40}$$

The main advantage of this functional formulation is that the formulation of the expression is free from the given argument. This eases the effort of specification when coding the formulae and makes the resulting code less error prone.

2.5.4 Accumulation Methods

It is also obvious that the summation can be generalized to an accumulation function. In such a case the underlying numerical data type has to fulfill the requirements of a commutative monoid [66] in order to retrieve sensible results, because the sum requires the existence of a neutral element (in the case of the summation over an empty sum) as well as a binary operation (the accumulation operation). Additionally it has to be noted that the set of elements on which the summation is based does not imply any natural

order. For this reason the binary operation has to yield identical results independently from the order of the elements. In order to fulfill this requirement, it is sufficient that the operation is associative and commutative.

Under these circumstances, the following accumulation operations can be formed. The set, the binary operation, and the neutral element are given as well.

- Sum, $\sum$ $(\mathcal{R}, +, 0)$
- Product Π, $(\mathcal{R}, \cdot, 1)$
- Exist $\exists$, $(\{0, 1\}, \vee, 0)$
- All $\forall$, $(\{0, 1\}, \wedge, 1)$
- Union $\bigcup$, $(\mathcal{T}, \cup, \emptyset)$
- Intersection $\bigcap$, $(\mathcal{T}, \cap, X)$
- Maximum Max, $(\mathcal{T}, \max(\cdot, \cdot), -\infty)$
- Minimum Min, $(\mathcal{T}, \min(\cdot, \cdot), +\infty)$

The union and the intersection operation show that the operations are not necessarily numerical. Indeed, a structure $\mathcal{T}$ which is a topology on a set $\mathcal{D}$ fulfills the requirements of a semigroup. It is also possible to characterize the minimum as well as a maximum of a function via this mechanism. It can be seen easily that the max function, which returns the maximum of two given arguments is commutative as well as associative. Furthermore an arbitrary neutral element is given so that $\max(-\infty, x) = x$ and due to the commutativity $\max(x, -\infty) = x$. Analogously, the minimum function can be introduced, where the neutral element is $+\infty$.

When traversal functions are used, the results are presented as sets. When using a computer, these sets are given as a certain sequence that covers additional information of the order of the elements. In order to eliminate the influence of the ordering of the

elements, only accumulation associative and commutative accumulation methods are allowable. Commutative monoids exactly provide these features and the results are (at least with exact numerics) independent of the special order.

In addition, the number of topological elements within a traversal set can be directly written as $\bigoplus$. The result of counting the elements is independent from the order of the elements.

An alternative accumulation method is to form a vector from the given elements. It is clear that such a method strongly depends on the order of the elements. In some cases, however, such a formulation can be of favor, especially, if only properties of a vector or a matrix are required, which are invariant with respect to permutations of the lines, e.g., the absolute value of the determinant. The symbol, which is used to denote this vectorization is $\bigsqcup$.

2.5.5 Multi-Argument Functions

Hitherto, the argument which is passed to a function was treated in a straight forward manner. Functions pass the argument given to the compound functions, these functions are evaluated and afterwards, an operation is performed using the result of the single evaluation of the compound functions, for instance an addition. For the summation an equivalent problem occurs. As a first example, which shall introduce the use of unnamed functions, a binary function $f(x, y)$ is given. A compound function shall be written which yields the following result: $g(x, y) = f(y, x)$.

Due to the possibility of altering the argument of the quantity functions, functions are not restricted to the evaluation of one single argument. As a consequence it can also be sensible to use a binary second order function. Here, the unnamed functions u_1 and u_2 represent the following dependences:

$$u_1(x_1, \ldots) := x_1, \; u_2(x_1, x_2, \ldots) := x_2 \tag{2.41}$$

This can be generalized in the following manner:

$$u_n(x_1, \ldots, x_n, \ldots) := x_n, \tag{2.42}$$

In order to specify the required commutation function, one can write the following functional expression:

$$g(x,y) := [f(u_2,u_1)](x,y) = f(u_2(x,y),u_1(x,y)) = f(y,x) \tag{2.43}$$

In order not to complicate the already introduced one-argument formalisms, it is generally assumed that all arguments are passed to the respective function in the correct order. Quantities can therefore be evaluated with respect to one of the passed arguments. In this case, the respective second order quantity function (for the sake of explicitness, the second order function is written as Q) can be evaluated.

If the second-order quantity function Q is evaluated with respect to the first argument it can be written shortly as $Q_{\underline{1}}$ or only as Q. An evaluation with the second-order function is written by the abbreviation $Q_{\underline{2}}$. Using this notation, the following expressions hold true:

$$[Q(u_1)](v,w) = Q_{\underline{1}}(v,w) = Q(v,w) = q(v) \tag{2.44}$$

$$[Q(u_2)](v,w) = Q_{\underline{2}}(v,w) = q(w) \tag{2.45}$$

If quantities functions are evaluated within the scope of a binary function, expressions can be formed in the following manner:

$$[F(Q(u_2),Q'(u_1)](v,w) = F(Q_{\underline{2}},Q')](v,w) = F(q(w),q'(v)) \tag{2.46}$$

If higher order functions are used with unnamed functions as arguments in the natural order such as $F(u_1,u_2,\ldots)$, the notation can be replaced by writing bullets $\bullet$ instead of the unnamed functions. This abbreviation leads to the following formalisms

$$[F(u_1,u_3)](v,w,x) = [F(\bullet,u_3)](v,w,x) = f(v,x)\ , \tag{2.47}$$

$$[F(u_1,u_2)](v,w,x) = [F(\bullet,\bullet)](v,w,x)) = f(v,w)\ . \tag{2.48}$$

It can be seen that the consequent passing of arguments to a function is used as default, whereas deviations from this standard have to be specified explicitly. The following binary

function shows the determination of the number of common vertices of two simplicial cells, for instance tetrahedra. This can be necessary, whether it has to be determined, if two cells have a common edge (two common vertices), a common facet (three common vertices), or they are identical (all four vertices are identical). As an auxiliary function, the Kronecker symbol δ function is used. If two elements are identical, this function returns unity, otherwise zero. The respective second order function is called δ.

$$\mathrm{cmv}(\mathbf{v}, \mathbf{w}) := [\sum_{VC}[\sum_{VC_2} \delta(\bullet, \bullet)]](\mathbf{v}, \mathbf{w}) \tag{2.49}$$

As for all other elements of this calculus, also the summations are second order functions. Hitherto, it has been implicitly assumed, that the base element of the summation is the argument passed to the sum function. As two arguments are available, it has to be specified which argument is used. In order to treat sums as any other second order function, a function is passed to the summation function, which determines the base element of the summation from the arguments passed.

2.5.6 Accumulation in Depth

In the following section a summation is considered in which different traversed elements are required in the summand function in order to evaluate quantities. A quantity q has to be evaluated in a traversed vertex $\mathbf{v}$ whereas a quantity q' is evaluated in the base element $\mathbf{c}$ of the traversal. The non-functional description of the summation can be written as follows:

$$\sum_{\mathbf{c} \in VC(\mathbf{v})} q(\mathbf{v}) \cdot q'(\mathbf{c}) \tag{2.50}$$

Using the methods from Section 2.5.3 the expression can not be transformed into a function that can be written as unary function to be evaluated in the vertex $\mathbf{v}$. The main reason for this problem is that only the cell $\mathbf{c}$ is passed to the inner function, whereas the vertex $\mathbf{v}$ is not available and, therefore, can not be passed to the summand function.

A simple method with which the argument value of the outside argument can be preserved when changing the argument list is based on named variables. Before the summation is carried out, a named variable, in this case $\underline{v}$ is assigned the (vertex) value of the argument passed. When the summands are evaluated, the function argument is the traversed cell incident to the initial vertex. However, the named variable $\underline{v}$ can be accessed in analogy to an unnamed variable. A formulation of (2.50) using only functional expressions and named variables is

$$\bigwedge^{\underline{v}:=u_1} \sum_{VC} q(\underline{v}) \cdot q'(u_1) = \bigwedge^{\underline{v}} \sum_{VC} q_{\underline{v}} \cdot q' \tag{2.51}$$

The $\bigwedge$ operator is used in order to explicitly assign the named variable $\underline{v}$ the value that is currently provided in the function argument. In analogy to unnamed functions, named functions can be written as underlined indices when quantities are applied on them. Furthermore, the definition of the named variable as the first argument ($\underline{v} := u_1$) can be simplified, because it is generally assumed that the first argument is preserved in order not to be overwritten or discarded in the summation. Therefore, the assignment can be neglected and only the name is written. Furthermore, the convention is introduced that named variables are underlined in order to distinguish them from other terms.

Compared to the lambda function which is known from the lambda calculus [67, 68], the order of the function remains unchanged, whereas named variables obtain a certain value. An appropriate formulation can also be obtained by the lambda function, where the use of named variables can be avoided. For the sake of clarity and conciseness it turned out to be more appropriate to use this formulation.

The summation can be further simplified by collapsing the $\bigwedge \sum$ formalism to a common summation symbol. The notation can be written shorter, without loss of generality. This reasoning can of course be applied to all accumulation mechanisms.

$$\bigwedge^{\underline{v}} \sum_{VC} q_{\underline{v}} \cdot q' = \sum_{VC}^{\underline{v}} q_{\underline{v}} \cdot q' \tag{2.52}$$

Using schemes of this type, arbitrary formulae which especially contain summations with function evaluation on different topological entities can be obtained. For simple equations,

the formalisms can be specified easily and no specification overhead is required. For more complicated expressions a concise and explicit manner of specification is provided.

2.6 Examples

In the last section a calculus has been introduced which offers a large variety of different methods for the discrete specification of formulae. The following section provides examples in order to show the advantages of this calculus in comparison to the standard formulation, especially when using computers for the implementation.

The main aim of the following examples is to show the required steps. Even though some methods only use features of the calculus, they are all formulated in a discrete manner. Methods are shown which are of discrete nature and are not results of a discretization process of a continuous problem, but which are modeled in a discrete manner.

In the following examples different underlying features are combined in order to calculate the required data. In the first example only topological properties are necessary to determine the solution. In the second example the considerations rely on quantities and the topological structure. The third example shows how geometrical problems can be solved. In general the geometrical treatment relies on quantities, however, implicit information about the geometrical structure of the cells is used. The same holds true for the fourth example which uses implicit information on both, geometry of the cells and the shape of the functions.

Even though also algebraic methods can be described via this formalism, the introduction requires the use of linearized equations. An example for algebraic methods is therefore shown in Chapter 4.

2.6.1 Topological Calculations

The first calculations are only of topological nature and are not based on engineering problems at all. Only topological properties of the cell complex are used. Even though such methods are not engineering problems, many engineering problems use these methods

in order to improve the quality of the underlying data structures or ease the decision which method to choose.

Number of Neighboring Edges of a Vertex

In order to estimate the number of diagonals of a multi-diagonal system matrix for instance for an equation system, it has to be determined how many vertices are connected to a vertex via an edge. As generally assumed, an edge is bounded by exactly two vertices. The number of incident edges and the number of vertices which are connected via an edge to a given vertex is identical. For this reason it is only necessary to count the edges which are incident with a vertex. The formula expressing this fact can be written as follows:

$$N(\mathbf{e}) = \bigoplus_{\mathrm{VE}}(\mathbf{e}) \tag{2.53}$$

In order to estimate the number of matrix diagonals, the maximum of neighboring edges within the cell complex has to be determined.

$$\mathrm{diag}(\mathcal{C}) = \mathrm{Max}_{\mathcal{C}\mathrm{V}} \bigoplus_{\mathrm{VE}}(\mathcal{C}) \tag{2.54}$$

In some applications it is necessary to obtain global information of the underlying cell complex. This mostly depends on the fact whether such a complex features a hole or consists of two non-connected subregions. These cases have to be discriminated before a sensible mathematical consideration is possible. The Euler number [64] is determined in the following manner from the number of cells, edges and vertices of a cell complex. This can be easily achieved by the following operation:

$$[\bigoplus_{\mathcal{C}\mathrm{C}} - \bigoplus_{\mathcal{C}\mathrm{F}} + \bigoplus_{\mathcal{C}\mathrm{V}}](\mathcal{C}) \tag{2.55}$$

One main advantage compared to the standard formulation is that this is an explicit and unambiguous definition, which is directly executable by a computer.

2.6.2 Discrete Solution Methods

A further step of complexity is the evaluation of quantities, namely functions which are defined on elements of at least one type of topological elements of the cell complex. For the following examples the cell complex considered is of dimension one, namely a graph. In all of the considered models the modeling directly results in a discrete problem. The main reason for the use of a discrete model is that the continuous problem is cumbersome to describe and does not introduce any improvements on the accuracy of the results. Often discrete simulation models are used, if the exact physical behavior is not sufficiently known or if the physical behavior is too complicated to model.

In the following, simulation methods for electrical and mechanical circuits [69, 70], methods for supply chain networks [71, 72] are shown. Furthermore, a large variety of graph based problems can be investigated using the same topological framework, for instance neural networks [73], the simulation of telecommunication networks [74], or control networks [75].

The underlying topological structure for all these simulation methods is a directed graph [76] (Fig. 2.9). Various implementations of graph data structures are currently available, mainly coupled with simulation tools for the branches mentioned above and only few of frameworks allow a data-structure independent implementation such as the Boost Graph Library [44].

Typical applications require the summation of all incident edges of a vertex with different weighting of ingoing and outgoing edges. In some cases, for instance for electrical or mechanical circuit simulation it is required to weight ingoing edges with the factor $+1$ and outgoing edges with the factor -1 or vice versa. In other cases, for instance supply chain simulation, only ingoing edges are considered, whereas outgoing edges are neglected. In terms of the topological framework of the GSSE, the direction information of the edges can be used in formulations by the orientation function. For the application of electrical networks this formulation is quite convenient. The first Kirchhoff Law can be written as follows

$$\sum_{VE}^{\underline{v}} \mathcal{O}(\bullet, \underline{v}) I = 0 \ . \tag{2.56}$$

Many other simulation applications provide an inherent flow direction, for instance supply chain simulation. While for the simulation of electrical circuits the orientation of edges and vertices has to be defined arbitrarily (see Fig. 2.10), the orientation of edges and vertices in a supply chain has to be adapted to the special supply chain or a control network (see Fig. 2.11).

In supply chain simulation, a processing facility, a machine, a transport lane, or a human is associated with a node or vertex. For a control network, a control element such as a control path or a controller associated to a vertex. If the output of one element associated with a vertex is required as input of another element, an edge incident with both vertices is introduced.

Typically, each element associated with a vertex fulfills a typical behavior, namely it takes its input as well as its internal state and processes one or more output values. In order to specify this behavior with the means shown in the previous section, the following formulation can be introduced

$$q = \sum_{VE}^{\underline{v}} \mathcal{O}'(\bullet, \underline{v}) \sum_{EV}^{\underline{e}} \mathcal{O}'(\bullet, \underline{e}) f(q) \ , \tag{2.57}$$

where q denotes the output quantity of a vertex. The processing of the input quantity is specified by the function f . The modified orientation function $\mathcal{O}$ is defined as follows:

$$\mathcal{O}'(\mathbf{v}, \mathbf{e}) := \begin{cases} \mathcal{O}(\mathbf{v}, \mathbf{e}) = 1 & : 1 \\ \mathcal{O}(\mathbf{v}, \mathbf{e}) = -1 & : 0 \end{cases}$$

It can be seen easily that a number of output quantities is calculated and subsequently multiplied with zero. Even though this leads to a correct result, the performance of the calculation is unnecessarily worsened. For this reason a traversal method IE can be used which only considers the vertices which are located on the ingoing edges of a given vertex.

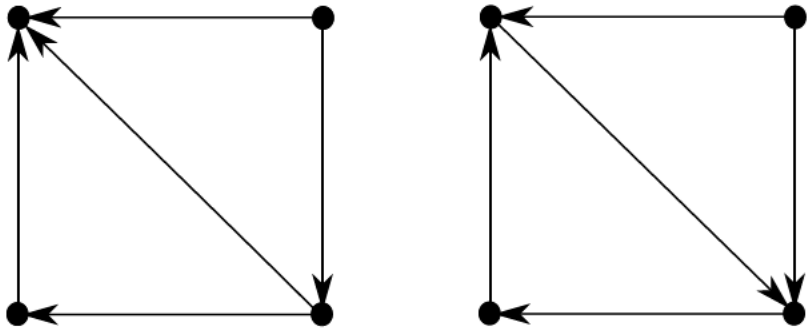

Figure 2.9: Directed graph with different orientations.

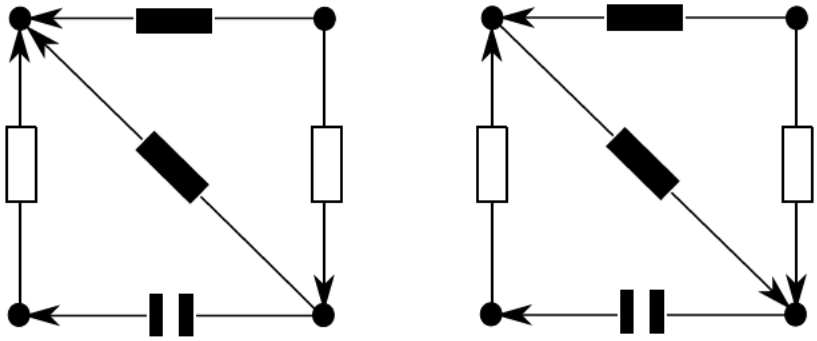

Figure 2.10: For the simulation of electrical circuits the orientation has to be given, but is not essential for the simulation.

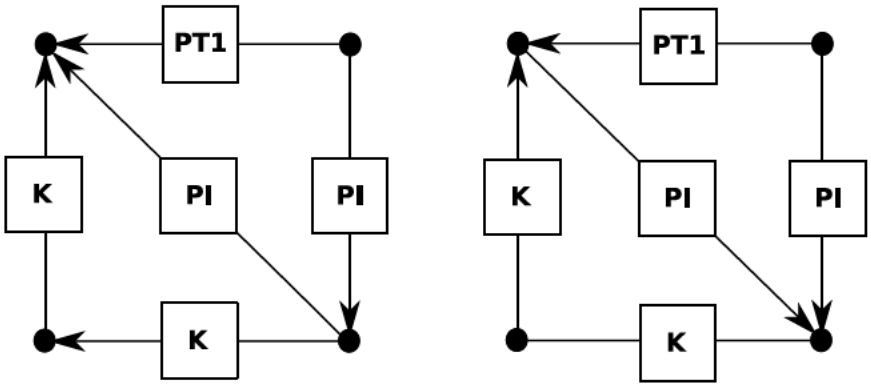

Figure 2.11: For the simulation of control networks the orientation has been explicitly defined. Changes of the direction lead to a structurally different behavior.

The traversal method IV returns the source vertex of an edge. Using this formulation (2.57) can be simplified as follows:

$$q = \sum_{IE} \sum_{IV} f(q) \; , \tag{2.58}$$

Such a method can be excellently applied to neural networks [77]. Each input is weighted with a predefined value w defined on the respective connection edge and the sum of all weighted input quantities is formed. Afterwards a threshold function T is applied to the weighted sum.

$$q = T(\sum_{IE} w \sum_{IV} q) \; , \tag{2.59}$$

It can be seen that graph based discrete models can be easily written in the specification language proposed in the previous section. Graph based simulation methods can be formulated using the traversal methods introduced in this section.

2.6.3 Geometric Examples

In some cases it is necessary to determine the volume of a cell. For instance, many formulae for distinct integrals lead to a formulation which contains the original volume. In the following case the calculation of the volume is shown for a tetrahedron. A typical formulation of the volume of a general simplex can be defined as follows.

$$\frac{\left| \det \begin{bmatrix} x_4 - x_1 & y_4 - y_1 & z_4 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \end{bmatrix} \right|}{3!} \tag{2.60}$$

For the determination of this determinant value it is necessary to obtain one definite vertex from the set of incident vertices CV. This can be easily provided by a first vertex function FCV which returns only one vertex of the cell. Furthermore, it is necessary to remove the respective vertex from the set of vertices CV. Using the $\bigsqcup$ operator, a matrix can be provided. The vector value quantity $\mathbf{x}$ contains the coordinate of the vertices.

$$\mathrm{V}(\mathbf{c}) = \bigwedge^{\underline{v}:=FCV(\bullet)} \frac{|\det[\bigsqcup_{CV\setminus\underline{v}}[\mathbf{x}_{\underline{v}} - \mathbf{x}]]|}{[\bigoplus_{CV}]!} \tag{2.61}$$

The expression $\mathbf{x}_{\underline{v}}$ evaluates the coordinate of the first of the incident vertices passed by the FCV function. The accumulation $\bigsqcup_{CV\setminus\underline{v}}$ forms a vector in which the resulting vectors of the subtractions $x_n - x_1$ are inserted.

2.6.4 Quadrature of Discretized Functions

The field of numerical quadrature can be divided into the integration of known analytical functions, and the quadrature of element-wise given functions. Both are used in different cases. The first method is commonly used for functions, which are not integrable by analytical means. The main question is to find an accurate approximation of the integral value, usually by point-wise evaluation of the function value or the derivative. An archetypical method for such an integration is the Gauß-integration method [52]. Such methods inherently exhibit a discretization error, which usually depends on the accuracy of the resolution of the tesselation. Finer resolutions lead to more accurate approximations of the integral.

In the second case the functions are defined explicitly, for instance as a member of the function space $\mathcal{F}$. In this case all integrals can be determined explicitly. The quadrature can be carried out without a discretization error caused by the interpolation of the point-wise given function. In many cases this kind of quadrature is not used for the purpose of obtaining the integral itself but for the specification of differential equations, for instance, in finite element methods.

In this section a method to calculate the integral of a certain subdomain of the simulation domain is shown. Such an integration domain is typically tessellated into several elements, on which typically the same shape functions are defined and coefficients are stored. The function can be retrieved by inserting the coefficients into the weighting coefficients of the function.

In order to determine the integral over the integration domain, it is necessary to calculate the integral on one cell, which depends on values stored either on the respective cell itself

or on subsets of the cell such as edges or vertices. The following integration is carried out for linearly bounded triangles. This method shall show the general methodology and is applicable for different shapes of elements and different shape functions, as long as the integration can be performed analytically. In other cases, methods of numerical integration have to be used for each single element. The analytical result of the integration leads to the following results

$$I(f) := \int_{\mathbf{c}} f(x)\, dV = \frac{(q(\mathrm{v}_1) + q(\mathrm{v}_2) + q(\mathrm{v}_3))}{3} \cdot \mathcal{V} = \big[\frac{\sum_{CV}[q]}{3} \cdot \mathcal{V}\big](\mathbf{c})\ , \tag{2.62}$$

where $\mathcal{V}$ denotes the volume of the cell. Combined with the formula for the value of the simplices, the formula can be generalized to simplices of arbitrary dimension.

$$I(f) := \big[\frac{\sum_{CV}[f]}{\bigoplus_{CV}} \cdot \bigwedge^{\underline{v}:=FCV(\bullet)} \frac{|\det[\bigsqcup_{CV\setminus\underline{v}}^{\underline{v}}[\mathbf{x}_{\underline{v}} - \mathbf{x}]]\ |}{[\bigoplus_{CV}]!}\big](\mathbf{c}) \tag{2.63}$$

The integration of the cell complex or a subset of the cell complex, which represents the integration domain $\mathcal{D}$ can be obtained by the following formula:

$$\int_{\mathcal{D}} f(x)\, dV = \big[\sum_{\mathcal{C}C}\big[\frac{\sum_{CV}[f]}{\bigoplus_{CV}} \cdot \bigwedge^{\underline{v}:=FCV(\bullet)} \frac{|\det[\bigsqcup_{CV\setminus\underline{v}}^{\underline{v}}[\mathbf{x}_{\underline{v}} - \mathbf{x}]]\ |}{[\bigoplus_{CV}]!}\big]\big](\mathcal{D}) \tag{2.64}$$

This formula yields the integral of a piecewise-linear function, where the subdomains, on which linear functions are defined, are simplices. This holds true for arbitrary dimensions and, therefore, represents a functional formulation of a general n-dimensional algorithm. As such, it can be directly used in a program, only syntactical changes are necessary due to the restrictions of the programming language.

Chapter 3

Differential Equations

In this chapter discretization schemes for differential equations are discussed in the context of the formalism introduced in the previous chapter. By now it was shown how functions are introduced to the computer, how functions can be evaluated, added and transformed to other bases. The main focus of the following sections is to show that all discretization schemes which are used for the discretization of differential equations can be formalized in the same manner using the formalusm presented in the previous chapter.

As will be discussed in Chapter 4, the outcome of a discretization scheme is an equation system, where the number of equations equals the dimension of the used function space. In order to fulfill this requirement an association scheme for equations and unknown variables is introduced. This association is based on the consideration that each basis function has one equation on which the coefficient of the respective function has the most influence. Such considerations are also referred to as control functions.

In the following sections the methods of formula specification of Chapter 2 are used in order to form expressions according to the respective discretization scheme. It shall be shown that typical discretization schemes can be specified using the formalism introduced. Moreover, the formalism implies a view on how the discretization scheme directly influences the underlying data structure. Each discretization scheme imposes different requirements on the underlying data structures with respect to traversal and storage of the quantities. A discussion on how the data structures can be chosen in an appropriate manner can be seen in Section 2.5.

In each of the sections the Laplace equation on a two-dimensional triangularly tessellated simulation domain is investigated and formulated.

3.1 Finite Element Schemes

Many methods employed in scientific computing are based on a finite element approach. The Galerkin method [55] for finite elements is discussed, however, other schemes can be implemented as well. The approach of using Galerkin schemes on a topologically based function spaces is shown. In the following, it is implicitly assumed that the differential operators used are linear. Due to the usual solution mechanisms comprising discretization and linearization this does not prohibit the proposed methods from being used for arbitrary problems.

3.1.1 Weak Formulation and Galerkin Schemes

The method is based on the notion of the weak formulation or weak solution, which is defined in the following manner: A function u is a weak solution of a differential equation $\mathcal{L}(u) = 0$ within the domain $\mathcal{D}$, iff for each function w the following condition holds true:

$$\int_{\mathcal{D}} \mathcal{L}(u)\, w \, dV = \langle \mathcal{L}(u), w \rangle = 0 \tag{3.1}$$

This condition can not hold true for arbitrary functions w, because the underlying function space does not necessarily provide a weak solution. Consequently, one attempts to fulfill such a condition as well as possible. For this reason, a space $\mathcal{W}$ of special weighting functions $w_1 \dots w_n$ is introduced, which is used to measure the deficiency of the numerical solution.

A widely used approach, which uses the shape functions as weighting functions is the Galerkin approach. It has been shown that such an approach has many advantages such as providing a symmetric equation system or system matrix.

The typical formulation of a differential equation using the Galerkin finite element method is written as

$$R_j := \sum_i \int_{\mathcal{D}} q_i \mathcal{L}(f_i')\, f_j' \, dV = \sum_i \langle u_i \mathcal{L}(f_i'), f_j' \rangle = 0 \,. \tag{3.2}$$

The coefficients q_i denote the weighting coefficients for the shape functions. The solution function can be written in terms of the shape functions in the following manner

$$u = \sum_i q_i f'_i \, . \tag{3.3}$$

It is assumed that the space of possible solution functions $\mathcal{F}'$ is derived from a given tesselation of the simulation domain as shown in Chapter 2. For the sake of simplicity linear shape functions are used, where single shape functions f' collocate with vertices of the tesselation of the simulation domain.

3.1.2 Re-Formulation on Topological Properties

The indices i and j from (3.2) and (3.3) refer to the system matrix or to the numbering of the degrees of freedom in the function space $\mathcal{F}'$. In this derivation, instead of indices, topological entities, in this simple case vertices, are used. Therefore, each function of the function space $\mathcal{F}'$ can be directly associated with its corresponding vertex so that one can write $f'(\mathbf{v})$ instead of f'_i, where $\mathbf{v}$ is the vertex with index i. In analogy, a function of the function space $\mathcal{F}$ can be written as restriction of a function corresponding to a vertex $\mathbf{v}$ restricted on a cell $\mathbf{c}$, namely $f(\mathbf{v}, \mathbf{c})$. It has to be assured that the vertex $\mathbf{v}$ and the cell $\mathbf{c}$ are incident in order to retrieve a valid, non-zero basis function of $\mathcal{F}$. As can be seen easily, these considerations can be used analogously for other topological entities on which quantities are stored. The substitution of the notation leads to the following formulation

$$R(\mathbf{w}, \mathcal{C}) := \big[\sum_{\underline{2}}^{\underline{v}} \langle q\mathcal{L}(f'), f'_{\underline{v}} \rangle\big](\mathbf{w}, \mathcal{C}) \tag{3.4}$$

Next, the shape functions as well as the weighting functions, both from the function space $\mathcal{F}'$ are written as functions of the function space $\mathcal{F}$. This can only be applied, if the function space is derived from a tesselation of the underlying simulation domain and yields

$$f' = \sum_{VC}^{\underline{v}} f(\bullet, \underline{v}) \tag{3.5}$$

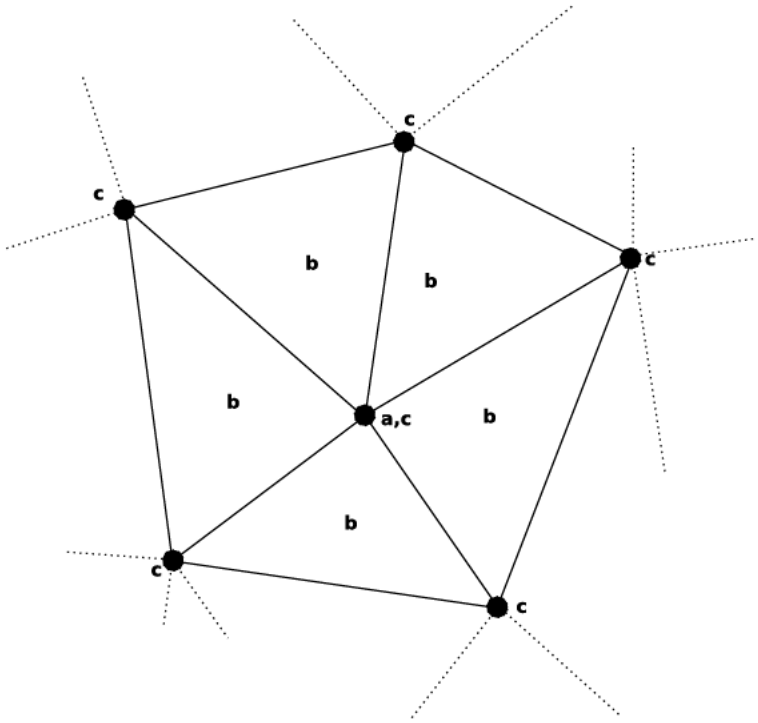

Figure 3.1: The finite element scheme applied to an unstructured cell complex.

with the resudial term R and

$$R := \sum_{CV} \langle q\mathcal{L}(\sum_{VC}^{\underline{v}} f(\bullet, \underline{v}), \sum_{VC(\mathbf{w})}^{\underline{w}} f(\bullet, \underline{w})\rangle \ . \tag{3.6}$$

As the summands of the inner sums do not depend on the summation variables of the other summation, the sums can be written in the following manner:

$$R(\mathbf{w}, \mathcal{C}) = [\sum_{CV_{\underline{2}}}^{\underline{w}:=u_1} \sum_{VC}^{\underline{v}} \sum_{VC(\underline{w})}^{\underline{c}} \langle q_{\underline{v}}\mathcal{L}(f(\underline{c}, \underline{v})), f(\bullet, \underline{w})\rangle](\mathbf{w}, \mathcal{C}) \tag{3.7}$$

Under the assumption that two functions of $\mathcal{F}$, namely $f(\mathbf{c}, \mathbf{v})$ and $f(\mathbf{d}, \mathbf{w})$ can only yield a non-zero inner product, $\mathbf{c}$ has to be identical to $\mathbf{d}$. Otherwise at least one of the functions is identically zero on the complete simulation domain. Furthermore, both of the vertices $\mathbf{v}$ and $\mathbf{w}$ have to be incident with the common cell $\mathbf{c}$, because of the definition of the function space $\mathcal{F}$. Figures 3.1 and 3.2 show the traversal mechanisms that have to be available for the implementation of finite element schemes.

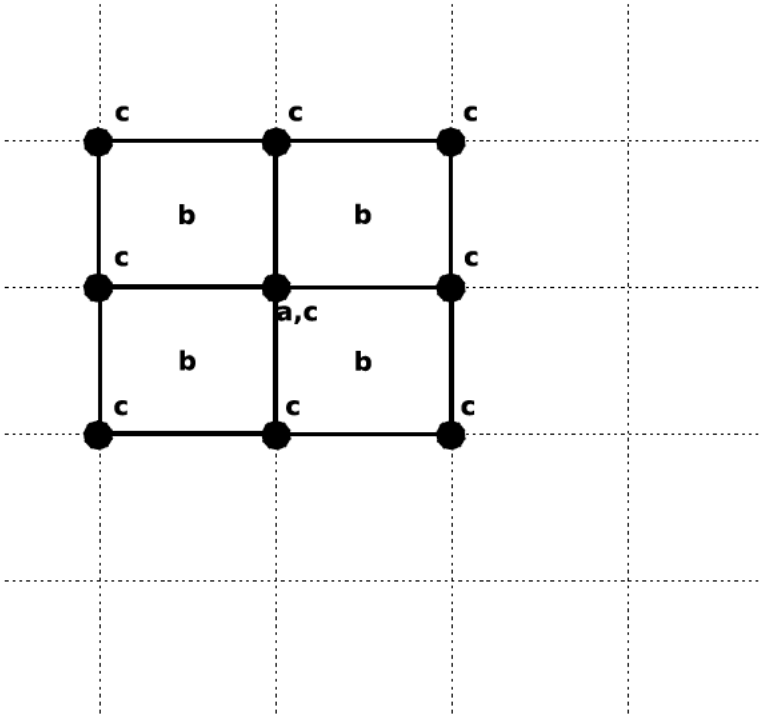

Figure 3.2: The finite element scheme applied to a structured cell complex.

$$(\mathbf{c} = \mathbf{d}) \wedge (\mathbf{w} \sim \mathbf{c}) \wedge (\mathbf{v} \sim \mathbf{c}) \tag{3.8}$$

Under the assumption of condition (3.8) the summation can be simplified to

$$R(\mathbf{w}) = [\sum_{VC}^{\underline{v}} \sum_{CV}^{\underline{c}} q \underbrace{\langle \mathcal{L}(f_{\underline{c},\underline{v}}), f_{\underline{c},\bullet} \rangle}_{K(\underline{c},\underline{v},\bullet)}] \tag{3.9}$$

The quadrature of the respective shape functions can be performed in a straight forward manner. Moreover, different differential equations, quadrature methods shape functions, weighting functions quadrature methods and even inner products can be chosen without affecting the overall algebraic structure of the equation system. The specialization to a certain set of methods can be reduced to formulating a proper coefficient function K.

3.1.3 Integration Example

In the implementation of the finite element scheme the actual differential operator is not relevant for algebraic structure of the formalism. This section shows the application of the finite element method to the Laplace equation

$$\Delta u = 0 \, . \tag{3.10}$$

For such an equation functions are required which are twice differentiable. The usual method to reduce this requirement is to apply the Green integral identity where one obtains the following expression

$$K(\mathbf{c}, \mathbf{v}, \mathbf{w}) = \int_{\mathcal{T}} \Delta f(\mathbf{v}) f(\mathbf{w}) dV = \int_{\mathcal{T}} \langle \mathrm{grad} f(\mathbf{v}), \mathrm{grad} f(\mathbf{w}) \rangle dV \, . \tag{3.11}$$

$\mathcal{T}$ denotes the local integration region. Using a geometrical transformation, simplicial elements can be transformed into unity simplices. For other elements, e.g. cuboids, it is also possible to transform the integration domain in order to simplify the calculation. The following considerations concern the standard method of simplicial (triangular) elements and linear shape functions. Methods in which the integrals can be solved analytically can be treated in the same manner.

In most cases the quadrature can be performed analytically, which implies that most of the calculations are carried out before the simulation is started. During the simulation process predetermined numbers are inserted. In many cases, e.g. more complicated differential equations, irregular shapes of the elements, or when using higher order polynomial approaches, it might be of favor to use alternative quadrature methods, mostly numerical quadrature means. In such a case the evaluation of the function K invokes e.g. a Gauß quadrature method. Moreover, it can be stated that a quadrature method for the determination of the integral function K does only require information which is associated with one of the arguments passed. In some special cases it might be desirable to use incidence traversal methods, e.g., to determine values which are associated to edges of the respective cell.

In the following considerations the integrals are not evaluated as a complete matrix but separately for each two vertices on a given triangular cell. The integral is evaluated using

the typical geometrical transformation to a standard triangular element with corner points in $(0,0)$, $(1,0)$ and $(0,1)$. The local coordinates of the transformation are denoted as η and ξ. The associated shape function for the single vertices are η, ξ and $1-\eta-\xi$. A transformation function can be found in the following way:

$$x = \xi(x_2 - x_1) + \eta(x_3 - x_1) \tag{3.12}$$

$$y = \xi(y_2 - y_1) + \eta(y_3 - y_1)\ , \tag{3.13}$$

where x_n and y_n denote the vertices of the triangle in any ordering. The derivatives of the variables in the standard range η, ξ with respect to the original variables x, y are:

$$\partial_x \xi = \frac{y_3 - y_1}{J}, \qquad \partial_y \xi = \frac{y_2 - y_1}{J}, \qquad \partial_x \eta = \frac{x_3 - x_1}{J}, \qquad \partial_y \eta = \frac{x_2 - x_1}{J}\ . \tag{3.14}$$

Here, J denotes the determinant of the transformation matrix. Together with this determinant the differentials can be transformed. The determinant is defined with respect to the cell. For the calculation the coordinate values of the vertices are required.

$$J = (x_2 - x_1)(y_3 - x_1) - (x_3 - x_1)(y_2 - y_1), \qquad dxdy = Jd\eta d\xi\ . \tag{3.15}$$

In the first case of the evaluation of the function K, the vertices $\mathbf{w}$ and $\mathbf{v}$ coincide and the point $(0,0)$ is used as point of the common vertex. The common shape function is denoted as $f(\mathbf{v}) = f(\mathbf{w}) = f(\mathrm{c}) = 1 - \xi - \eta$.

$$K(\mathbf{c}, \mathbf{w}, \mathbf{v}) = \textstyle\int_{\mathcal{T}} (\mathrm{grad} f(\mathbf{v}), \mathrm{grad} f(\mathbf{v})) dxdy =$$

$$\textstyle\int_0^1 \int_0^{1-\eta} (\partial_x f^{\mathrm{l}}(\mathbf{v}))^2 + (\partial_y f^{\mathrm{l}}(\mathbf{v}))^2 J d\eta d\xi =$$

$$\textstyle\int_0^1 \int_0^{1-\eta} (\underbrace{\partial_\xi f^{\mathrm{l}}(\mathbf{v}))\partial_x \xi}_{(y(\mathbf{v}_3) - y(\mathbf{v}))/J} + \underbrace{\partial_\eta f^{\mathrm{l}}(\mathbf{v}))\partial_x \eta}_{(y(\mathbf{v}) - yy(\mathbf{v}_2))/J})^2 + (\underbrace{\partial_\xi f^{\mathrm{l}}(\mathbf{v}))\partial_y \xi}_{(x(\mathbf{v}_2) - x(\mathbf{v}))/J} + \underbrace{\partial_\eta f^{\mathrm{l}}(\mathbf{v}))\partial_y \eta}_{(x(\mathbf{v}) - x(\mathbf{v}_3))/J})^2 J d\eta d\xi =$$

$$\textstyle\int_0^1 \int_0^{1-\eta} (y_3 - y_2)^2 + (x_3 - x_2)^2 / J d\eta d\xi = \frac{\|\mathbf{x}(\mathbf{v}_3) - \mathbf{x}(\mathbf{v}_2)\|}{2J} = \frac{l(\mathrm{edge}(\mathbf{v}_3, \mathbf{v}_2))}{2J} \tag{3.16}$$

It can be seen that the value of the integral depends on the volume of the cell as well as on the length of the edge opposing the common edge (see Fig. 3.3). For this reason a topological function $\mathrm{opp}(\mathbf{c}, \mathbf{v})$ can be introduced which returns the edge opposing a

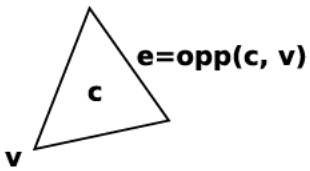

Figure 3.3: The opposing edge of a vertex within a cell.

vertex within a triangular cell. The quantities J and L are defined on the cells and edges, respectively.

$$\mathbf{w} = \mathbf{v} \Rightarrow K(\mathbf{c}, \mathbf{w}, \mathbf{v}) = \frac{L(\mathrm{opp}(\mathbf{c}, \mathbf{v}))^2}{J(\mathbf{c})} \tag{3.17}$$

For the second case it is assumed that the vertices $\mathbf{v}$ and $\mathbf{w}$ are different. In this case, the vertices $\mathbf{v}$ and $\mathbf{w}$ are located on the points $(1, 0)$ and $(0, 1)$. The shape function collocated with $\mathbf{v}$ is referred to as $f(\mathbf{v})$ and the shape function collocated with $\mathbf{w}$ is referred to as $f(\mathbf{w})$.

$$\begin{aligned} \mathbf{w} \neq V \Rightarrow K(\mathbf{c}, \mathbf{w}, \mathbf{v}) = \\ \int_{\mathcal{T}} (\mathrm{grad} f(\mathbf{v}), \mathrm{grad} f(\mathbf{w})) dx dy = \\ \int_0^1 \int_0^{1-\eta} (\partial_x f(\mathbf{v}) \partial_x f(\mathbf{w}) + \partial_y f(\mathbf{v}) \partial_y f(\mathbf{w})) J d\eta d\xi = \\ -\frac{(y(\mathbf{w}) - y_1)(y(\mathbf{w}) - y_1) + (x(\mathbf{w}) - x_1)(x(\mathbf{v}) - x_1)}{2J} \end{aligned} \tag{3.18}$$

To precisely specify this formula using the formalism presented in Chapter 2, the localities of the quantities are defined exactly, for the third vertex within the triangular cell, the following topological function can be used (see Fig. 3.4).

$$\mathrm{res}(\mathbf{c}, \mathbf{w}, \mathbf{v}, \ldots) := \mathrm{vert}(\mathbf{c}) \setminus \{\mathbf{w}, \mathbf{v}, \ldots\} , \tag{3.19}$$

where the function vert() denotes all vertices incident with a cell. The final formula yields

$$K(\mathbf{c}, \mathbf{v}, \mathbf{w}) = [\bigwedge_{\underline{z} := \mathrm{res}(\bullet, \bullet, \bullet)}^{Z} [-\frac{(y_{\underline{2}} - y_{\underline{z}})(y_{\underline{3}} - y_{\underline{z}}) + (x_{\underline{2}} - x_{\underline{z}})(x_{\underline{3}} - x_{\underline{z}})}{2J_{\underline{1}}}]](\mathbf{c}, \mathbf{v}, \mathbf{w}) \tag{3.20}$$

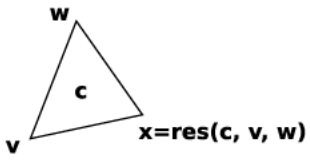

Figure 3.4: The third vertex within a triangle. $\mathbf{x} = \mathrm{res}(\mathbf{c}, \mathbf{v}, \mathbf{w})$.

These considerations can be performed for other differential operations, element shapes, and shape functions, as long as the integrals are evaluable analytically. As the necessary quantities and geometrical properties are always associated with topological elements incident to the common cell, further topological functions such as the opp() function may become relevant for the calculations. One restriction which is crucial for the evaluation of the integral term K is that only quantities are required which are associated with elements that are incident with the cell C or which are global.

3.1.4 Conclusion

One major advantage of the formalism according to Chapter 2 is that the requirements on the underlying structures as well as the complexity can be seen easily. While the initial formulation using the indices i and j does not necessarily reveal the actual nature of the traversal of the underlying topological complex, the final formulation gives a precise definition of the topological traversal functions required for the implementation of the traversal.

In order to deal with higher-order schemes, it is necessary to use functions which traverse all cells which are incident to an element and traversal schemes which traverse all elements which are incident to one cell. Furthermore, it has to be considered that on one element two values of one quantity can be located, if more collocation points are in the interior of this element.

3.2 Finite Volume Schemes

Finite volume methods are often used for the simulation of systems which inherently involve the property of continuity, e.g. the conservation of fluxes or currents. Problems of computational fluid dynamics as well as semiconductor device simulation [33, 58] are typically implemented in finite volume schemes.

3.2.1 Secondary Graph Method - Gauß Integral Theorem

As a first step a function space $\mathcal{F}'$ is defined on the tessellated simulation domain. Typically, linear shape functions are assumed, however, it only has to be assured that all shape functions are continuous [58]. For the secondary graph simulation method only vertices and edges of the cell complex are used. Furthermore, the cell complex is associated its dual graph comprising the cells of the Voronoi tesselation as well as the boundary surfaces of these boxes. Dual elements of vertices are cells which cover the set of points closer to the respective vertex than to any other vertex [78]. Each edge is assigned a part of the boundary surface of the dual cell. If the original tesselation fulfills the Delauney property, a dual graph can be constructed in a unique manner and the cells are disjoint and bounded by the surfaces. If the Delauney property holds true for a cell complex, each vertex $\mathbf{v}$ is assigned a dual cell $\mathbf{c} = \text{dual}(\mathbf{v})$. Consequently, an edge incident with the vertex $\mathbf{v}$ bounds the cell $\mathbf{c}$.

Each local shape function is defined with respect to a vertex and is non-zero on all cells incident with this vertex. In the finite volume method, the governing differential operator $\mathcal{L}$ has the form

$$\int_{\mathcal{I}} \mathcal{L} f dV = \int_{\mathcal{I}} \text{div}\phi(f) + G(f) dV = 0 \,, \tag{3.21}$$

where ϕ is a vector valued and depends on f and G is an ordinary function. The equation is re-written under the Gauß integral theorem and yields

$$\int_{\mathcal{I}} \mathcal{L} f dV = \int_{\partial_{\mathbf{T}}} \phi_n(f) dA + \int_{\mathcal{I}} G(f) dV = 0 \,, \tag{3.22}$$

where ϕ_n denotes the normal component of the resulting vector of the functional ϕ. A first approach for the formulation of finite volume schemes is based on the assumption that for each vertex the local shape function defined on the incident cells is evaluated by calculating an integral on the surface of the dual cell. The part of the boundary of the dual cell of the vertex $\mathbf{v}$ which is in a cell $\mathbf{c}$ is used as integration domain.

$$R = \int_{\mathcal{T}} \mathcal{L} f dV \tag{3.23}$$

$$R(\mathbf{v}) = [\sum_{VC}^{\underline{v}} \sum_{CV}^{\underline{c}} K(\bullet, \underline{c}, \underline{v})](\mathbf{v}) \ , \tag{3.24}$$

where K denotes the following integral formula. The function value within the cell (and therefore also on the dual surface) depends on all quantity q values associated with a vertex incident to the respective cell.

$$\Phi(\mathbf{v}, \mathbf{c}, \mathbf{w}) := \int_{\partial \text{dual}(\mathbf{v}) \cap C} \phi_n(f(\mathbf{c})) dA \tag{3.25}$$

$$f(\mathbf{c}) := \sum_{CV}^{\underline{c}} q \cdot f(\bullet, \underline{c}) \tag{3.26}$$

The function $f(\mathbf{c})$ denotes the value of the solution function within the respective cell of evaluation, $f(\mathbf{w}, \mathbf{c})$ denote the cell based shape functions of the function space $\mathcal{F}$.

3.2.2 Edge-Based Boundary Integrals

In the former section the integration domain of the dual cell of the vertex was separated cell-wise by

$$\partial_{\text{dual}}(\mathbf{v}) = [\bigcup_{VC} \partial \text{dual}(\bullet) \cap u_1](\mathbf{v}) \ . \tag{3.27}$$

As a simplification of this method, the boundary surface can be written in the following manner:

$$\partial_{\text{dual}}(\mathbf{v}) = \bigcup_{VE} \text{dual}(\bullet) \ . \tag{3.28}$$

This means that the surface can be written in terms of edges incident to a given vertex. The boundary integral can therefore be written as follows:

$$\mathcal{L}f(v) = \sum_{VE}^{\underline{v}} \sum_{EC}^{\underline{w}} K(\underline{v}, \underline{e}, \bullet) \, , \tag{3.29}$$

Even though more traversal methods are required, the evaluation of the inner integral can be simplified, because the integration domain (surface) is a subset of a plane.

$$\Phi(\mathbf{v}, \mathbf{e}, \mathbf{c}) := \int_{\text{dual}(\mathbf{e}) \cap \mathbf{c}} \phi_n(f(\mathbf{c}))dA \tag{3.30}$$

Under the assumption that there is no change of the normal derivative throughout the surface element or – the weaker assumption – that the integral mean of the normal derivative throughout the surface element can be calculated by a point-to-point interpolation along the edge, shape functions only need to be explicitly defined on the edges whereas the function value on the cells themselves is not relevant for the calculations.
As a consequence, the integration of the functional ϕ_n does not need to be performed explicitly for each part of the boundary of the dual cell of a vertex. In such a case only one mean value for the flux functional ϕ_n has to be determined and multiplied with the area of the surface part A associated to the edge.

$$R(\mathbf{v}) = [\mathcal{L}f](\mathbf{v}) = [\sum_{VE}^{\underline{v}} \Phi(\underline{v}, \bullet)](\mathbf{v}) \tag{3.31}$$

In this case the integral is evaluated on the complete surface part dual to the edge $\mathbf{e}$.

$$\Phi(\mathbf{v}, \mathbf{e}) := \int_{\text{dual}(\bullet)} \phi_n(f(\bullet))dA \tag{3.32}$$

Accordingly, the function passed to the functional ϕ_n does not depend on the actual position on the surface and is therefore multiplied with the area of the surface in order to determine the flux. The integral can be cancelled and J yields

$$\Phi(\mathbf{v}, \mathbf{e}) := \phi_n(f(\mathbf{e}))A(\mathbf{e}) \, , \tag{3.33}$$

where $\phi_n(f(\mathbf{e}))$ is evaluated on the point at the intersection of the edge and the surface element dual to the edge. This point is - due to the Delauney property - usually located in the middle between the bounding points of the edge.

This formula is based on the assumption that the flux at the edge is representative for the whole surface and does not vary along the surface or the integral at the surface only depends on the flux at the edge. It is clear that this assumption can not be fulfilled for different kinds of shape functions, but for most of the cases such a condition is fulfilled approximately. For linear shape functions the condition is fulfilled exactly, because the tangential derivative between two neighboring elements is identical and the derivatives are constant throughout the cells.

In some cases, e.g. the shape functions for the Scharfetter Gummel discretization scheme [3] on a two-dimensional simulation domain, the shape functions are not defined at all for the elements. Moreover, it is assumed that there are sufficiently well behaved shape functions (which are not explicitly given) that fulfill the required properties approximately.

3.2.3 Topological Structure of the Finite Volume Scheme

The finite volume scheme shown in Section 3.2.1 requires the following topological operations: from a vertex all incident cells are required (determination of the integration domain) and for these cells all incident vertices have to be traversed (determination of the function values). Such a traversal scheme is identical to the traversal scheme of finite elements which only have vertex-based shape functions.

The method described in Section 3.2.2 requires all incident edges of the initial vertex for the determination of the integration domain. This integration domain is covered by the union of two cells which are incident with the edge. In order to determine the shape functions given on these cells, the function coefficients associated with the incident vertices of the cell have to be determined.

If further requirements of Section 3.2.2 are given or assumed, the evaluation of the finite volume differential operator is obtained by traversing all incident edges of the vertex and

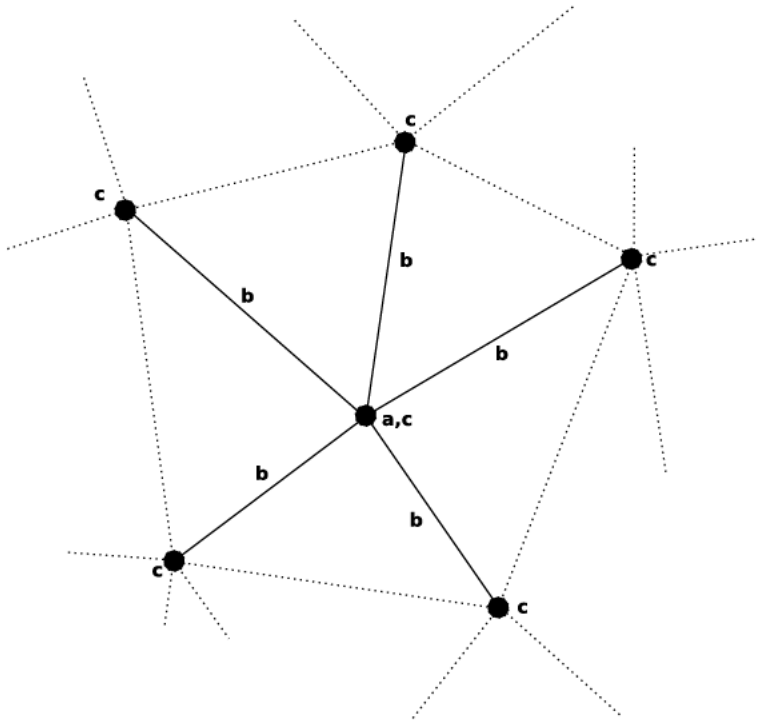

Figure 3.5: Traversal scheme of a finite volume discretization from Section 3.2.1.

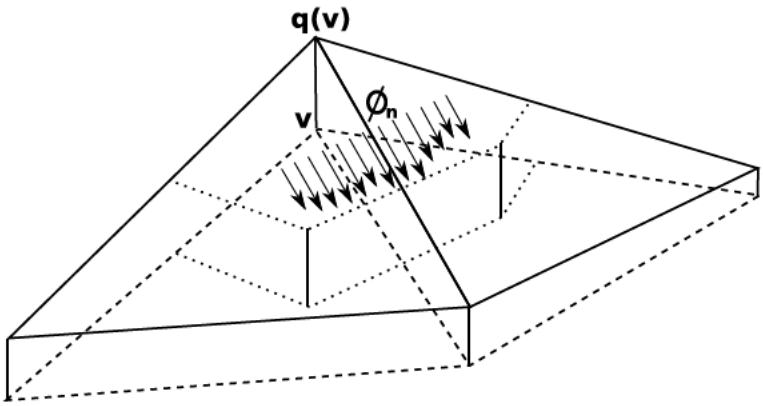

Figure 3.6: Traversal scheme of a finite volume discretization from section 3.2.2, first method.

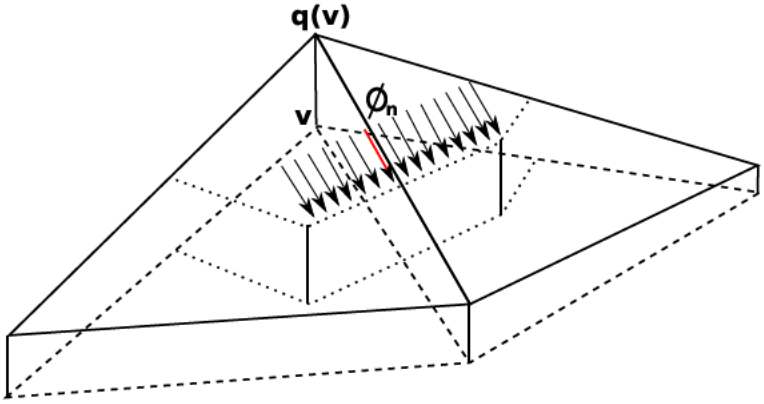

Figure 3.7: Traversal scheme of a finite volume discretization from Section 3.2.2, second method.

subsequently traversing the vertices on the edge for calculating the function value (on the edge only).

In the following sections, the second method of Section 3.2.2 is used for the specification of the finite volume schemes. For the discretization of the Laplace equation the requirements for this method are given for the discretization of the semiconductor transport equations [79] the requirements are assumed to be fulfilled approximately.

3.2.4 Laplace Equation

One method typically used in simulation is the dual graph method. As an introductory example, the numerical solution of the Laplace equation is shown. In this case the functional ϕ is the gradient and $g = 0$.

For linear shape functions, the function $f(\mathbf{e})$ along one edge can be written as

$$f(\mathbf{e}) = \sum_{EV}^{\underline{e}} q f(\bullet, \underline{e}) \,, \tag{3.34}$$

The function $f(\mathbf{e}, \mathbf{v})$ denotes a shape function defined on the edge-skeleton of the cell complex. These functions have local support on one edge $\mathbf{e}$ and yield unity in the vertex

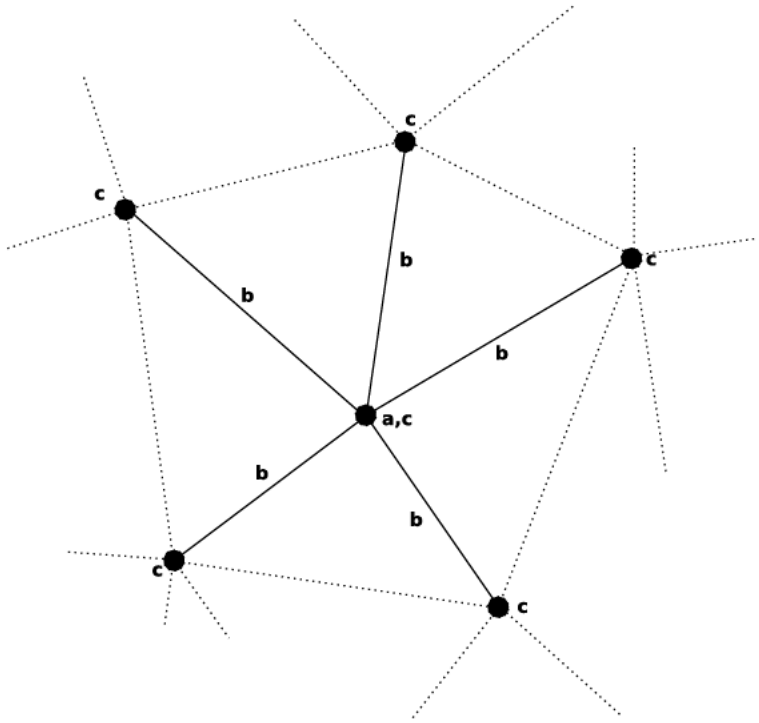

Figure 3.8: Topological neighborhood required for finite volume schemes.

v. Consequently, the gradient is constant along the edge and, therefore, the evaluation point of the function as described in () does not play any role.

$$\Phi(\mathbf{e},\mathbf{v}) := \frac{1}{l_{\underline{e}}} \sum_{EV}^{\underline{e}} \mathcal{O}(\underline{e},\bullet) q \tag{3.35}$$

The complete expression for the discretized Laplace equation yields

$$R(\mathbf{v}) := m(\mathbf{v})\Delta(f(\mathbf{v})) = 0\,,$$

$$R = [m\mathcal{L}f] = \sum_{VE}^{\underline{v}} \mathcal{O}(\bullet,\underline{v}) \frac{A}{l} \sum_{EV}^{\underline{e}} \mathcal{O}(\underline{e},\bullet) q\,. \tag{3.36}$$

3.2.5 Volume Integration

Hitherto, the boundary integrals have been discussed, while the integration of the terms g was not mentioned at all. These integrals are usually on the associated dual cell of the

vertex. In the following it is assumed that for the integration the functional g has the form of a (n-dimensional) Dirac pulse which is located in the geometrical point of the vertex.

Another method is to assume that the variable value is distributed uniformly on the cell. Even though such an interpretation of the quantity value is possible, it inevitably leads to an inconsistence: Given the Poisson equation, the functional g is the cell-wise constant right hand side of the equation and does not depend on the function f. If $g(f)$ is constant, the assumption of linear shape functions as chosen for the Laplace equation is inconsistent, because the Laplace operator applied to the shape function always yields zero.

Therefore, the integral is evaluated by multiplying the function value associated with the vertex with the volume of the dual cell of the vertex. If the functional g depends on f, e.g. $g = \sin(f)$, the evaluation of the cell is more complicated and not further discussed here. In most of the cases it is appropriate to approximate $f = f(\mathbf{v})$ throughout the dual cell of $\mathbf{v}$. Therefore, the following approximation can be derived:

$$\int_{\text{dual}(\mathbf{v})} g(F)dV = g(f(\mathbf{v})) \cdot m(\text{dual}(\mathbf{v})) = g(f(\mathbf{v})) \cdot m(\text{dual}(\mathbf{v})) \,, \tag{3.37}$$

where the function m denotes the volume of a given dual cell. For the sake of brevity, these two functions can be combined and one obtains the final formulation.

The distribution of quantities can be obtained from the topological view on the discretization scheme. All solution quantities which are required to form the solution function are associated with vertices. Furthermore, all quantities which are required to specify the function g are associated with the vertex.

Geometrical quantities are associated with the topological elements or their dual elements for which they are required. As an example, the volume of a cell (finite volume) is assigned to the vertex which is dual to the respective cell. The area of the dual surface of an edge is stored as an edge-based quantity. The length of an edge is also stored as edge-based quantity.

3.2.6 Drift-Diffusion Semiconductor Equations

For the discretization of the drift-diffusion semiconductor equations the flux terms have to be considered. In a divergence formulation, the stationary drift-diffusion semiconductor equations [33] can be specified as

$$\begin{aligned} \mathrm{div}\,\overrightarrow{\phi} &= e_0(n - p + C)\ , \\ \mathrm{div}\overrightarrow{J_n} &= e_0 R\ , \\ \mathrm{div}\overrightarrow{J_p} &= -e_0 R\ . \end{aligned} \tag{3.38}$$

The associated flux terms $\overrightarrow{D}$, $\overrightarrow{J_n}$ and $\overrightarrow{J_p}$ are usually defined in the following way

$$\begin{aligned} \overrightarrow{\phi_n} &= -\varepsilon \mathrm{grad}\psi \\ \overrightarrow{J_n} &= e_0 n \mu_n \mathrm{grad}\psi + e_0 D_n \mathrm{grad}n \\ \overrightarrow{J_n} &= e_0 p \mu_p \mathrm{grad}\psi - e_0 D_p \mathrm{grad}p \end{aligned} \tag{3.39}$$

The solution functions are the carrier concentrations n and p and the electrostatic potential ψ. Here, ε denotes the permittivity coefficient, μ denotes the mobility of the carriers, R denotes the generation/recombination rate for the carriers, and N denotes the net doping.

For the sake of simplicity, only the electron carrier density is considered. By replacing the respective signs, the calculations can also be performed for the hole continuity equation. However, the final result will also be given for holes.

First, the Poisson equation is discretized in analogy to the Laplace equation. The respective flux relation for the dielectric displacement can be written in as expression evaluated

on an edge. For this evaluation, the function values of the potential are evaluated on the bounding vertices of the edge.

$$\phi_n = \varepsilon \cdot \mathrm{grad}(n)\psi$$

$$\phi_n(\mathbf{e}) = [\varepsilon \cdot \sum_{EV}^{\underline{e}} \mathcal{O}(\underline{e}, \bullet)\psi] \tag{3.40}$$

The application of the finite volume scheme on this flux operator yields the following discretization scheme. In addition, the right hand side is approximated by multiplying with the cell volume.

$$R_\psi(\mathbf{v}) := [\sum_{VE}^{\underline{v}} \frac{A\varepsilon}{d} \sum_{EV}^{\underline{e}} \mathcal{O}(\underline{e}, \bullet)\psi - me_0(n - p + C)](\mathbf{v}) \tag{3.41}$$

It is reasonable to associate the flux related quantity ε with the edge so as to obtain a straight-forward formulation. If the dependence of the electric field and displacement is modeled in a non-linear manner, all coefficients which are related to the determination of the displacement from the field strength can be associated to the respective edge.

Secondly, the continuity equation for electrons is considered. In contrast to the linear interpolation for the potential along a connecting edge of two vertices, the carrier concentration is interpolated using the Scharfetter-Gummel discretization [3]. The interpolation does hereby depend on the potential which is given on the boundary nodes of the respective edge.

Even though the Scharfetter Gummel discretization was usually defined for one-dimensional discretization, many simulation tools [59] use the interpolation along the edges for two-dimensional or three-dimensional simulation. In its original formulation the Scharfetter Gummel discretization of the electron flux can be written as

$$\phi_n := U_{\mathrm{th}}\mu_n(n_i\mathcal{B}(-\Delta\psi) - n_j\mathcal{B}(\Delta\psi)) \tag{3.42}$$

where n_i and n_j denote the carrier concentration in the boundary vertices indiced by i and j. $\mathcal{B}$ denotes the Bernoulli function

$$\mathcal{B}(x) = \frac{x}{e^{-x} - 1}\,. \tag{3.43}$$

In analogy to the permittivity coefficient, the mobility for carriers can be stored on the edges. Furthermore, coefficients describing the dependence of the mobility on carrier concentration and potential can be associated with the respective edge. In the following model, U_{th} is assumed to be a constant or a model-dependent global variable which is not associated to topological elements.

$$\phi_n := U_{\text{th}}\mu_n \sum_{EV}^{\underline{e}} \mathcal{O}(\underline{e}, \bullet)(n\mathcal{B}(-\mathcal{O}(\underline{e}, \bullet)\Delta\psi_{\underline{e}}) \tag{3.44}$$

The potential dependent term in the Scharfetter Gummel scheme $\Delta\psi$ denotes the potential difference between two bounding vertices of an edge, $\psi_j - \psi_i$. Based on the edge the formulation can be written in the following manner:

$$\Delta\psi = \sum_{EV}^{\underline{e}} \mathcal{O}(\bullet, \underline{e})\psi \ . \tag{3.45}$$

3.2.7 Conclusion

The formulation of the discretized differential operators is aided by the topological operations and the functional means of specification.

The underlying topological framework needs to provide edges and vertices (actually no cells) and for the sake of simplicity, even graph data structures providing incidence relations and traversal functions can be employed for the implementation. Furthermore, a geometrical algorithm for the determination of the areas and volumes of the dual surfaces and cells is required. Another requirement is that an orientation function $\mathcal{O}$ is available which determines, if a vertex is a source or a sink of an edge. For the use within a formula the explicit orientation of the single edges is not relevant. The formula is said to be invariant with respect to changes of the topological orientation. There are containers to provide storage means for both, quantities on vertices as well as quantities on edges.

In contrast to the initial formulation of the Scharfetter Gummel scheme, an algorithm for the implementation of the discretization scheme can be directly derived from the given formula.

3.3 Boundary Element Method

In finite simulation problems the boundary of the simulation domain is treated separately and has to be considered. The actual simulation domain is chosen in a manner that all irregularities such as non-linearities and inhomogeneities are covered. Physical phenomena beyond the boundary are usually not of great importance and are therefore neglected. The boundary conditions which are assumed at the boundaries of the simulation domain are artificially defined by the shape of the considered simulation domain. As a consequence, it may occur, e.g., that for wave equations, waves are reflected at the artificial boundaries so that artefacts occur in the simulation result, which compromise the quality of the results. Less remarkable but still evident is a behavior that can be observed when applying homogeneous Neumann boundary conditions on the solution of the Laplace equation. If more space is between the boundary and the relevant configurations, the solution can eventually become more precise, when the infinity of the surrounding space is of relevance.

Boundary element methods [56, 80] circumvent these difficulties, because the tesselation of the underlying space is only required on the boundaries. The surrounding of the boundary is assumed to be linear, homogeneous, and isotropic. In this case it is not necessary to tessellate the domain far distant from the actual places of interest, but only the boundary has to be tessellated. Therefore, quantities are only stored on topological elements on the boundary.

A feature which makes the application of boundary elements attractive to simulation is that boundary elements and finite elements can be coupled in a simple manner. A practical example of the boundary element method is shown in [12], where a superconductive quadrapole coil is simulated and the effects of the surroundings are explicitly considered.

3.3.1 Standard Formulation

In contrast to finite elements or finite volumes, shape functions do not have local support but they are non-zero throughout the simulation domain. Normally, basis functions are defined in a manner that they are defined by functions having boundary facets as local

support. These facets are incident with a common vertex. In the two-dimensional case, the facets are (boundary) edges incident with a boundary vertex.

The typical formulation of a boundary element scheme can be written in the following manner (Galerkin formulation)

$$R_i := \int_{\partial\mathcal{D}} \int_{\partial\mathcal{D}} \mathcal{M} f_i(s) f_j(s') G(s,s') ds ds' = 0 \ , \tag{3.46}$$

where i and j are indices of vertices. For each vertex an equation $R_i = 0$ is assembled. The fundamental solution [81] function G denotes a fundamental solution of the respective differential operator $\mathcal{L}(f)$:

$$\mathcal{L}(G(\mathrm{r},\mathrm{r}')) = \delta(\mathrm{r}-\mathrm{r}') \tag{3.47}$$

The parameters s and s' denote the position at the boundary surface or curve. The linear boundary operator $\mathcal{M}$ can be written as linear combination in the following manner.

$$\mathcal{M}(f) := \alpha f + \beta \partial_n f \tag{3.48}$$

3.3.2 Implementation-Based Formulation

The outer integral along the boundary of the simulation domain can be simplified in the following manner by assuming the local support of the basis function:

$$R(\mathbf{v}) := \sum_{\mathcal{B}V}^{\underline{v}} q \int_{\partial D} \int_{\partial D} \mathcal{M}(f(\bullet))(s) f(\underline{v})(s') G(s,s') ds ds' = \sum_{\mathcal{B}V}^{\underline{v}} q K(\underline{v}, \bullet) \ . \tag{3.49}$$

The double integral is evaluated over the boundary of the simulation domain $\mathcal{D}$. $\mathcal{M}$ denotes the boundary operator, G denotes the fundamental solution. The basis function f is defined by the vertex and stands for the function which equals unity in the argument vertex. The quantity q defines the weighting coefficient for the respective basis function. The integral expression depends on the type of weighting functions and the shape functions. In this case the Galerkin method is used which means that the shape functions and the weighting functions are equal. Furthermore, the integral expression depends on the

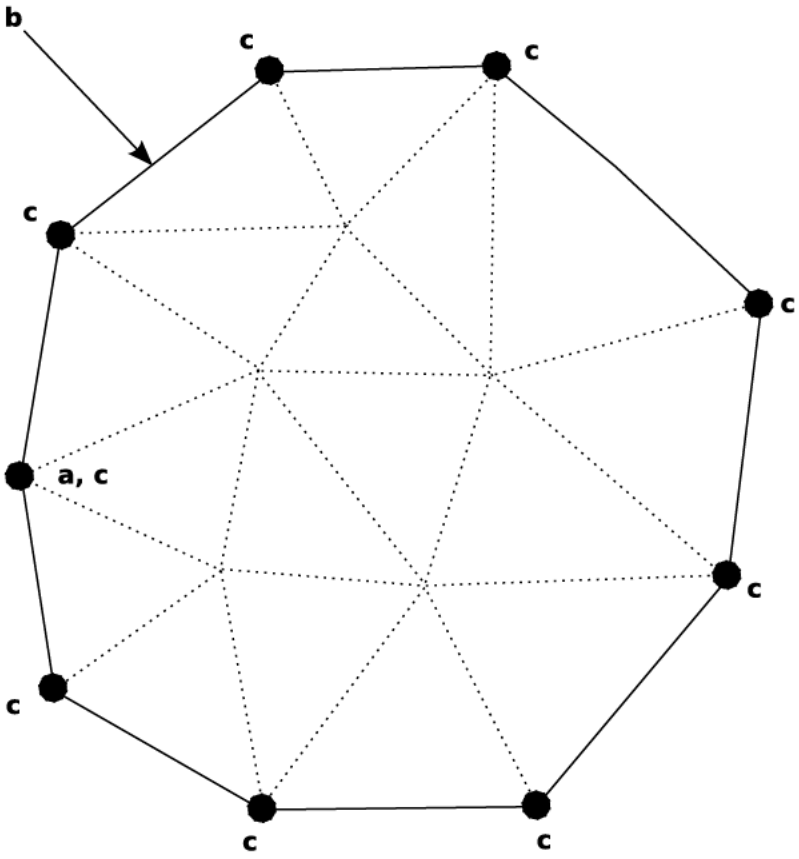

Figure 3.9: Topological dependences and requirements for the implementation of the boundary element scheme.

choice of the boundary conditions used. Figure 3.9 shows the topological dependences and requirements for the implementation of the boundary element discretization scheme. For the implementation the topological incidence function $\mathcal{B}V$ is required. Due to the linearity of the boundary condition, the formulation can be re-written in the following form, where the coefficients α and β can be extracted by treating the expressions αf and $\beta \partial_n f$ of (3.48) separately.

$$K = \alpha K'(\bullet, \bullet) + \beta K''(\bullet, \bullet) \tag{3.50}$$

It can be seen that the coefficients of the boundary condition are associated with the respective boundary vertices. The evaluation of the integral terms K depends on the shape of the boundary facets or edges as well as on the shape of the basis functions and the weighting functions.

3.3.3 Conclusion

Due to the formulation according to (3.50) the topological structure of the discretization scheme becomes clear. The formal representation eases the view on the underlying data structures and it is shown that couplings between all boundary elements exist. Furthermore it can be seen directly that the resulting system matrix from this discretization scheme is nonzero in all elements.

3.4 Finite Difference Schemes

Finite difference schemes differ from the other mentioned schemes, because it does not rely on a functional discretization, but only represents the functions used for the solution of the governing equations by a mapping of points and values. Even though this does not lead to a solution function, it can be shown that for many cases that such a method is consistent and convergent and produces adequate solutions. Especially for time-stepping, finite difference schemes are often used in combination with other discretization schemes so that the results obtained by the simulation are defined as continuous functions on single time slices.

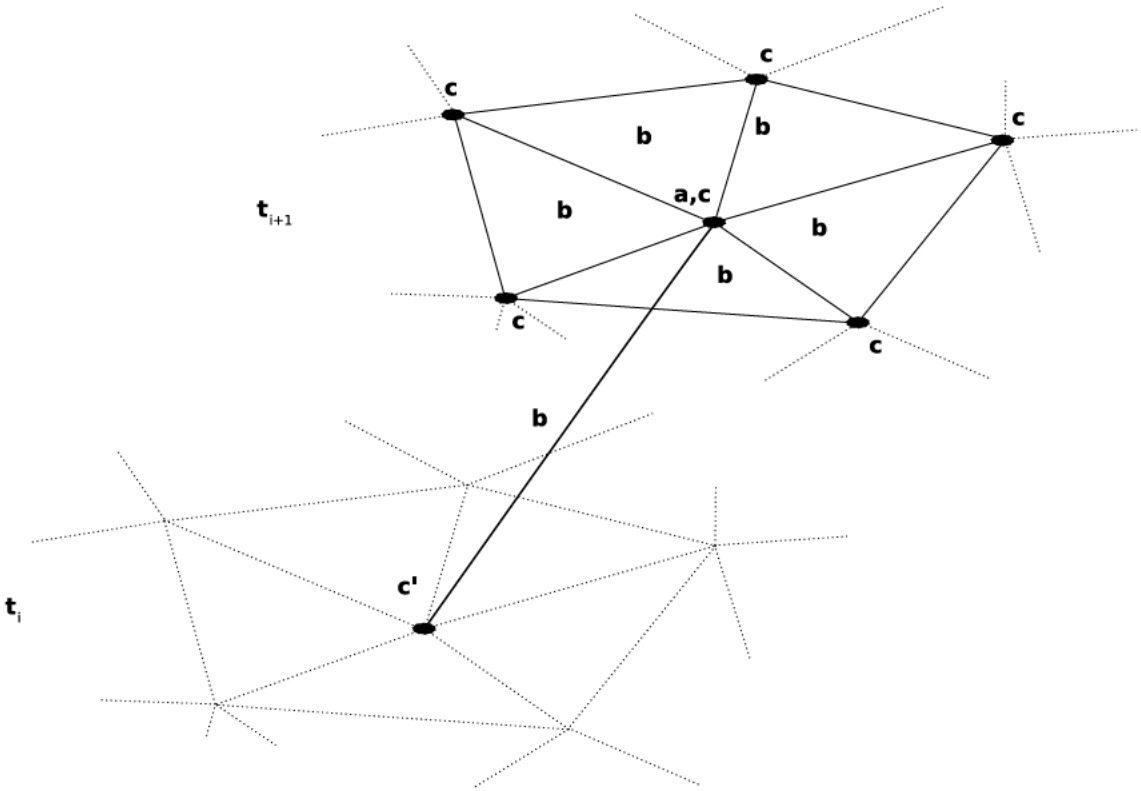

Figure 3.10: The backward Euler scheme is a time discretization scheme using finite differences for the time discretization and, for instance, finite elements for spatial discretization.

Finite difference schemes is very flexibly employable and a common basis for all operations cannot be defined by introducing a function space and treating the results of the finite difference scheme in the same manner as, e.g., the result of a finite element solution function. Finite difference schemes treat functions as defined by argument-value pairs and form interpolation functions ad-hoc for the special field of application.

The aim of this section is to find a formulation based on the systematics according to Section 2.5 and investigate the differences of the topological base operations.

3.4.1 Conventional Method

Again, the Laplace operator is considered for the specification of a finite difference scheme. A typical formula for a finite difference scheme may have the following shape:

$$(\partial_{xx} + \partial_{yy})u = \frac{u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1}}{h^2} \tag{3.51}$$

From this formulation the following implicit assumptions are taken: Firstly, a quantity u is defined on a grid which is located by two independent indices i and j indicating discrete values on the x as well as the y axis. Furthermore, finite difference schemes are restricted to grids, namely to topological structures in which each vertex is implicitly assigned neighboring vertices by the structure of the grid.

If the distance between two neighboring vertices, here denoted as h is equal for all vertices, independently from the direction, the formulae can be written in a very simple manner. Otherwise, the distance between the single vertices has to be calculated separately either by using the length of the edges or alternatively by explicitly using the vertex coordinates.

3.4.2 Topological Neighborhood Considerations

The first step of a finite difference simulation is the determination of the neighborhood. In a very simple two-dimensional case, the neighborhood of a vertex is defined by the four vertices which are covered by the edges incident with the original vertex. It is also assumed that the respective vertex has to fulfill some geometric criterion to be considered as neighboring. In order to obtain all neighboring vertices of a given basis vertex, the

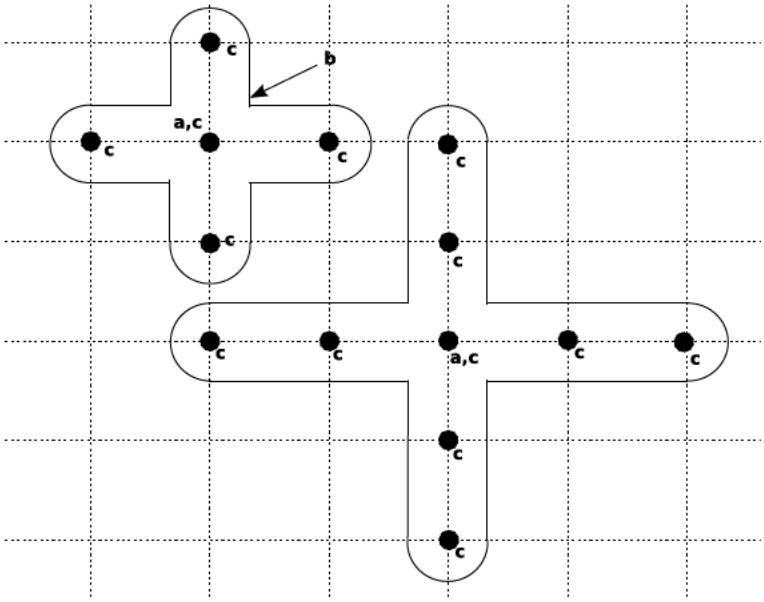

Figure 3.11: The finite difference scheme applied to a structured cell complex. Two neighborhoods **b** are shown for the initial vertex **a**. The neighborhood comprising five points can be used in order to determine differential operators up to second order. The neighborhood comprising nine points can be used in order to determine differential operators up to fourth order.

traversal function VNV is introduced. Figures 3.11 and 3.12 show possible neighborhoods used for the finite difference simulation applied to structured grids and to unstructured meshes.

For each neighboring vertex $\mathbf{w}$ of a basis vertex $\mathbf{v}$ (also the vertex $\mathbf{v}$ itself, which is within its own neighborhood), the following series expansion is written with respect to the location of the initial vertex $\mathbf{v}$:

$$u(\mathbf{v},\mathbf{w}) = u(\mathbf{v}) + \partial_x(x(\mathbf{w}) - x(\mathbf{v})) + \partial_y(y(\mathbf{w}) - x(\mathbf{v})) + \ldots + \mathcal{O}(h^n) \tag{3.52}$$

For a given truncation error, an appropriate number of neighboring vertices has to be chosen. If a given truncation error is required for a Taylor expansion, for instance $\mathcal{O}(h^3)$, first order and second order terms have to be considered. Each partial derivative of u as well as u itself is considered as solution variable in a coordinate system. If a trunction error of higher order is required, the number of solution variables increases. Accordingly, a

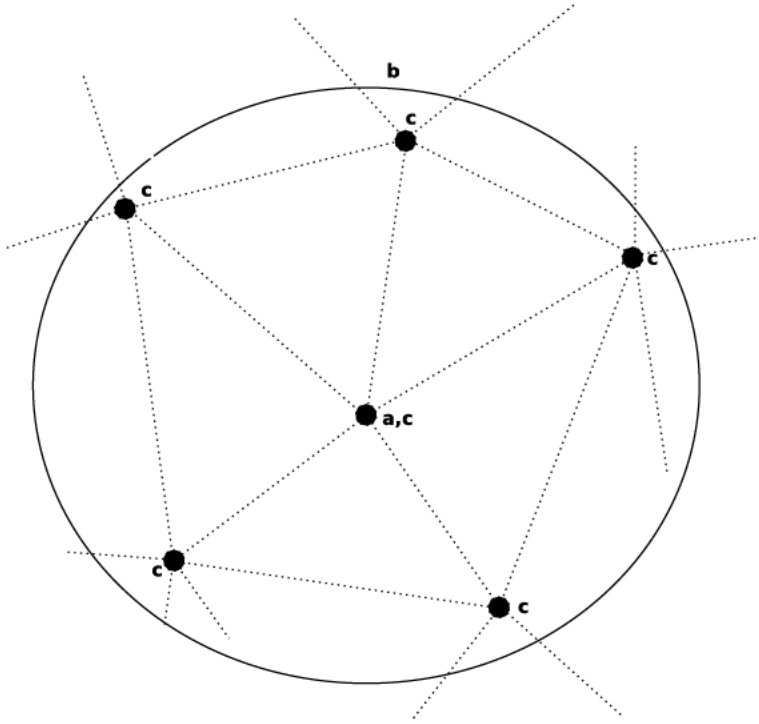

Figure 3.12: The finite difference scheme applied to an unstructured cell complex. The neighborhood **b** comprises five vertices **c**. The vertices **c** which are used for the evaluation of the respective differential operator can be either determined by finding the vertices with the smallest distance to the initial vertex **a** or by determining the vertices which are incident with edges incident with the initial vertex **a**.

larger number of vertices (neighborhood) is required to solve the equation and determine the dependence of the position of the vertices on the value of the derivatives determined. Moreover, it is possible to determine higher-order derivatives of a function by using a higher-order Taylor expansion. This implies that the use of higher-order derivatives necessarily increases the number of vertices (neighborhood) used for the Taylor expansion. One obtains a local equation system which can be solved analytically in special cases, especially under the assumption that the local grid intervals are equal (which shall be assumed in the following). One obtains the following equation system

$$\begin{aligned} \mathbf{u}(\mathbf{w}) &= \mathcal{G}(\mathbf{w}) \cdot [\partial](\mathbf{w})(+[\mathcal{O}(h^N)]) \\ [\partial](\mathbf{w}) &= \mathcal{G}(\mathbf{w})^{-1} \cdot \mathbf{u}(\mathbf{w})(+[\mathcal{O}(h^N)]) \ , \end{aligned} \tag{3.53}$$

where the vector of function values $u(\mathbf{w})$ is denoted as $\mathbf{u}(\mathbf{w})$. The vector $[u, \partial_x u, \partial_y u, \ldots]$ is denoted as $[\partial]$ evaluated on the vertex $(\mathbf{w})$. The matrix containing the geometrical coefficients is referred to as $\mathcal{G}$. By inverting the geometrical coefficient matrix $\mathcal{G}$, one obtains the vector of derivatives. The matrix $\mathcal{G}$ can be written as follows:

$$\mathcal{G} = \bigsqcup_{VNV}^{\underline{v}} [1; x - x_{\underline{v}}; y - y_{\underline{v}}; (x - x_{\underline{v}})^2/2; (x - x_{\underline{v}})(y - y_{\underline{v}}); (x - x_{\underline{v}})^2/2; \ldots]^T \ . \tag{3.54}$$

The vector $\mathbf{u}$ can be written as the vector of quantities within the neighborhood

$$\mathbf{u} = \bigsqcup_{VNV} q = (q_1, \ldots, q_n)^T \ . \tag{3.55}$$

A linear differential operator $R = \mathcal{L}$ can be written as inner product of a given vector $\mathbf{d}$ and the vector of derivatives $[\partial]$.

$$R = \mathcal{L} = \langle \mathbf{d}, [\partial] \rangle (= 0) \tag{3.56}$$

By extending the term of canonic partial differential operators $[\partial]$, one obtains the residual expression R. It has to be assured that the order of elements that are passed to the $\bigsqcup_{VNV}$

operation is the same for the evaluation of the matrix $\mathcal{G}$ and the vector of function values $\mathbf{u}$.

$$R = \mathbf{d} \cdot \mathcal{G}^{-1} \cdot \mathbf{u} \tag{3.57}$$

Inserting (3.54) without the second order terms and (3.55) into (3.56) yields the following expression.

$$R = \mathbf{d} \cdot [\bigsqcup_{VNV}^{\underline{v}} [1; x - x_{\underline{v}}; y - y_{\underline{v}}; \ldots]^T]^{-1} \cdot \bigsqcup_{VNV} q \tag{3.58}$$

If the geometric constants are known explicitly, the inverse of the matrix $\mathcal{G}$ can be derived directly. In this case the matrix $\mathcal{G}$ can be written as:

$$\mathcal{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & h & 0 & h^2/2 & 0 \\ 1 & -h & 0 & h^2/2 & 0 \\ 1 & 0 & h & 0 & h^2/2 \\ 1 & 0 & -h & 0 & h^2/2 \end{bmatrix}$$

The inverse $\mathcal{G}^{-1}$ of the matrix $\mathcal{G}$can be obtained as follows:

$$\mathcal{G}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1/(2h) & 1/(2h) & 0 & 0 \\ 0 & 0 & 0 & 1/(2h) & 1/(2h) \\ -2/h^2 & 1/h^2 & 1/h^2 & 0 & h^2/2 \\ -2/h^2 & 0 & 0 & 1/h^2 & 1/h^2 \end{bmatrix}$$

The standard finite difference formulae can be obtained easily from the coefficients of the matrix. By the specification of $\mathbf{d}$, a linear combination of lines of the matrix $\mathcal{G}^{-1}$ is obtained by multiplication. The line vector $\mathbf{k} = \mathbf{d} \cdot \mathcal{G}^{-1}$ denotes the coefficients with which the values of the single neighboring vertices are coupling. It is associated with a vertex and contains coupling coefficients, each of which is directly associated with a neighboring vertex. Alternatively, the vector $\mathbf{k}(\mathbf{w})$ can be written element-wise, where the elements are written as $K(\mathbf{w}, \mathbf{v})$. In this case $\mathbf{w}$ denotes the vertex and $\mathbf{v}$ denotes a vertex in the neighborhood of $\mathbf{w}$. By evaluating the inner product of $\mathbf{k}$ with the function values of the

single vertices stored in the vector $\mathbf{u}$, the residual expression is obtained. The evaluation of the inner product finally yields

$$R = [\mathbf{k} \cdot \bigsqcup_{VNV} q] = [\bigsqcup_{VNV}^{\underline{w}} K(\underline{w}, \bullet) \cdot \bigsqcup_{VNV} q] = [\sum_{VNV}^{\underline{w}} K(\underline{w}, \bullet) q] \tag{3.59}$$

It can be seen easily that the algebraic structure of (3.59) is similar to other discretization schemes. While other discretization schemes use topologically specified double sums with topological traversal in order to obtain the couplings between solution quantities, the definition of the neighborhood can be defined more freely. While in one case the neighborhood is defined topologically like for finite elements or finite volumes, the neighborhood can also be determined by geometrical considerations.

3.4.3 Laplace Operator

In the special case of the Laplace operator the vector $\mathbf{d}$ has the following form

$$\mathbf{d} = [0, 0, 1, 0, 1, 0, \ldots] \tag{3.60}$$

For a grid with a regular distance h between neighboring vertices, the geometrical coefficient matrix $\mathcal{G}$ yields for five and nine neighboring points:

$$\mathcal{G}_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & h & 0 & h^2/2 & 0 \\ 1 & -h & 0 & h^2/2 & 0 \\ 1 & 0 & h & 0 & h^2/2 \\ 1 & 0 & -h & 0 & h^2/2 \end{bmatrix}$$

$$\mathcal{G}_9 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & h & 0 & h^2/2 & 0 & h^3/6 & 0 & h^4/24 & 0 \\ 1 & -h & 0 & h^2/2 & 0 & -h^3/6 & 0 & h^4/24 & 0 \\ 1 & 0 & h & 0 & h^2/2 & 0 & h^3/6 & 0 & h^4/24 \\ 1 & 0 & -h & 0 & h^2/2 & 0 & -h^3/6 & 0 & h^4/24 \\ 1 & 2h & 0 & 2h^2 & 0 & 3h^3/2 & 0 & 8h^4/3 & 0 \\ 1 & -2h & 0 & 2h^2 & 0 & -3h^3/2 & 0 & 8h^4/3 & 0 \\ 1 & 0 & 2h & 0 & h^2 & 0 & 3h^3/2 & 0 & 8h^4/3 \\ 1 & 0 & -2h & 0 & h^2 & 0 & -3h^3/2 & 0 & 8h^4/3 \end{bmatrix}$$

After eliminating derivatives which do not appear in the series expansion of the single points, the matrices can be re-written. The derivative vector $[\partial]$ is written as:

$$[\partial] = [u, \partial_x u, \partial_y u, \partial_{xx} u, \partial_{yy} u, \partial_x xxu, \partial_{yyy} u, \partial_{xxxx} u, \partial_{yyyy} u \ldots] \,. \tag{3.61}$$

The vector $\mathbf{d}$ is reduced to the following form:

$$\mathbf{d} = [0, 0, 1, 1, 0, \ldots] \tag{3.62}$$

Inserting into the formula (3.58) yields the well known expressions. The function $K(\mathbf{v}, \mathbf{w})$ can be written as

$$K(\mathbf{w}, \mathbf{v}) := \left\{ \begin{array}{l|c} \mathbf{v} = \mathbf{w} & -4h^{-2} \\ \mathbf{v} \neq \mathbf{w} & h^{-2} \end{array} \right.$$

For nine points the following formula is obtained:

$$K(\mathbf{w}, \mathbf{v}) := \left\{ \begin{array}{l|c} \mathbf{v} = \mathbf{w} & -5h^{-2} \\ d(\mathbf{w}, \mathbf{v}) = h & 4/3h^{-2} \\ d(\mathbf{w}, \mathbf{v}) = 2h & -1/12h^{-2} \end{array} \right.$$

3.4.4 Conclusion

The only topological requirement on the finite difference method is that a neighborhood of vertices can be found for each single vertex. This topological function for retrieving all neighboring vertices of a given vertex is denoted as traversal function VNV.

It also has to be mentioned that the neighboring vertices of a given vertex can be found much easier, if a structured grid is given. However, a finite difference method can also be applied, if no further topological information is available. For instance, the neighborhood can be defined by the closest n points of a given point.

If the topological information is given, one can chose to find the neighborhood by finding all incident points which are covered by an edge or a cell incident to the original vertex. It is also possible to use more vertices by choosing more neighboring cells by further applying the incidence relation such as for higher order finite difference schemes on structured grids [82].

3.5 Final Considerations

The following remarks are common for all discretization schemes for differential equations and show their structural similarity, when written in the formalism according to Chapter 2. Even if two or more solution variables are available on a topological element, the neighborhood is usually the same for all solution quantities, so that the considerations can be restricted to one solution quantity. In order to obtain the internal couplings within the system matrix, all quantities as well as their couplings have to be considered.

The first common feature which is fulfilled by all discretization schemes is that the solution quantity is located on topological elements. These elements are mostly vertices. For higher-order arrangements, especially for finite elements also other topological elements such as edges can be used.

Each of the discussed discretization schemes establishes a topology defined by neighborhoods [64], where each element is assigned a neighborhood of elements. The neighborhood can be either directly retrieved from the underlying cell complex or derived from geometrical properties, e.g. for the finite difference method. In the following, the discretized differential operator is written as weighted sum of quantity values in the neighborhood of a topological element.

The process of obtaining neighboring elements is defined by two steps: firstly, the neighborhood b of the initial element a is retrieved by the traversal method AB. Secondly, all elements c of the neighborhood elements b, on which relevant quantities are stored, are retrieved by the traversal methods BC.

$$\sum_{AB}^{\underline{a}} K(\underline{a}, \bullet) \sum_{BC}^{\underline{b}} qK'(\underline{a}, \underline{b}, \bullet) \,, \tag{3.63}$$

Here AB and BC denote the topological functions, of incident elements. It can be seen that for all discretization schemes coupling functions K and K' are required which yield coupling coefficients that finally are entered into the system matrix of the final equation system.

Discr. Scheme	Neighborhood	Neighb. Elem.
FEM (linear)	$\bigcup_{VC}$	$[\bigcup_{VC} \bigcup_{CV}]$
FVM (linear)	$\bigcup_{VE}$	$[\bigcup_{VE} \bigcup_{EV}]$
BEM (linear)	The cell complex $\mathcal{B}$	all vertices $\bigcup \mathcal{B}V$
FDM (linear)	$\bigcup_{VN}$	$\bigcup_{VNV}$

For finite elements the neighborhood consists of the cells which are incident to the initial element. For finite volumes the neighborhood comprises all edges which are incident with the initial vertex. Boundary elements have a global neighborhood, i.e. each vertex is in the neighborhood of any other vertex. Finite differences do not explicitly give a neighborhood definition. It can be either derived from topological or from geometrical features.

Chapter 4

Algebraic Systems

In Chapter 3 the most commonly used discretization schemes were presented using the topological formulation derived in Chapter 2. Further steps have to be accomplished in order to obtain an equation system and consequently an appropriate solution.

Firstly, it has to be remarked that the given formulation of the discretized differential equations only concerns the residua of the respective discretized operations and not the dependences of the unknown variables as required for linear equation systems. For this reason a data structure which is compatible with the specification methods of Section 2.5 is introduced, which stores values as well as the linearized dependences for given variables. A method is shown which automatically derives the linear dependences on the unknown variables and assembles the linearized equations line-wise.

One commonly used method which is especially employed for finite elements, namely the assembly via local shape matrices, is discussed and compared to the line-wise assembly method. Furthermore, difficulties and advantages of these methods are compared to line-wise assembly with a focus on boundaries and interface conditions.

In most cases methods for the solution of nonlinear equation systems rely on linear or linearized equation systems. These systems result from the nonlinear system by a linearization (first order multi-variable Taylor approximation) around a given vector $\mathbf{x}_0$ of linearization. Thus, for each variable x_i a value for the linearization is defined. In the neighborhood of the linearization vector, the respective function can be approximated via the linear (affine) function $f(\mathbf{x}_0) + \langle \mathbf{c}, \mathbf{x} \rangle$ using an appropriately chosen initial vector $\mathbf{x}_0$.

In order to obtain a solution, the resulting equation system has to be solved. This means that the equations have to be assembled into a system matrix and/or a right-hand-side vector.

4.1 Line-Wise Assembly

The assembly method which was implicitly presented in Chapter 2 was line-wise assembly, where one equation (e.g. setting zero a residuum) relates to one matrix line and one or more right-hand-side vector entries. In the following it is assumed that only one right-hand-side vector is available, the generalization to more right-hand-side vectors can be achieved easily. For eigenvalue problems only the matrices can be specified.

4.1.1 Algebra of Linearized Equations

One equation, comprising a matrix line and one right-hand-side vector entry is considered a mathematical entity. It can be viewed as an equation which depends on a number of unknown variables on which the equation linearly depends.

$$\sum_i c_i \cdot x_i = r_0 \ , \tag{4.1}$$

where r_0 denotes the right hand side entry, c_i denotes the coefficients with which the solution variables x_i are weighted. Alternatively, the entity can be seen as residuum R that implicitly has to be zero.

$$R = \sum_i c_i \cdot x_i - r_0 (= 0) \tag{4.2}$$

This view is perhaps more illustrative, because addition, multiplication, function application, and so forth of residual expression seems to be more natural than the respective application of operations on equations. If the operations are applied to residual expressions, it is always assumed that the expression is followed by a $(= 0)$. For the sake of simplicity, the matrix line can be seen as line vector and can be written in a matrix formalism:

$$R = \langle \mathbf{c}, \mathbf{x} \rangle - r_0 (= 0) \ , \tag{4.3}$$

where $\mathbf{c}$ denotes the vector of coefficients c_i and and $\mathbf{x}$ denotes the solution variables x_i. Furthermore, the operations used have to provide an algebraic structure. The first and most important requirement on such a structure is closedness. This means, that the result of applying operations on one or more expressions still remains an expression of the same type. Note, if linear (residual) expressions depending on one or more variables are applied to a function the result is still a linear expression.

It can be seen easily that the application of arbitrary functions on linear expressions does not necessarily lead to linearized expressions. Only very few operations (i.e. linear operations) preserve the algebraic structure of a linear equation. This especially holds true for addition and scalar multiplication by which an affine linear algebra is created. Of course, in general, especially when solving non-linear equations such an algebraic structure cannot be preserved.

This problem can be easily fixed, if after each nonlinear operation a subsequent linearization step is performed. It can be shown easily that the number of intermediate linearization steps is not relevant as long as removable discontinuities are avoided.

In the following sections the basic operations are shown. As an example, the method is demonstrated on a simple nonlinear discretized differential equation system. Afterwards the assembly of the system matrix is shown. In a second example, the same calculations are carried out for an eigenvalue equation system.

4.1.2 Constants

One basic element of the algebraic structure a constant linear equation. It has to be noted that a constant symbol without any variable dependence does not have any sense, if interpreted as function, because, if the respective constant is non-zero, the equation always leads to a contradiction, e.g. $R = 3(= 0)$. As a constant which is added to a linear or linearized residuum gives an expression which is sensible e.g. $R_1 + R_2 = 2 + x_1(= 0)$, where it is possible to retrieve a proper solution. With respect to the matrix formulation, an additive constant expression influences the right hand side entry of the respective line or the equation system.

4.1.3 Basic Operations

Firstly, basic linear operations such as addition and inversion are discussed. These operations directly lead to linear expressions and do not require subsequent linearization. According to the linearity of the operations, addition, subtraction, and negation are applied in the following manner:

$$(\langle \mathbf{c_1}, \mathbf{x} \rangle + r_1) + (\langle \mathbf{c_2}, \mathbf{x} \rangle + r_2) = \langle \mathbf{c_1} + \mathbf{c_2}, \mathbf{x} \rangle + (r_1 + r_2) \tag{4.4}$$

$$(\langle \mathbf{c_1}, \mathbf{x} \rangle + r_1) - (\langle \mathbf{c_2}, \mathbf{x} \rangle - r_2) = \langle \mathbf{c_1} - \mathbf{c_2}, \mathbf{x} \rangle + (r_1 + r_2) \tag{4.5}$$

$$-(\langle \mathbf{c}, \mathbf{x} \rangle + r) = \langle -\mathbf{c}, \mathbf{x} \rangle - r \ . \tag{4.6}$$

If the multiplication of two linearized expressions is performed, higher order terms consisting of bilinear expressions are neglected and truncated.

$$(\langle \mathbf{a}, \mathbf{x} \rangle + r_a) \cdot (\langle \mathbf{b}, \mathbf{x} \rangle + r_b) = r_a \cdot r_b + r_a \langle \mathbf{b}, \mathbf{x} \rangle + r_b \langle \mathbf{a}, \mathbf{v} \rangle + \underbrace{\langle \mathbf{b}, \mathbf{x} \rangle \langle \mathbf{a}, \mathbf{x} \rangle}_{\hookrightarrow 0} =$$

$$r_a \cdot r_b + \langle r_a \cdot \mathbf{b}, \mathbf{x} \rangle + \langle r_b \cdot \mathbf{a}, \mathbf{x} \rangle \tag{4.7}$$

Functions can be applied to a linearized expression as follows. First, the function as well as its derivative have to be known. This function is denoted as f, its derivative is referred to as f'. The application of the function on the linearized expression yields

$$f(\langle \mathbf{c}, \mathbf{x} \rangle + r) = f(r) + \langle f'(r) \cdot \mathbf{c}, \mathbf{x} \rangle. \tag{4.8}$$

The verification can be easily performed by using the chain rule of differentiation. Furthermore, it has to be mentioned that division can be considered a binary function, the application of which on linearized expressions is straight forward. However, it has to be mentioned, that the occurrence of solution variables as divisors is often avoided by proper multiplication.

In the following a linearized expression will be written shortly as follows:

$$\langle \mathbf{c}, \mathbf{x} \rangle + r =: [c_1, c_2, \ldots, c_n; r] \tag{4.9}$$

Accordingly, the rules for addition, multiplication, and function application are written as

$$[c_1, \ldots; r] + [d_1, \ldots; s] = [c_1 + d_1, \ldots, r + s]$$

$$[c_1, \ldots; r] \cdot [d_1, \ldots; s] = [s \cdot c_1 + r \cdot d_1, \ldots, r \cdot s]$$

$$f([c_1, \ldots; r]) = [f'(r) \cdot c_1, \ldots, f(r)] \tag{4.10}$$

If two linearized expressions are divided, the case might occur that both, numerator and denominator are identically zero and Bernoulli's (del Hospital's) rule has to be applied to obtain the correct result. This is numerically unstable and accordingly leads to various problems regarding the evaluation of the quotient rule of differentiation. Inserting the values leads to divisions by zero or – even worse – to a division by a very small floating point number. Furthermore, for the correct evaluation of the fully linearized result comprising the coefficients for the variables, higher order terms (which are not available then) of the linearized expressions has to be considered.

For this reason, a critical function with a removable discontinuity, for instance $f(x) = \sin(x)/x$ is implemented with a Taylor series expansion in order to avoid the division. The application of the function f does not cause problems because the function as well as its derivative are continuous and can be determined in a straight forward manner.

4.1.4 Linear Expressions and Functional Description

In the following sections the linear dependence of equations on different solution variables x_i is discussed. Typically, a solution variable is defined as a quantity on the underlying cell complex. Furthermore, each quantity value that is a solution value requires to be assigned a definite position i in the solution vector.

In order to determine the position of the quantity associated with a given topological element $\mathbf{v}$, an index function $i(\mathbf{v})$ is introduced. If more solution quantities are required, the function defermining the position of the solution within the vector can be obtained by different index functions (i_n and i_ψ for the quantities n and ψ).

In the following the residual expressions of discretized differential equations are formulated with linearized expressions. A residual expression is formulated and defines a dependence between single quantity values

$$R = f(q(\mathbf{v}_1), q(\mathbf{v}_2))(= 0) \ . \tag{4.11}$$

Initially the quantity q may have any value and can not be neglected. The equation system may depend on the quantity q which is evaluated on many different topological elements. For each of these elements an index function i is available that assigns each topological element a position in the solution vector. The formulation of the residual equation (4.11) can be written as follows:

$$R(\mathbf{v}) = q(\mathbf{v}) + 1 \cdot x_{(i(\mathbf{v}))} \ . \tag{4.12}$$

This expression can be written using the lin() function as

$$R(\mathbf{v}) = [\mathrm{lin}(q, i)](\mathbf{v}) \ \ , \tag{4.13}$$

where lin() is defined in the following way:

$$Q := \mathrm{lin}(q, j) = q + x_j = [\underbrace{\ldots, 1, \ldots}_{\text{j−th position}}; q] \ . \tag{4.14}$$

In the following examples, the residual expressions q are replaced by their linearized analoga Q, which implies that each quantity is added an increment x_j, where j is the position, given by the index function i. This function i represents the position of the matrix column which is relevant for the quantity q on the given element. If the residual equations are given in this manner, the solution consists of a vector of solution variables x_j which are added to the quantities q in order to obtain the final solution.

4.1.5 Linear Example - Poisson Equation

This example shows the application of the Poisson equation in a thermodynamic simulation. The equation system consists of four points from which two are boundary points

Figure 4.1: Simple one-dimensional simulation domain comprising four points. On the two boundary points $\mathbf{v}_1, \mathbf{v}_4$ an (implicit) Dirichlet boundary conditions is applied. On the two interior points $\mathbf{v}_2, \mathbf{v}_3$ a discretized differential equation is applied.

with homogeneous Dirichlet boundary conditions. Furthermore a constant right hand source term is given which equals unity. The four points are given on a straight line in equal distances according to Figure 4.1. The governing equation yields $\Delta T = 1$ for the interior points and $T = 0$ for the boundary points.

In the equation system two unknown variables are defined, namely temperature values for the two inner vertices. The discretization formula resulting from finite differences for the vertices yields

$$R = [\sum_{VNV}^{\underline{w}} QK(\bullet, \underline{w}) - 1] \tag{4.15}$$

The coupling terms K yield -2, if the arguments are identical, unity otherwise. Setting the temperature identically zero at the boundary points leads to the following simplification of the equations

$$\begin{aligned} R(\mathbf{v}_1) &:= -2Q(\mathbf{v}_1) + Q(\mathbf{v}_2) - 1 \\ R(\mathbf{v}_2) &:= -2Q(\mathbf{v}_2) + Q(\mathbf{v}_1) - 1 \ . \end{aligned} \tag{4.16}$$

The expression for the solution quantity is now replaced by a linearized expression of the following form:

$$\begin{aligned} Q(\mathbf{v}_1) &= [1, 0; q(\mathbf{v}_1)] \\ Q(\mathbf{v}_2) &= [0, 1; q(\mathbf{v}_2)] \ . \end{aligned} \tag{4.17}$$

Before the linear equation system is solved, the quantity is set to an arbitrary value, for instance zero. For the equations (4.16) one obtains

$$R(\mathbf{v}_1) = -2Q(\mathbf{v}_1) + Q(\mathbf{v}_2) - 1 = [-2, 1; -2q(\mathbf{v}_1) + q(\mathbf{v}_2) - 1]$$

$$R(\mathbf{v}_2) = -2Q(\mathbf{v}_2) + Q(\mathbf{v}_1) - 1 = [1, -2; -2q(\mathbf{v}_2) + q(\mathbf{v}_1) - 1] \tag{4.18}$$

It can be seen that the two linear expressions $R(\mathbf{v}_1)$ and $R(\mathbf{v}_2)$ contain the information of the system matrix. The assembly of the matrix from the linear expressions is straight forward, once the linear expressions are obtained.

The solution vector $\mathbf{x}$ contains the single solution elements x_i. These elements are applied to the quantities according to (4.12). This means, that the solution x_i is added to the respective solution quantity q. If the initial value of the quantity is zero, the solution quantity x_i has the same value of the solution of the initially given problem. Otherwise, the solution contains an update vector which has to be added in order to obtain the correct solution. This is equivalent to the Newton method, however, since the equation is linear, the solution is obtained in a single step.

4.1.6 Example - Nonlinear Conductivity

In the following section it is assumed that the thermal capacitance ρ depends on the temperature of the respective material. In analogy a nonlinear dependence of the thermal conductance κ with respect to the temperature can be specified. However, for the sake of simplicity the heat conductance κ is assumed to be constant in the following example.

$$\operatorname{div} \kappa \operatorname{grad} T = \rho(T) \; . \tag{4.19}$$

A finite volume discretization yields for the same four-point geometry as in the linear example of the last section:

$$R := \sum_{VNV}^{\underline{v}} \kappa Q K(\bullet, \underline{v}) - \rho(Q) \tag{4.20}$$

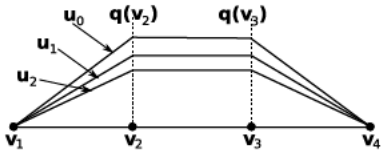

Figure 4.2: The solutions of the linearized problems converge to the solution of the nonlinear problem using the Newton method.

By inserting the neighborhood information and setting the thermal generation term $q = 1$ and $\rho(q) = q^2$, the following expression is obtained.

$$\begin{aligned} R(\mathbf{v}_1) &:= -2Q(\mathbf{v}_1) + Q(\mathbf{v}_2) - \rho_0 Q(\mathbf{v}_1)^2 \\ R(\mathbf{v}_2) &:= -2Q(\mathbf{v}_2) + Q(\mathbf{v}_1) - \rho_0 (\mathbf{v}_2)^2 \end{aligned} \tag{4.21}$$

When using the Newton method it is always assumed that a given (already good) result is used to obtain a better result (see Fig. 4.2). For the linear problem, a Newton scheme leads to a solution within one single step, because all higher-order terms vanish. For this reason in the linear case the choice of an initial guess does not play a role.
For the nonlinear case, it is however important to chose a proper initial guess. In the following, the initial values $q(\mathbf{v}_1) = q(\mathbf{v}_2) = -0.2$ is chosen. Therefore, the expressions $Q(\mathbf{v}_1)$ and $Q(\mathbf{v}_2)$ yield

$$\begin{aligned} Q(\mathbf{v}_1) &:= [1, 0; -0.2] \\ Q(\mathbf{v}_2) &:= [0, 1; -0.2] \end{aligned} \tag{4.22}$$

First, the application of the $\rho_0 Q(\mathbf{v}_1) \cdot Q(\mathbf{v}_1)$ is performed as follows

$$\begin{aligned} Q(\mathbf{v}_1) \cdot T_1 &= [1, 0; -0.2] \cdot [1, 0; -0.2] = \\ Q(\mathbf{v}_1) \cdot [-0.4, 0; 0.04] &= \cdot [-0.4, 0; 0.04] \end{aligned} \tag{4.23}$$

Repeated application of this step (eventually) leads to a solution of this equation system. However, it has to be considered that nonlinear methods have different rates of

convergence and sometimes solutions can not be found at all [82]. Nevertheless, the presented method with linearized expressions can be used for the formulation of all kinds of nonlinear problems, the solution of these problems can be treated separately from the assembly.

4.1.7 Derivative Considerations

The discussed equations form a linear equation system whose solution vector gives an increment or update for the respective solution quantities. In the Newton scheme an equation system $\mathcal{E}$ is linearized in the neighborhood of a vector $\mathbf{x}_0$ of quantity values for q.

$$\mathcal{E}(\mathbf{x}) = \underbrace{\mathcal{E}(\mathbf{x}_0)}_{\mathbf{b}} + \underbrace{\mathcal{E}'(\mathbf{x}_0)}_{\mathbf{A}}(\mathbf{x} - \mathbf{x}_0) + \mathbf{O}(\|\mathbf{x} - \mathbf{x}_0\|^2) \,, \tag{4.24}$$

where $\mathbf{A}$ denotes the system matrix and $\mathbf{b}$ denotes the right hand side vector. All terms of higher than second order are neglected. An extension of the linearized expressions might also contain second order expressions, however, the memory consumption is raised from N to N^2 in the worst case, where N is the number of unknown variables.

For the first order case, the evaluation of the residuum in the respective linearization vector $\mathbf{x}_0$ results in the (inverse) right hand side. The evaluation of the derivatives $\partial_{x_i}\mathcal{E}_j$ in the linearization vector $\mathbf{x}_0$ yields the entries of the system matrix.

From this point of view, it can be stated that forming the derivatives from the discretized expression is performed implicitly when using linearized expressions. This is of course evident, when the Taylor expansion is considered, however, a lot of work for the differentiation of the respective equations can be saved. It also has to be stated that using this method for forming the derivatives only determines the derivative only for a given linearization vector $\mathbf{x}_0$ and not an analytic expression. This implies that such a method is not apt for the differentiation of general algebraic formulae, however, it performs well for the specific matter.

4.1.8 Refinement of Models

A method which is typically used in scientific computing is the refinement of formulae which have been proven to work. As an example one begins with a linear differential equation for instance the Poisson equation. Then one adds the isothermal drift diffusion current relations and performs the necessary testing. Afterwards, the temperature is added using another solution quantity and the non-isothermal drift-diffusion model is implemented.

In all stages of the implementation additional derivatives of residua with respect to other values have to be considered. If many different solution variables are involved, this requires an effort which approximately depends quadratically on the number of the solution variables. Even if done aided by computational algebra tools, this is very cumbersome and often results in oversights and flaws which are difficult to find.

The use of linearized equations offers the advantage that all derivatives are implicitly calculated and adding further solution functions does not imply further efforts, except the formulation of the residuum of the new governing equation. All derivatives with respect to the other variables are implicitly determined.

4.1.9 Eigenvalue Problems

The use of linearized expressions can also be generalized so as to be employed for the specification of eigenvalue equations. In this case, expressions of the form $\mathbf{Ax} = \lambda\mathbf{Bx}$ the linearized expression can be specified. The Schrödinger equation which is typically used for the solution of quantum mechanical problems is a basic example for the treatment of eigenvalue equations using linear expressions.

$$\Delta\psi + V\psi = \lambda\psi \; . \tag{4.25}$$

The eigenvalue equation system can be written in the following line-wise form:

$$\mathcal{E}_k = \sum_i c_{i,k} v_i + \lambda \sum_j d_{j,k} v_j (= 0) \; . \tag{4.26}$$

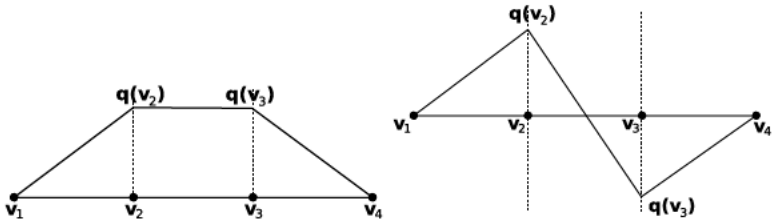

Figure 4.3: The eigenvalue problem is specified on a four point one-dimensional equidistant domain, where the two boundary points are set to zero. Two eigenvectors are shown.

For the sake of simplicity and in order to obtain a consistent short and concise notation comparable to linearized expressions (4.9), the following scheme is introduced:

$$\sum_i c_i v_i + \lambda \sum_j d_j v_j =: [c_1, \ldots, c_N; d_1, \ldots, d_N] \tag{4.27}$$

As these eigenvalue equations are always linear, only linear operations have to be considered and a linear space is obtained.

$$[c_1, \ldots, c_N; d_1, \ldots, d_N] + [e_1, \ldots, e_N; f_1, \ldots, f_N] =$$

$$[c_1 + e_1, \ldots, c_N + e_N; d_1 + f_1, \ldots, d_N + f_N] \tag{4.28}$$

$$[c_1, \ldots, c_N; d_1, \ldots, d_N] - [e_1, \ldots, e_N; f_1, \ldots, f_N] =$$

$$[c_1 - e_1, \ldots, c_N - e_N; d_1 - f_1, \ldots, d_N - f_N] \tag{4.29}$$

$$a \cdot [c_1, \ldots, c_N; d_1, \ldots, d_N] =$$

$$[a \cdot c_1, \ldots, a \cdot c_N; a \cdot d_1, \ldots, a \cdot d_N] \tag{4.30}$$

$$\lambda([c_1, \ldots, c_N; 0, \ldots, 0]) = [0, \ldots, 0; c_1, \ldots, c_N] \tag{4.31}$$

An algebraic structure which introduces linear(ized) expressions to eigenvalue- expressions has to provide the following elements:

$$\hom([c_1, \ldots, c_N; q]) = [c_1, \ldots, c_N; 0, \ldots, 0] \tag{4.32}$$

$$\lambda([c_1, \ldots, c_N; q]) = [0, \ldots, 0; c_1, \ldots, c_N] \tag{4.33}$$

The following discretization is obtained in a one-dimensional simulation domain according to Figure 4.1 comprising four vertices when using finite differences. The solution is referred to as ψ which is stored in the quantity q. In analogy to (4.12), the term Q is introduced.

$$\begin{aligned} R(\mathbf{v}_1) &= -2Q(\mathbf{v}_1) + Q(\mathbf{v}_2) + V(\mathbf{v}_1)Q(\mathbf{v}_1) - \lambda Q(\mathbf{v}_1) = 0 \\ R(\mathbf{v}_2) &= -2Q(\mathbf{v}_2) + Q(\mathbf{v}_1) + V(\mathbf{v}_2)Q(\mathbf{v}_2) - \lambda Q(\mathbf{v}_2) = 0\ . \end{aligned} \tag{4.34}$$

The potential is assigned constant values $V_1 = V_2 = 1$. For the probability function ψ_i the following term is inserted

$$\begin{aligned} Q(\mathbf{v}_1) &= [1, 0; 0, 0] \\ Q(\mathbf{v}_2) &= [0, 1; 0, 0]\ . \end{aligned} \tag{4.35}$$

Accordingly, one obtains the following eigenvalue expressions:

$$\begin{aligned} R(\mathbf{v}_1) &= -2[1, 0; 0, 0] + [0, 1; 0, 0] + [1, 0; 0, 0] - \lambda[1, 0; 0, 0] \\ R(\mathbf{v}_1) &= [-1, 1; 1, 0] \\ R(\mathbf{v}_2) &= [1, -1; 0, 1] \end{aligned} \tag{4.36}$$

In analogy to linearized expressions, linear eigenvalue expressions can be used in order to fill the eigenvalue matrices. One often observes that the matrix B is the unit matrix. Often it is necessary to change the order of the equations in order to maintain the unit matrix for B.

4.2 Element-Wise Assembly

An alternative to the line-wise assembly as shown above is the assembly via small local sub-matrices or stencil matrices. The major advantage of this method over the line-wise assembly is that normally only one incidence relation is required. The method of element-wise assembly typically used for finite element and finite volume simulation tools.

In contrast to the line-wise assembly, element-wise assembly considers the influence of the topological elements of highest order used in the topological neighborhood, mostly cells,

on the final equation system. If used for the solution of a differential equation with one solution function, each cell influences a set of $n \times n$ matrix entries where n is the number of incident of the element.

4.2.1 Comparison to Line-Wise Assembly

In many cases element-wise assembly requires less numerical costs to assemble, because cell-related quantities can be calculated together and need not be re-calculated in each step. However, most of the cell-wise calculations can be stored into the cells as a pre-processing step.

Another feature which makes the element-wise assembly faster than line-wise assembly is that the number of operations which have to be performed for a cell is better to predetermine, because cells usually have the same topological shape throughout the simulation domain. If the number of operations is constant, the compiler can perform various optimizations, e.g. perform parallel execution or handle this as a loop of fixed iterations.

For the parallel treatment of big equation systems the parallel assembly can be easier performed via line-wise application, because matrix insertions are performed line-wise and therefore no concurrent access methods are required.

Compared to element-wise assembly, line-wise assembly imposes higher requirements on the underlying data structures, while the matrix assembly is straight forward. For the implementation typically two traversal functions have to be available, for instance a function that yields all vertices incident to a given cell and another function that yields all cells incident to a given vertex. The assembly of the system matrix is rather simple, because the system matrix is separated into disjoint regions, namely lines, where each line can be assembled separately, especially using parallel computing mechanisms.

In contrast, element-wise assembly only requires one traversal function which provides all elements incident subsets of a given element, for instance the set of all vertices which are incident to a given cell. Therefore requires the possibility of inserting element submatrices or local matrices into the system matrix. In this case it is possible that two local matrices overlap. When using parallel processes for the assembly it is possible that

two or more processes require access to the same matrix element which imposes various difficulties of synchronization and ressource management.

These different models of assembly lead to completely different matrix interfaces: Whereas for line-wise assembly it is only required to access each non-zero matrix element at most once, the element-wise access requires different operations, even deleting single matrix rows [83]. Other solver interfaces require to add single matrix elements into the system matrix, where each matrix element can be accessed for several times [84, 36, 85]. For the sake of completeness, it has to be said, that the frameworks investigated for element-wise assembly do not require the topological operations which are needed for line-wise assembly. Figure 4.4 shows the application of line-wise assembly for a finite volume method with an initial vertex $\mathbf{v}$ and five neighboring vertices $\mathbf{w}$. The assembly method yields one matrix line which is directly inserted into the system matrix.

Linear solver environments such as Trilinos [4] and PetSC [48] support element-wise access to single matrix entries. A matrix entry can either be added a given value or is overwritten. Furthermore, it is possible to insert local sub-matrices for optimized finite element assembly.

4.2.2 Applications on Finite Element Schemes

For finite elements the element-wise assembly is carried out by the quadrature of all integral expressions which are related to a common cell $\mathbf{c}$. In the case of a triangular cell which contains three vertex values, nine integrals have to be calculated. When Galerkin schemes are used, the calculation can be reduced to six integrals, because of the symmetry of the local matrix. The matrix can be written in the following manner, where the vertices of the cell $\mathbf{c}$ are denoted as $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$

$$\mathbf{K} = \begin{bmatrix} K(\mathbf{v}_1, \mathbf{c}, \mathbf{v}_1) & K(\mathbf{v}_1, \mathbf{c}, \mathbf{v}_2) & K(\mathbf{v}_1, \mathbf{c}, \mathbf{v}_3) \\ K(\mathbf{v}_2, \mathbf{c}, \mathbf{v}_1) & K(\mathbf{v}_2, \mathbf{c}, \mathbf{v}_2) & K(\mathbf{v}_2, \mathbf{c}, \mathbf{v}_3) \\ K(\mathbf{v}_3, \mathbf{c}, \mathbf{v}_1) & K(\mathbf{v}_3, \mathbf{c}, \mathbf{v}_2) & K(\mathbf{v}_3, \mathbf{c}, \mathbf{v}_3) \end{bmatrix}$$

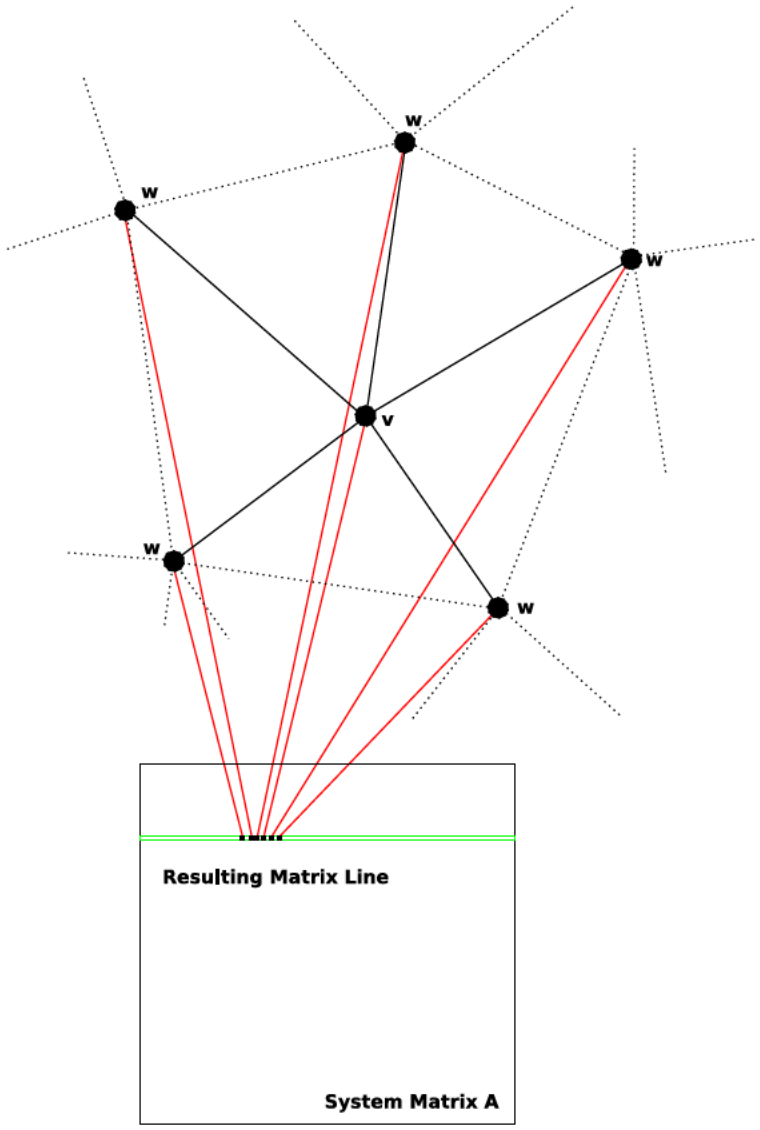

Figure 4.4: Line-wise assembly. A finite volume method with an initial vertex v and five neighboring vertices w in the underlying topological structure is used to assembly an algebraic equation. The assembly method yields one matrix line which is directly inserted into the system matrix.

In the notation according to Chapter 2, such a local element matrix can be assembled using the vectorization command $\bigsqcup$ in order to retrieve a matrix.

$$\mathbf{K}(\mathbf{c}) = \bigsqcup_{CV}^{\underline{c}} \bigsqcup_{CV_{\underline{c}}}^{\underline{v}} K(\bullet, \underline{c}, \underline{v}) \tag{4.37}$$

As can be seen in Figure 4.5, the matrix is assembled by local element matrices which are added to the system matrix. With the formalism introduced in Chapter 2 the local system matrix can be formulated.

4.3 Boundaries and Interfaces

From the line-wise point of view it is easy to define an equation for a boundary vertex. If, for instance, Dirichlet boundary conditions are employed, a respective residuum is formulated and a new equation for the vertex is inserted into the system matrix.

Especially element-wise formulation methods have difficulties with the specification of boundary conditions, because a special treatment for a boundary equation in a distinct vertex has to be considered in the calculation of all sub-matrices which are incident to the given vertex.

As a consequence, many equations for boundary points are treated algebraically which causes severe difficulties when introducing non-conventional boundary conditions, interface conditions, or conditions at triple points.

Every differential equation has to be specified appropriate boundary conditions in order to yield a valid solution. Without specifying a boundary condition, differential equations do not deliver a unique solution but offer a large function space of possible solutions.

In some cases it can be appropriate to have different sections, where interface conditions are given. This often turns out to be problematic for element-wise implementation approaches and various efforts have been spent on workarounds to handle this problem [83].

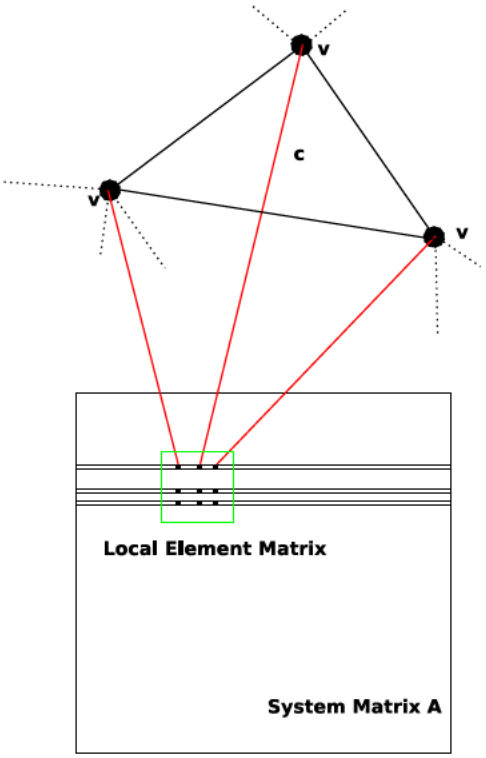

Figure 4.5: Element-wise assembly. For the finite element method all coupling coefficients which are determined by integrals over a common cell **c** are assembled together.

4.3.1 Boundary Conditions

Boundary conditions typically set the function at the boundary (Dirichlet boundary condition) or its normal derivative (Neumann boundary condition) to a certain value . For differential equations of higher than second-order, e.g. the BiLaplace equation $\Delta\Delta u = 0$, higher order derivatives can be given.

With an identical formulation but with a modified functional meaning, the discretized differential equation can be applied to a boundary vertex or another boundary element. Using finite elements for the Poisson equation, the application of the discretization formula for the interior to a boundary point implicitly yields zero Neumann boundary conditions. This property of finite element methods is called *natural boundary condition*. The same holds true for the discretization of the Poisson equation using finite volume schemes.

In combination with the boundary conditions some problems occur which can be handled using the method of line-wise assembly rather than the method of element-wise assembly. For the application of boundary conditions two different methods are available. The first and easiest method is to assign the respective function which is collocated with a boundary point a given value $q_B(\mathbf{v})$. In line-wise assembly, this can be carried out easily by inserting a (trivial) boundary expression or equation

$$R(\mathbf{v}) := q(\mathbf{v}) - q_B(\mathbf{v}) = [\ldots, 1, \ldots; -q_B(\mathbf{v})](= 0) \; . \tag{4.38}$$

In order to reduce the numerical effort, simple boundary conditions such as zero Neumann and Dirichlet conditions are eliminated and directly inserted in the interior equations. In simple cases this can speed up the calculations and reduce the size required for storing matrix coefficients.

For Dirichlet boundary conditions the solution variables $x_{i(\mathbf{v})}$ for boundary points are replaced by the boundary value $q_B(\mathbf{v})$ so that $x(\mathbf{v}) \rightarrow [0, \ldots; q_B(\mathbf{v})]$. Other boundary conditions can only be eliminated in special cases, for instance, if natural boundary conditions are applied.

Finite differences allow to eliminate all boundary conditions. These conditions are derived using the same scheme as shown in Section 3.4.

$$\mathcal{M}(u) = \alpha u + \beta \partial_n u + \gamma = 0 \tag{4.39}$$

A (linearized) expression in the following manner is retrieved, where for each boundary vertex $\mathbf{v}_B$ exactly one interior vertex $\mathbf{v}_I = BI(\mathbf{v}_B)$ can be obtained. The boundary functional $\mathcal{M}$ is specified by the following equation:

$$R(\mathbf{v}_B) = k(\alpha, \beta)Q + \gamma + k'(\alpha, \beta)Q[BI] = 0 \ , \tag{4.40}$$

where the affine residual expression R for the boundary condition contains the boundary variable as well as unknown variables from the interior of the equation system. For the determination of the normal derivative at the boundary no other boundary vertices are required. One obtains

$$x_{i(\mathbf{v}_B)} = -\frac{k'Q[BI] + \gamma}{k} \ , \tag{4.41}$$

The Dirichlet boundary condition and the natural boundary condition are most simple to treat, because either a point can be eliminated or no special treatment has to be performed at all.

In many cases, such as in device simulation, a constant flux or current is required through a certain boundary region. In such a case it is assumed that the connection between the circuit and the device is ideal (i.e. all contacts are at the same potential). In order to obtain the boundary equations for all elements (in general vertices), first all elements at the boundary have the same value - in this case - for the potential. From the set of boundary conditions one element is chosen. This can be accomplished by a topological function that assigns each vertex $\mathbf{v}$ a vertex $V\mathcal{B}V(\mathbf{v}) = \mathbf{v}_F$. For all boundary vertices $\mathbf{v}$ but one first vertex $\mathbf{v}_F$ the following equation is given:

$$R = Q - Q[V\mathcal{B}V] \tag{4.42}$$

For the boundary vertex $\mathbf{v}_F$ the flux boundary condition is given. In finite volumes, this can be specified in the following manner:

$$R(\mathbf{v}_F) = [-\Phi_B + \sum_{\mathcal{B}V} \sum_{VE} \Phi](\mathbf{v}_F) = 0 \ , \tag{4.43}$$

where $\mathcal{B}$ is the respective boundary region and the function $\mathcal{B}V$ determines all vertices incident to the boundary region $\mathcal{B}$. Φ_B denotes the given current which flows through the boundary region.

When assembling this system, the variables corresponding to the boundary vertices can be reduced to one variable which contains the potential of the boundary vertex $\mathbf{v}_F$ (and of course the potential of all other points).

4.3.2 Interfaces and Triple-Points

Interfaces and especially triple points, namely points which are at the intersection of three or more regions of different governing equations are problematic for many programming environments. Problems occur when the simulation domain is initially segmented and the equations are assembled for the single topological elements for each segment separately. In such a case it is very cumbersome to fill in the respective matrix entries into a common system matrix and manage the couplings between single interface points of different segments which coincide. In [83] methods are shown which are numerically stable for combining two system matrices each of which is the result of discretizing segments with a common boundary.

The line-wise assembly method using underlying segmentation and incidence information provides proper means for the specification of the governing equations of an interface point or a point incident to three segments, namely a triple point. For instance, for a triple point the finite volume discretization the divergence of the flux Φ is set zero.

$$R = \sum_{V\mathcal{S}}^{\underline{v}} \varepsilon \sum_{VE(\underline{v})} \Phi \tag{4.44}$$

The simulation domain is segmented into three or more segments $\mathcal{S}$. Three segments incide in the triplepoint-vertex. In the different segments different permittivities $\varepsilon(S)$ are given. The flux Φ is defined on an edge and describes the edge parallel component of a given vector field multiplied with the respective boundary area.

4.4 Solution of Algebraic Problems

This section gives a brief overview of methods for the solution of linear equation systems, nonlinear equation systems, and eigenvalue problems, as they occur as the result of a discretization scheme. The intention of this section is to show the common features which are required for the solution of these algebraic problems. Firstly, a problem regarding the numbering of the single elements and equations is discussed.

4.4.1 Numbering of Elements

The association of the elements with actual positions in the system matrix is quite informal. A numbering of the elements may be given as the result of data structural treatment, e.g. in order to discriminate elements, however, this shall not be used in order to associate the single elements with a matrix row. The same shall also be avoided for the association with a residual expression assembled with respect to a topological element and the respective matrix line.

However, for the sake of simplicity, it can be arranged that the row association for a quantity can be identical with the line association of the residual expression with respect to the topological element. The element-wise assembly, e.g. of finite elements, implicitly forces this identity of row-association and line-association, because otherwise the local element matrix can not be entered into the system matrix appropriately.

Many solvers for linear equation systems or eigenvalue systems have their own optimized matrix restructuring mechanisms. Even though there are methods which assign elements a certain line or row within the respective system matrix, e.g. the Cuthill McKee algorithm [86], the use of multiple quantities on the same simulation domain makes the vertex ordering mechanism difficult and an appropriate association of a vertex with a matrix row or line can not be given. Furthermore, the Cuthill McKee algorithm is intended for the use of vertices only. Although the topological structure can be written as incidence graph, the application of such an algorithm becomes quite cumbersome. Similar approaches, however, can be used at a matrix level to order the equations as well as the unknown variables.

Once an each matrix column is associated with a topological element (and a quantity), the elements of the matrix can be inserted. Further optimization mechanisms, for instance the pre-elimination of variables, can be performed as a pre-processing step of the linear/eigenvalue solver mechanism.

4.4.2 Algebraic Equation Systems

There are many algebraic environments or frameworks available [4, 47] which can deal with a large variety of different algebraic problems. These frameworks offer different algorithms for the solution of the same algebraic equation, while maintaining the same data interface for writing as well as retrieving coefficient data. Therefore, it is simple to adapt and optimize the solution procedure to the special algebraic problem and chose parameters in order to obtain an optimal solution behavior. This is especially desirable, when high accuracy results are required or the solution has to be found with resource constraints such as time and memory consumption.

4.4.3 Re-Writing the Solution

After the algebraic solution has been found the solution has to be written back to the quantities. First, the solution vector has to be re-inserted into the solution quantity, in many cases the elements of the solution vector are added to the quantity values. In case of eigenvalue systems, each eigenvalue is assigned one quantity.

For nonlinear systems using a gradient formulation, line-wise assembly methods offer the advantage that the equations have to be specified only once with respect to the (initially preset) solution vector. The solution vector is used as linearization vector for the next linearization process, while the calculation is performed in the same manner based on the values stored in the solution quantity. This quantity is therefore updated non-linear solution method precedes iteratively.

Chapter 5

Software and Implementation

The framework of discretized expressions which have been established in Chapter 2 and applied to differential equations is mainly intended for a simple and concise way of implementation, which is of course required in order to implement such a scheme on a computer. Furthermore, the discrete formulation scheme is also designed to be implementable using the high level programming language C++ [23].

As topological basis of this framework the Generic Scientific Simulation Environment (GSSE) [40, 21, 22] is taken which implements the required topological functionality for the functional structures. It is required for the implementation of the topological functions shown in Section 2.4 as well as for quantity handling. The implementation is based on the cursor/property map concept as shown in [87], where for more complex data structures different iteration mechanisms are provided.

The functional layer introduces a functional programming implementation based on the Phoenix 2 library [29]. This library provides overloading of operators and gives the opportunity to implement higher order functions as shown in Section 2.2. One major advantage of the Phoenix 2 library is the use of explicitly named variables which is required for naming accumulation variables according to Section 2.5. Consequently, the Phoenix 2 library provides appropriate means for the creation of a domain specific embedded language [88] (DSEL) within C++ as shown in Section 5.1. A possible implementation of linearized expressions is shown in Section 5.2 .

5.1 A Language for the Specification of Discretized Expressions

A DSEL is presented which forms a unique mapping for discrete expressions according to Chapter 2 into the C++ programming language. Before details on the actual implementation of the single expressions are given, the notation is explained. This code is valid within the C++ syntax. Using the respective functional library for accessing the topological structure, these expressions serve as basis for the implementation of the respective discretization formulae.

5.1.1 The Equations

For the sake of simplicity the treatment of the Laplace operator using the discretization schemes from Chapter 3 is discussed. For the first implementation only residual expressions are determined, however, the linearized expressions are shown in Setion 5.2. The expressions to be discretized are briefly reviewed:

$$R_{\mathrm{FEM}} := \sum_{VC}^{\underline{v}} \sum_{CV}^{\underline{c}} qK(\underline{c}, \underline{v}, \bullet) \; , \tag{5.1}$$

$$R_{\mathrm{FVM}} := \sum_{VE}^{\underline{v}} \mathcal{O}(\bullet, \underline{v}) \frac{A}{l} \sum_{EV}^{\underline{e}} \mathcal{O}(\underline{e}, \bullet) q \; , \tag{5.2}$$

$$R_{\mathrm{BEM}} := \sum_{\mathcal{B}V}^{\underline{v}} qK(\underline{v}, \bullet) \; , \tag{5.3}$$

$$R_{\mathrm{FDM}} := \sum_{VNV}^{\underline{v}} q \cdot K(\underline{v}, \bullet) \; . \tag{5.4}$$

In the following implementation the coupling coefficients are determined locally with respect to geometric of physical quantities on the respective topological elements and are denoted as higher order functions `K`. For finite volumes the discretization formula is explicitly written.

```
R_FEM = sum<VC>(_v)[sum<CV>(_c)[ q * K(_c, _v, _1) ]];
R_FVM = sum<VE>(_v)[0(_1, _v)*A/d*sum<EV>(_e)[0(_e, _1) * q]];
R_BEM = sum<CxV>(_v)[q * K(_v, _1)];
R_FDM = sum<VNV>(_v)[q * K(_v, _1)];
```

It can be seen that the summation symbols $\sum$ are replaced by `sum`. The different kinds of traversal methods are written within angles, for instance the vertex on cell function CV is referred to as `<CV>`. The named variables can be directly transformed into the DSEL by replacing $\underline{v}$ by the similar and suggestive symbol `v`. This notation was taken from the Phoenix2 library [29] that intrinsically provides these symbols. Furthermore, the development of Phoenix2 was the main inspiration for choosing this special notation. The `let`-symbol is also taken from the Phoenix2 library and given the symbol $\vee$ within the functional calculus. Even though, the lambda function is also introduced in the Phoenix2 environment and can be directly introduced for creating higher order functions, its use was not required for the specification of discretized differential expressions. Furthermore, its use lead to a more complicated notation and drastically worsened the readability.

Higher order functions such as the quantities as well as the coupling functions can be taken directly from the Phoenix2 library, where a simple interfacing to the underlying topological and quantity structures is possible.

Within the syntax of C++ an own language (DSEL) can be defined with which different expressions can be specified in a straight-forward manner. For the specification of these objects, object generators [88] as well as template expressions [89, 90] are used.

5.1.2 GSSE

A detailed view on the implementation of the topological data structures is given in [40]. The discussed environment comprises a topology library (GTL) that is based on the properties of a single cell and on the properties of a the cell complex. By single cell properties the internal structure of one single cell can be determined. For instance, a triangle consists of three bounding edges and three bounding vertices. Typically, the cell topological properties for all cells are identical within a cell complex. The complex

properties, however, contain the incidence information of topological elements of different cells, for instance the set of all cells incident to a vertex. Once cell and complex properties are obtained, all different traversal mechanisms can be obtained.

Next, methods for the storage of quantities (data access) are introduced. This container provides associative data access with two different keys, namely with topological elements, such as vertices and cells, and with quantity keys. One quantity (section) is defined on various topological elements of the same quantity key. Furthermore, it is possible to access all quantity values which are associated with the same topological element.

Benchmarks of the topological library are given that show that the performance of the provided libraries is in the same order of magnitude as comparable conventionally designed special purpose libraries. A comparison of incidence traversal operations to the boost graph library shows that on most platforms the GTL yields higher performance than an equivalent implementation of the boost graph library. The functional library is compared to the imperative specification of the respective expression. This library is comparable to state of the art highly specialized libraries [40].

5.1.3 Quantity Accessors

In order to introduce the principles of designing classes for the use in combination with the Phoenix 2 library, a quantity accessor which provides access to the underlying quantity is shown. The Phoenix style function classes have to provide an evaluation method `eval`. In this method an environment `env` containing all function arguments (externally written as `_1` ...) as well as all named variables (`_a` ... `_z`) are passed. During the construction of an instance of the class, the topological complex of the class as well as the quantity name is passed. Schematically, the class can be written as follows:

```
template <class Complex>
struct accessor
{
  accessor(Complex & C, Complex::quan_name_t name)
    : C(C), name(name) {}
```

```
   ...

   template<Env>
   eval(Env & env)
   {
      return C.retrieve_quantity(at<0>(env.args), name);
   }
};
```

5.1.4 Accumulation

Defining the cell complex as constant within the class, it is possible to implicitly avoid writing quantity access. In the following the implementation of an accumulation function `sum` using the Phoenix2 library is briefly shown. In this case the function for the evaluation has to be passed a function object containing the summand function. The evaluation function can be written as follows:

```
eval(Env & env, Summand & sum)
{
  base_elem = at<0>(env.args);
  Iterator iter(base_elem);
  result = 0;

  while (iter.valid())
  {
    result += summand.eval(newenv(env, *iter));
    ++iter;
  }
}
```

The first element of the passed environment is used to construct an iterator which traverses, for instance, all incident neighboring elements of a given dimension. For the evaluation of the summand, the value of the iterator has to be passed to the evaluation function. For this reason a new environment comprising arguments and named variables is created, which contains all elements of the old environment as well as the traversed

element which implicitly stands for the first argument. This means that for all function evaluations of the summand the first argument is the element passed.

5.1.5 Application

The application of the discretization statements on the respective elements such as vertices can be written in the following manner:

```
R_FVM = sum<VE>(_v)[O(_1, _v)*A/d*sum<EV>(_e)[O(_e, _1) * q]];

vertex_iterator vit = C.vertex_begin();

while (vit.valid())
{
  R = R_FVM(*vit);
  std::cout << *vit << R << std::endl;
}
```

Using an iterator, all vertices of a given cell complex can be traversed. The traversal loop, terminates because the iterator becomes invalid. The residual expression `R` is calculated for each vertex. Afterwards, the vertex and the respective residuum are listed.

5.2 Linearized Expressions as Data Types

The implementation of linearized expressions as data types can be performed via operator overloading. The class comprises an associative data structure, e.g. a map which is used to assign each solution variable a coefficient. If the matrix is sparse, the use of a map seems to be more appropriate, because only few values have to be stored. It is also possible to integrate line-wise compressed data structures of a matrix data type into the linearized equation interface. If full matrices are used, the appropriate data structure for the linear expression is a simple vector or an array. However, for the following considerations the map related implementation is preferred.

5.2.1 Class Layout

Usually the coefficients are floating point numbers. In order to offer the choice of the data type to the user, templating is used. For this reason, the class can be schematically written in the following way:

```
template<typename NumericT>
class linearized_expression
{
  typedef linearized_expression<NumericT> self;
  typedef std::map<int, NumericT>::iterator map_iter_t;

  std::map<int, NumericT> coefficients;

  std::map<int, NumericT> lambda_coefficients;
    ... or ...
  NumericT right_hand_side;
};
```

This snippet shows the implementation for both, the eigenvalue expression as well as the linear (affine) expression. In one case all λ multiplied variables are stored in a separate map.

5.2.2 Addition

The addition of the maps can be implemented for the affine expression as follows:

```
self operator+=(self& s)
{
  map_iter_t iter = coefficients.begin();

  while (iter != coefficients.end())
   s.coefficients[(*iter).first] += (*iter).second;

   s.right_hand_side += right_hand_side;
}
```

Other operations such as subtraction or multiplication can be implemented in the same manner.

5.2.3 Function Application

For the application of a function on a linear expression, the function as well as its derivative is required. For this reason, the following functional structure is required in order to apply a function on the linear expression:

```
class sine_on_linear_expression
{
  double operator()(double x) const {return sin(x);}

  class derivative
  {
    double operator()(double x) const {return cos(x);}
  }
};
```

It can be seen that the class is written as function object which yields the respective function value and which additionally comprises a nested class called `derivative`. This nested class again is a function object which implements the `operator()`. The application of this operator yields the derivative of the function.

The application of the function to a linear expression can be treated as follows:

```
template<typename Func, typename NumericT>
apply_func(linearized_expression<NumericT> & expr)
{
  linearized_expression<NumericT> result;
  Func f;
  Func::derivative f_;

  double deriv = f_(expr.RHS);
  result.RHS = f(expr.RHS);
```

```
  map_iter_t iter = coefficients.begin();

  while (iter != coefficients.end())
  {
    expr.coefficients[(*iter).first] =
      (*iter).second * deriv;
  }
}
```

Using further beautifications, typically object generators [88], explicit function objects can be constructed from the `sine_on_linear_expression` class so that the application of the expression can be written as follows

```
func<sine_on_linear_expression> Sin;

linearized_expression<double> expr1;
linearized_expression<double> expr2;
...
expr2 = Sin(expr1);
```

5.2.4 Linearized Expressions and Discrete Expressions

The next step is to provide a function for the functional calculus of Section 5.1, where linearized equations are treated. For this reason the function lin() is introduced as higher order function in the context of the proposed DSEL.

In analogy to (5.1) the term `Q` is replaced by the functional expression `lin(q, i)`, where the quantity `i` is the index function (Section 4.1.4), which yields the respective matrix column associated to the solution quantity. All further calculations are performed using the following notation:

```
quantity_accessor i("i");
quantity_accessor q("i");
Q = lin(q, i)
```

```
R_FEM = sum<VC>(_v)[sum<CV>(_c)[ Q * K(_v, _c, _1) ]];

vertex v;

linearized_equation<> eqn = R_FEM(v);
```

It can be seen that the function object `Q` is defined which yields a linearized expression for the solution quantity. Furthermore, the discretized differential operator `R_FEM` is specified as shown in Section 5.1. The function is evaluated on a vertex `v` and yields a linearized equation `eqn`.

This method can be repeated for all vertices and a set of linearized equations is retrieved. Then, each of the linearized expressions is inserted into the system matrix and the equation system is solved. Afterwards the solution is written back to the solution quantity as specified in Section 4.4.3.

5.3 Outlook

Linearized expressions or in analogy the eigenvalue expressions have been assembled and were transferred to the system matrix. In the simplest case an interface provides access to single elements of the system matrix so that all elements, including the right hand side can be copied into the matrix.

It has to be mentioned that copying data which are completely assembled is not efficient, because new memory has to be allocated and time is wasted. For these reasons linear solvers can be designed to provide access to internal data structures which can be directly interfaced by the linearized function.

Consequently, it might be advantageous to use the linearized equation as interface only which is wrapped on each internal matrix data structure. In many cases all relevant data (e.g. which elements are non-zero) are already available in these structures and only few adaptions to the common interface have to be made.

Chapter 6

Summary and Outlook

The main achievements in which the proposed methods differ from state-of-the-art frameworks are, that arbitrary discretization schemes and different kinds of algebraic equations (linear, nonlinear, eigenvalue) can be treated using the same mechanisms.

At the level of discretizing differential equations enormous flexibility can be obtained, because different discretization schemes can be tested at the same time. Furthermore, the formulation of single discretized equations using topological incidence functions for traversing the respective elements makes the formulation independent of the dimension and the archetype of the used elements, even if different mechanisms for the calculation of the geometry related factors have to be used.

At the level of assembly, the use of linearized equations eases filling the matrices and implicitly couples a basis function with a governing equation (which is mainly evaluated in the neighborhood of the local support of the basis function). Furthermore, the use of linearized equations implicitly forms the derivatives required for the use of gradient methods for the solution of nonlinear equation systems. Therefore, gradient based schemes such as the Newton scheme can be implemented straightforwardly.

6.1 Discretization Formalisms

The initial intention on which the discrete calculations of Chapter 2 is based is a common framework for the specification of different discretization schemes. It could be shown in

Chapter 3 that the commonly used differential equations can be discretized using the provided framework of functional expressions.

The basic features of the discrete formalism is the use of accumulation operations. As a consequence, not only local formulae can be evaluated but values within the neighborhood of a given topological element can be accessed and used for the evaluation of the differential operator in that topological element. While in other environments the use of traversal operations is strictly reduced to a set of base operations, the provided framework relies on the underlying topological environment which provides different kinds of traversal operations as well as topological functions.

The main advantage is that discrete expressions can be directly transformed into code which eases the implementation and reduces the possibility of flaws and oversights. Furthermore, the absence of iteration variables within the traversal operations makes the code easier to adapt and maintain.

6.2 Linearized Expressions

The main aim of the linearized expression approach is to ease the specification of discretized differential expressions for numerical computation. During the implementation of a specific physical model, the most error prone as well as time consuming part is the linearization of the expression which is required to obtain the system matrix. For this reason, many different derivatives of the discretized expression have to be calculated and implemented.

One of the major advantages of the linearized expressions is that they can be directly used for discretization schemes. While the abstract formulation of the equation presented in Chapter 3 shows the principles of calculating derivatives of functions which are based on topological complexes, the use of linearized expressions enables the calculation of the respective system matrices for the given problems.

One overcomes these problems by implementing only one linear functional data structure which comprises the linear functional dependence of the discretized expressions on the solution variables in the neighborhood of a linearization point or vector. Such an algebraic

structure can be easily implemented by explicitly defining basic arithmetic operations such as addition, multiplication and function application.

As a result, equation systems and system matrices can be derived automatically only by the specification and evaluation of a discretized residual expression. Derivatives with respect to the independent variables are implicitly available and do not have to be calculated explicitly by hand. Therefore, the implementation effort can be reduced enormously and testing and validation of a model can be simplified drastically.

The approach for linearized expressions can of course be used for arbitrary expansions with respect to many different variables. As an example, one can use this approach to specify higher-order Taylor series and explicitly store second order derivatives, or use other than polynomial functions. As long as the formulation describes a linear function space, and special operations preserve the structure (e.g. differentiation for Fourier series), the obtained data structure can be used for many different purposes.

By the possibility to formulate linearized expressions in a functional manner one can specify discretized differential equations in a straightforward manner, while the effort for the specification is reduced to a minimum. Consequently, expressions can be written concisely and are expanded and differentiated automatically via the specification environment.

Bibliography

[1] B. A. Allan and The Common Component Architecture Forum. Toward a 1.0 Specification of the Common Component Architecture for High Performance Computing. In *Proc. of the PARA Conf.*, Umea, Sweden, June 2006.

[2] H. P. Langtangen and X. Cai. Mixed Language Programming for HPC Applications. In *Proc. of the PARA Conf.*, page 154, Umea, Sweden, June 2006.

[3] D.L. Scharfetter and H.K. Gummel. Large-Signal Analysis of a Silicon Read Diode Oscillator. *IEEE Trans. Electron Dev.*, 16(1):64–77, 1969.

[4] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[5] R. Verfürth. *A Review of A Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques.* Wiley-Teubner, Stuttgart, 1996.

[6] J. T. Oden. A Posteriori Error Estimation, Verification and Validation in Computational Solid Mechanics. *ASME, USACM Standards*, 2002.

[7] S. Prudhomme, J. T. Oden, T. Westermann, J. Bass, and M. E. Botkin. Practical Methods for A Posteriori Error Estimation in Engineering Applications. *Int. J. Numer. Meth. Eng.*, 56:1193–1224, 2003.

[8] O.C. Zienkiewicz and J.Z. Zhu. A Simple Error Estimator and Adaptive Procedure for Practical Engineering Analysis. *Int. J. Numer. Meth. Eng.*, 24:337–357, 1987.

[9] S. Nicaise. A Posteriori Error Estimations of Some Cell Centered Finite Volume Methods for Diffusion-Convection-Reaction Problems. *SIAM J. Numer. Anal.*, 44(3):949–978, 2006.

[10] H. J. Dirschmid. *Tensoren und Felder*. Springer Verlag, 2000.

[11] T. Grasser, T.-W. Tang, H. Kosina, and S. Selberherr. A Review of Hydrodynamic and Energy-Transport Models for Semiconductor Device Simulation. *Proc. IEEE*, 91(2):251–274, 2003.

[12] S. Russenschuck. *Electromagnetic Design and Optimization of Accellerator Magnets*. CERN, 2004. Habilitation thesis.

[13] S. Holzer, M. Wagner, A. Sheikholeslami, M. Karner, G. Span, T. Grasser, and S. Selberherr. An Extendable Multi-Purpose Simulation and Optimization Framework for Thermal Problems in TCAD Applications. In *Workshop on Thermal Investigations of ICs and Systems (THERMINIC)*, pages 239–244, 2006.

[14] Simulink. *FEMLab*, 2007. http://www.femlab.de.

[15] W. Y. Yang, W. Cao, T.-S. Chung, and J. Morris. *Applied Numerical Methods using MATLAB*. Wiley Interscience, 2005.

[16] *Neural Network Toolbox 5*, 2004. http://www.mathworks.com.

[17] *Signal Processing Toolbox 6*, 2004. http://www.mathworks.com.

[18] *Image Processing Toolbox 6*, 2004. http://www.mathworks.com.

[19] D. R. Musser and A. A. Stepanov. Generic Programming. In *Proc. of the ISSAC'88 on Symb. and Alg. Comp.*, pages 13–25, London, UK, 1988. Springer-Verlag.

[20] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[21] R. Heinzl, M. Spevak, P. Schwaha, and S. Selberherr. A Generic Topology Library. In *Library Centric Sofware Design, OOPSLA*, pages 85–93, Portland, OR, USA, October 2006.

[22] M. Spevak, R. Heinzl, P. Schwaha, T. Grasser, and S. Selberherr. A Generic Discretization Library. In *Library Centric Sofware Design, OOPSLA*, pages 95–100, Portland, OR, USA, October 2006.

[23] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, New York, 1995.

[24] Douglas Gregor, Jaakko Järvi, Mayuresh Kulkarni, Andrew Lumsdaine, David Musser, and Sibylle Schupp. Generic Programming and High-Performance Libraries. *Intl. J. of Parallel Prog.*, 33(2), June 2005.

[25] A. Stepanov and M. Lee. The Standard Template Library (STL). Technical report, HP Labs, October, 1994.

[26] Boost. *Boost Lambda Library*, 2001. http://www.boost.org.

[27] J. Järvi and G. Powell. The Lambda Library: Lambda Abstraction in C++. In *Proceedings of the Second Workshop on C++ Template Programming (TMPW'01) at OOPSLA 2001*, October 2001.

[28] B. McNamara and Y. Smaragdakis. Functional Programming in C++ using the FC++ Library. *SIGPLAN*, 36(4):25–30, April 2001.

[29] Boost. *Boost Phoenix 2*, 2006. http://spirit.sourceforge.net/.

[30] G. Berti. GrAL - The Grid Algorithms Library. In *ICCS '02: Proc. of the Conf. on Comp. Sci.*, volume 2331, pages 745–754, London, UK, 2002. Springer-Verlag.

[31] G. Berti. *Generic Software Components for Scientific Computing*. Dissertation, Technische Universität Cottbus, 2000.

[32] K. Kramer, W. Nicholas, G. Hitchon, University of Wisconsin. *Semiconductor Devices, a Simulation Approach*. Prentice Hall Professional Technical Reference, 1997.

[33] S. Selberherr. *Analysis and Simulation of Semiconductor Devices*. Springer, Wien–New York, 1984.

[34] C. S. Rafferty and R. K. Smith. Solving Partial Differential Equations with the Prophet Simulator, 1996.

[35] T. Binder, A. Hössinger, and S. Selberherr. Rigorous Integration of Semiconductor Process and Device Simulators. *IEEE Trans.Comp.-Aided Design of Int. Circ. and Systems*, 22(9):1204–1214, 2003.

[36] R. Sabelka and S. Selberherr. A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures. *Microelectronics Journal*, 32(2):163–171, 2001.

[37] A. Bonfiglioli, B. Carpentieri, and M. Sosonkina. EulFS: A Parallel CFD Code for the Simulation of Euler and Navier-Stokes Problems on Unstructured Grids. In *Proc. of the PARA Conf.*, Umea, Sweden, June 2006.

[38] K.-A. Mardal. SyFi - An Element Matrix Factory, with Emphasis on the Incompressible Navier-Stokes Equations. In *Proc. of the PARA Conf.*, Umea, Sweden, June 2006.

[39] A. Fabri. CGAL- The Computational Geometry Algorithm Library, 2001.

[40] R. Heinzl. *Concepts for Scientific Computing*. Dissertation, TU Wien, Institut für Mikroelektronik, Vienna, Austria, September 2007.

[41] W. Bangerth, R. Hartmann, and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*, 2004. http://www.dealii.org.

[42] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – A General Purpose Object Oriented Finite Element Library. Technical Report ISC-06-02-MATH, Institute for Scientific Computation, Texas A&M University, 2006.

[43] W. Bangerth. A Framework for the Adaptive Finite Element Solution of Large Inverse Problems. Technical Report 04-39, Institute for Computational Engineering and Sciences (ICES), University of Texas at Austin, 2004.

[44] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[45] L.Q. Lee, J.G. Siek, and A. Lumsdaine. The generic graph component library. *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 399–414, 1999.

[46] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. *Proc. Supercomp. '90*, pages 2–11, 1990.

[47] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Petsc home page, http://www.mcs.anl.gov/petsc, 1998.

[48] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2005.

[49] X. S. Li, O. A. Marques, and Y. Zhang. EigenAdept- The Expert Eigensolver Toolbox. In *Proc. of the PARA Conf.*, Umea, Sweden, June 2006.

[50] *http://www.ansys.com*, 2000.

[51] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. A High Performance Generic Scientific Simulation Environment. In *Proc. of the PARA Conf.*, page 61, Umea, Sweden, June 2006.

[52] J. Stoer. *Numerische Mathematik 1*. Springer, 1999.

[53] Boost. *Boost Rational Library*. http://www.boost.org.

[54] H. J. Dirschmid. *Matrizennumerik*. Technische Universität Wien, 1997. lecture notes.

[55] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method*. McGraw-Hill, Berkshire, England, 1987.

[56] C. A. Brebbia and J. Dominguez. *Boundary Elements: an Introductory Course*. WIT Press, 1998.

[57] J. C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. Chapman and Hall, 1989.

[58] A. F. Franz, G. A. Franz, S. Selberherr, C. Ringhofer, and P. Markowich. Finite Boxes - A Generalization of the Finite Difference Method Suitable for Semiconductor Device Simulation. *IEEE Trans. Electron Dev.*, 30(9):1070–1082, 1983.

[59] Institut für Mikroelektronik, Technische Universität Wien, Austria. *MINIMOS-NT 2.1 User's Guide*, 2004.

[60] T. Binder. *Rigorous Integration of Semiconductor Process and Device Simulators*. Dissertation, TU Wien, Institut für Mikroelektronik, 2002.

[61] E.P. Mücke, I. Saias, and B. Zhu. Fast Randomized Point Location without Preprocessing in Two-and Three-Dimensional Delaunay Triangulations. *Computational Geometry: Theory and Applications*, 12(1-2):63–83, 1999.

[62] D. T. Lee and F. P. Preparata. Location of a Point in a Planar Subdivision and its Applications. *Proceedings of the eighth annual ACM Symposium on Theory of Computing*, pages 231–235, 1976.

[63] N. Sarnak and R.E. Tarjan. Planar Point Location Using Persistent Search Trees. *Communications of the ACM*, 29(7):669–679, 1986.

[64] G. E. Bredon. *Topology and Geometry*. Springer Science and Business Media, 1993.

[65] T. Barth. A Posteriori Error Estimation and Mesh Adaptivity for Finite Volume and Finite Element Methods. In *Springer series Lecture Notes in Computational Science and Engineering (LNCSE)*, volume 41, pages 183–203, 2004.

[66] S. Lang. *Algebra, third Edition*. Addison Wesley Publ., 1993.

[67] Corrado Böhm, editor. *Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, March 25-27, 1975*, volume 37 of *Springer series Lecture Notes in Computational Science (LNCS)*, Rome, March 25-27 1975. Springer.

[68] G. Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Addison Wesley Publ., 1988.

[69] D. Pederson. A Historical Review of Circuit Simulation. *Circuits and Systems, IEEE Transactions on*, 31(1):103–111, 1984.

[70] C.J. Price. Function-Directed Electrical Design Analysis. *Artificial Intelligence in Engineering*, 12(4):445–456, 1998.

[71] M. Dong. *Process Modeling, Performance Analysis and Configuration Simulation in Integrated Supply Chain Network Design*. PhD thesis, Virginia Polytechnic Institute and State University, 2001.

[72] L. Chwif, M.R.P. Barretto, and E. Saliby. Supply Chain Analysis: Spreadsheet or Simulation. *Proceedings of the 2002 Winter Simulation Conference*, pages 59–66, 2002.

[73] J.K. Wu. *Neural Networks and Simulation Methods*. Marcel Dekker, 1994.

[74] L.E. Breslau. Advances in Network Simulation. *Computer*, 33(5):59–67, 2000.

[75] A. Weinmann. *Regelungen II. Analyse und Technischer Entwurf*. Springer, 1995.

[76] F. Harary. *Graph Theory*. Perseus Books, 1999.

[77] S. Haykin. *Neural Networks*. Macmillan Publishing Company, 1994.

[78] S. Fortune. Voronoi Diagrams and Delaunay Triangulations. *CC Press Discrete Mathematics and its Applications Series*, pages 377–388, 1997.

[79] M.S. Lundstrom. *Fundamentals of Carrier Transport*. Cambridge University Press, 2000.

[80] L. Gaul, M. Kögl, and M. Wagner. *Boundary Element Methods for Engineers and Scientists*. Springer, Verlag, 2003.

[81] P.K. Kythe. *Fundamental Solutions for Differential Operators and Applications*. Birkhauser Verlag AG, 1996.

[82] H. R. Schwarz. *Numerische Mathematik*. B. G. Teubner, Stuttgart, 1997.

[83] S. Wagner. *Small-Signal Device and Circuit Simulation*. Dissertation, TU Wien, Institut für Mikroelektronik, 2004.

[84] R. Bauer. *Numerische Berechnung von Kapazitäten in dreidimensionalen Verdrahtungsstrukturen*. Dissertation, TU Wien, Institut für Mikroelektronik, 1994.

[85] R. Sabelka. *Dreidimensionale Finite Elemente Simulation von Verdrahtungsstrukturen auf Integrierten Schaltungen*. Dissertation, TU Wien, Institut für Mikroelektronik, 2001.

[86] E. Cuthill, , and J. McKee. Reducing The Band Width of Sparse Symmetric Matrices. In *Proc. ACM National Conference*, 1969.

[87] D. Abrahams, J. Siek, and T. Witt. New Iterator Concepts. Technical Report N1477 03-0060, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2003.

[88] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[89] T. L. Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[90] T. L. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

Own Publications

[1] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. A Novel Technique for Coupling Three-Dimensional Mesh Adaptation with an A Posteriori Error Estimator. In *2005 PhD Research in Microel. and Elect.*, pages 175–178, Lausanne, Switzerland, July 2005.

[2] P. Schwaha, R. Heinzl, M. Spevak, and T. Grasser. Coupling Three-Dimensinoal Mesh Adaptation with an A Posteriori Error Estimator. In *Proc. Conf. on Sim. of Semiconductor Processes and Devices*, pages 235–238, Tokio, Japan, September 2005.

[3] R. Heinzl, P. Schwaha, M. Spevak, and T. Grasser. Adaptive Mesh Generation for TCAD with Guaranteed Error Bounds. In *2005 Europ. Sim. and Modeling Conf.*, pages 425–429, Porto, Portugal, October 2005.

[4] M. Spevak and T. Grasser. Discretization Schemes for Macroscopic Transport Equations on Non-Cartesian Coordinate Systems. In *2005 Europ. Sim. and Modeling Conf.*, pages 474–478, Porto, Portugal, October 2005.

[5] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. Concepts for High Performance Generic Scientific Computing. In *Proc. of the 5th MATHMOD*, volume 1, pages 4-1–4-9, Vienna, Austria, February 2006.

[6] M. Spevak, R. Heinzl, P. Schwaha, and T. Grasser. Simulation of Microelectronic Structures using A Posteriori Error Estimation and Mesh Adaptation. In *Proc. of the 5th MATHMOD*, page 317, Vienna, Austria, February 2006.

[7] M. Spevak, R. Heinzl, P. Schwaha, and T. Grasser. Process and Device Simulation With a Generic Scientific Simulation Environment. In *Proceedings, MIEL 2006*, volume 2, pages 475–478, Belgrad, Serbia and Montenegro, April 2006.

[8] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. Multidimensional and Multitopological TCAD with a Generic Simulation Environment. In *Proc. of the ICCDCS*, pages 173–176, Playa del Carmen, Mexico, April 2006.

[9] R. Heinzl, M. Spevak, and P. Schwaha. Concepts for High Performance Generic Scientific Computing. In *Proc. of the 12th Conf. Student EEICT 2006*, volume 4, pages 446–450, Brno, Czech Rep., April 2006.

[10] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. High Performance Process and Device Simulation with a Generic Environment. In *Proc. of the 14th Iranian Conf. on El. Eng. ICEE 2006*, pages 446–450, Teheran, Iran, May 2006.

[11] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. A High Performance Generic Scientific Simulation Environment. In *Proc. of the PARA Conf.*, page 61, Umea, Sweden, June 2006.

[12] P. Schwaha, R. Heinzl, M. Spevak, and T. Grasser. Advanced Equation Processing for TCAD. In *Proc. of the PARA Conf.*, page 64, Umea, Sweden, June 2006.

[13] M. Spevak, R. Heinzl, P. Schwaha, and T. Grasser. A Computational Framework for Topological Operations. In *Proc. of the PARA Conf.*, page 57, Umea, Sweden, June 2006.

[14] R. Heinzl, P. Schwaha, M. Spevak, and T. Grasser. Performance Aspects of a DSEL for Scientific Computing with C++. In *Proc. of the POOSC Conf.*, pages 37–41 Nantes, France, July 2006.

[15] P. Schwaha, R. Heinzl, M. Spevak, and T. Grasser. A Generic Approach to Scientific Computing In International Congress on Computational and Applied Mathematics (ICCAM) page 137, Leuven, Belgium, July 2006.

[16] M. Spevak, R. Heinzl, P. Schwaha, and T. Grasser Automatic Linearization using Functional Programming for Scientific Computing. In International Congress on Computational and Applied Mathematics (ICCAM) page 147, Leuven, Belgium, July 2006.

[17] R. Heinzl, M. Spevak, P. Schwaha, and S. Selberherr. A Generic Topology Library. In *Library Centric Sofware Design, OOPSLA*, pages 85–93, Portland, OR, USA, October 2006.

[18] M. Spevak, R. Heinzl, P. Schwaha, T. Grasser, and S. Selberherr. A Generic Discretization Library. In *Library Centric Sofware Design, OOPSLA*, pages 95–100, Portland, OR, USA, October 2006.

[19] R. Heinzl, M. Spevak, P. Schwaha, T. Grasser, S. Selberherr: Performance Analysis for High-Precision Interconnect Simulation. In European Simulation and Modeling Conference (ESMC). pages 113–116, Toulouse, France, October 2006.

[20] M. Spevak, R. Heinzl, P. Schwaha, and S. Selberherr. A Computational Framework for Topological Operations. *Applied Parallel Computing. State of the Art in Scientific Computing* volume 4699, pages 781-790, 2007.

[21] M. Spevak, and T. Grasser. Discretization of Macroscopic Transport Equations on Non-Cartesian Coordinate Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 26, pages 1408–1416, 2007.

Curriculum Vitae

Michael Spevak was born in Vienna on the 24th of August 1982.

In June 2000 he finished the high school education at the Gymnasium Stockerau with the Matura.

From October 2000 to November 2004 he was studying electrical engineering at the Technische Universität Wien, where he received the degree of Diplomingenieur for electrical engineering at the Institute for Microelectronics.

Since October 2004 he has been studying mathematics at the Technische Universität Wien.

From December 2004 to February 2007 he worked at the Institute for Microelectronics of the Technische Universität Wien as doctoral candidate and as university assistant.

Since March 2007 he has been working as patent attorney candidate in Vienna.