computer graphics & vision

Inffeldgasse 16/2, A-8010 Graz

www.icg.tu-graz.ac.at

TUG

WAVELET BASED REAL-TIME DEFORMABLE OBJECTS

A wavelet approach for BEM

Antonio Rella

Inst. for Computer Graphics and Vision
Graz University of Technology, Austria

Master thesis
Graz, December 14, 2009

contact: Antonio Rella antonio.rella@student.tugraz.at

Master Thesis

Wavelet based real-time deformable objects
A wavelet approach for BEM

Master's Thesis

at

Graz University of Technology

submitted by

Antonio Rella

Institute for Computer Graphics (ICG),
Graz University of Technology
Inffeldgasse 16/2
A-8010 Graz, Austria

December 14, 2009

Advisor: Dipl.-Ing. Dr.techn. Markus Grabner

Abstract

Calculation of deformation of 3-D models is a well know realm in analytic mathematics. As early as a century past algorithms have been developed to compute such deformations. In general, these computations for physically correct and accurate deformations are cumbersome and time-consuming. For a few years computers became more powerful and have the capabilities to compute deformations for complex models in an appropriate period of time. Beside these physically correct deformations, realistic and intuitive deformations have been described by less time consuming algorithms, which also could be computed in real-time. However, these deformations are not precise and can only satisfy an observer at first sight. This thesis is concerned with the boundary element computation algorithm for accurate deformation descriptions of three dimensional models. For this method the underlying coefficient matrix is fully populated and therefore more difficult to solve. The approach of lazy wavelets, on the contrary, is able to remove less relevant geometrical information while accepting the emerging error, to achieve a more sparse coefficient matrix and to ease the calculation.

Kurzfassung

Berechnungsalgorithmen zur Verformung von 3-D Modellen sind ein bekanntes Gebiet in der der analytischen Mathematik. Schon vor einem Jahrhundert wurden verschiedene Methoden entwickelt um solche Verformungen zu errechnen. Rechenschritte für die Berechnung einer exakten Verformung sind im Allgemeinen sehr aufwendig und zeitintensiv. Erst seit ein paar Jahren sind Computer soweit entwickelt, dass sie diese Verformungen auch für komplexere Modelle in einem angemessenen Zeitrahmen berechnen können. Neben den exakten, realen Verformungen wurden auch realistische, intuitive Verformungen beschrieben, deren Rechenschritte nicht so aufwendig sind und in Echtzeit berechnet werden können. Allerdings sind diese nicht exakt und genügen dem Betrachter nur bei einem schnellen Blick. Diese Diplomarbeit befasst sich mit der Berechnungsvorschrift der Randelementenmethode, auch bekannt als Boundary element method, für die Verformung von drei-dimensionalen Modellen. Die zugrunde liegende Koeffizientenmatrix ist hierbei dicht besetzt und schwer zu lösen. Der Ansatz von lazy wavelets hingegen ist im Stande weniger relevante geometrisch Information bei der Beschreibung dieser Matrix auszublenden um letztendlich eine dünn besetzte Matrix für eine einfachere Berechnung zu erhalten.

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

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

Acknowledgements

I am indebted to my student mates and colleagues at the ICG (Institute for Computer Graphics) who have provided invaluable help and feedback during my work. I especially wish to thank my advisor, Markus Grabner, for his careful attention to my questions and countless hours of toil in correcting draft versions of this thesis.

Without the support and understanding of my father, helping me in first English translations and diction, my friends giving me thought-provoking impulses as well as the continued support of my partner, this thesis would not have been possible.

Antonio Rella
Graz, Austria, December 2009

Credits

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

This thesis is concerned with a 3-D deformation method to compute surface displacements due to acting forces on an arbitrary body. Its main goal is to find a way for a nearly real deformation in an appropriate period of time. *Real deformation* means in this context, the deformation is physically correct and not an approximation. Since real deformations are computationally very expensive, a new way will be presented here, where errors are accepted to reduce this computational effort. However, if no error is accepted the result is physically correct and precise. A second goal is the real-time capability, which will give a more intuitive feeling for a virtual deformation to the user. Further, a force feedback device will be included to feel the reacting forces. This approach uses the wavelet technique and the boundary element method having a much smaller underlying matrix compared to other real deformation method, to achieve these goals. Additionally to decrease the computing time, the software will be implemented on the GPU for parallel processing using CUDA. The use in application areas, such as in medicine and in engineering, gives the motivation to deform models in an accurate and fast way. These applications, using the boundary element method for virtual deformations, or similar neighbor topics – heat transfers and fluid flows –, are widely used. For instance for training purposes in virtual medical surgery, allowing a medical trainee to interact with virtual human organs and to deform them by stitching them up.

1.1 Overview

The following chapters explain in detail the required methods and functions to understand how deformations are described using the boundary element method and wavelets. Chapter 2, *Related work*, gives an overview on the related work in the context of deformation methods and explains the main differences between them. This thesis uses the boundary element method as a mathematical instrument to describe and compute body deformations. The theoretical part about this method is explained in the Chapter 3, *Boundary element method*, showing how to prescribe deformations, setting up the underlying coefficient matrix and computing a final result. Since this topic is normally not part of computer graphics it is explained in-depth in a self-contained way. Chapter 4, *Elastostatics*, is designed to describe 3-D deformations for solid materials in their mathematical and analytical way and depicts the definition of displacement, of stress and of strain. Similar to the boundary element chapter it is explained in-depth since this topic is not frequently addressed by the computer graphics community. Chapter 5 guides through *3-D multiresolution surfaces* and explains several surface subdivision algorithms needed to create a surface representation with subdivision connectivity of the model's boundary, as well as the usage of wavelets to store this newly created surface with subdivision connectivity. Chapter 6 deals with *Matrix representations and storage* and shows several

techniques to store a sparse matrix efficiently. Chapter 7 covering *Iterative numerical solvers for linear equation systems* gives an overview to some common iterative solvers and to the finally implemented bi-conjugate gradient solver. The used haptic force feedback device and its properties are shown in Chapter 8, *Haptic Device*. The appendix also guides through the installation under Linux as there might be conflicts between installed Linux distributions and device driver versions. The first Chapter of contribution, Chapter 9 called *Extension to the boundary element method* gives a 1-D example and shows the reformulation of the boundary integral equation needed for this work. Chapter 10, *Wavelets*, explains the application of the boundary element methods on the new created surface with subdivision connectivity, using wavelets for the boundary element values representation. Some problems and implemented solutions of the application are shown in Chapter 11, *Application*. The setup of the implemented application and its execution parameters can be found in the appendix. The results are collected in Chapter 12, *Results*, and shows performance, accuracy and computed data volumes in context of CPU and GPU tested on several computer platforms. In Chapter 13, *Conclusion and future work*, benefits and drawbacks of the thesis are being evaluated.

1.2 Contribution

This thesis uses wavelets to simplify and reduce the mathematical data amount for a faster computation, while accepting an error threshold during the calculation. For different types of interpolating a function over the boundary of the body different types of wavelets can be used. For a constant interpolation one surface patch relates to one displaceable surface point. For linear interpolation a surface patch partly relates to at least three displaceable surface points. The difficulty now lies in the usage of linear wavelets, interpolating over the element and relating the integrated data to the several surface points. To assemble the coefficient matrix in the linear case a reformulation of the boundary integral equation is needed. Since subdivision connectivity, here also denoted as sub-connectivity, of the model is needed to apply the wavelets approach, the mesh points of the surface are needed to be recomputed to achieve sub-connectivity, which was not part of this thesis. The sub-connectivity was created out of a simple base mesh using subdivision methods for test purposes. For a model with sub-connectivity, the resulting coefficient matrices is going to be transformed using linear wavelets into a new representation form. Within this new matrix formulation it is allowed to ignore small matrix entries resulting more sparse coefficient matrices consisting around 80% of zero entries naturally depending on the error threshold accepted. The sparse matrix storage uses the compressed row storage format. In a first solution the matrix has been setup completely and small values have been deleted. However, this solution is very time consuming since all connections are taken into account, are computed and evaluted, just to be deleted afterwards. An improvement has been implemented whereas all needed connections are determined in a first step and computed in a second step. The solution uses the subconnectivity to determine in upper subdivision levels if a connection in lower levels needs also to be checked or is allowed to be skipped. An iterative solver using the bi-conjugate gradient method is fed with this sparse matrix and computes the deformation result. This approach has been tested with several surface meshes on several computer platforms. Another important goal, which has been achieved for simplified cases, is the real-time capability of the computation. By the usage of the wavelets approach the computing time is still reduced dramatically, as there is less data stored in the matrix. The iterative solver is the most time consuming part in the computation process and was decreased by using the parallel computation capability of the graphics card. The solver was implemented in the graphics programming language CUDA to solve the equation system and results in a second speedup.

Part II

Related work

Chapter 2

Deformation methods

2.1 State of the art

Calculation of deformation of 3-D models is a well know realm in analytic mathematics. As early as a century ago algorithms have been developed to compute such deformations. Generally, they can be classified into two groups. On the one hand, real deformations which are physically correct and depend on material properties like the stiffness tensor. They follow physical laws for elastic deformation and have high computation loads. On the other hand, realistic deformations which are intuitively correct, but do not describe a correct physical deformation. They commonly satisfy the impression of a deformation for an observer. However, since real deformation approaches have very high computing times to calculate an exact solution of a given problem, realistic deformations have the right to exist next to them and are welcome for applications whose deformation's accuracy is secondary. The finite element method or boundary element method are common for real deformation computations. A second classification can be done by the type of computation, namely by differentiating into classical analytical methods and into numerical methods. For analytical solutions an integral formulation of the deformation is described over the whole body, but they are limited, since there are difficulties to carry out the integrals procedure (Equation 2.1.1). Analytical formulations have been found for very specific cases only.

$$\iiint_{\Omega} \cdots \mathrm{d}x\mathrm{d}y\mathrm{d}z = ??? \tag{2.1.1}$$

In contrast, for numerical solutions this integral is evaluated partially over the body and is well adapted for deformation problems in the mechanics of solids (Equation 2.1.2).

$$\iiint_{\Omega} \cdots \mathrm{d}x\mathrm{d}y\mathrm{d}z \approx \sum_{e} \iiint_{\Omega_e} \cdots \mathrm{d}x\mathrm{d}y\mathrm{d}z \tag{2.1.2}$$

The formulated problem has many solutions as depicted in Figure 2.1.

2.2 Computer trends

In the last decade, personal computers became powerful machines pushed by the computer game industry whose capacity is increasing with each game generation produced. The customer needs to be satisfied with better, faster and finally also more realistic impressions of the game. The first step, it can be said, was the introduction of 3-D graphics which later have been departed from the CPU, due

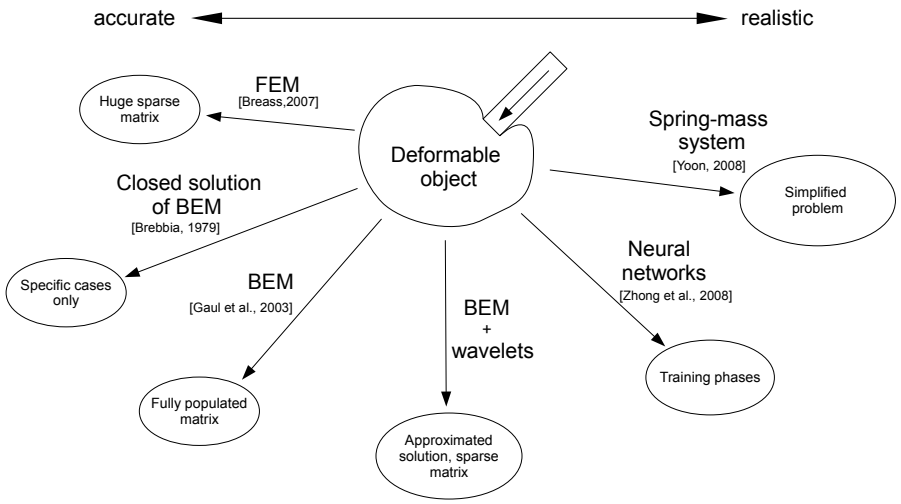


Figure 2.1: Approaches for deformable objects

to the high computation loads for 3-D rendering. An special processor, the GPU, which has been developed just for 3-D graphics computations and renderings, was placed on the graphics card and has been improved faster than *Moores's law* predicts. This law prescribes a long-term trend in the history of computer hardware as shown in Figure 2.2 [nVidia, 2008]. In order to render 3-D models equipped with textures and lights, these processors use a pipeline technique to calculate the pixels on the screen. Figure 2.3, referred from the book of Foley et al. [1995], shows this pipeline principle. Furthermore, this procedure is parallelized by many cores inside the GPU and, since these pipeline steps were hard wired on the GPU, it was not possible to interrupt or change these computation procedures. Only by the introduction of vertex and fragment shader – in Figure 2.3 they are named programmable vertex and fragment processors – it was possible to influence this pipeline and add new code segments to create better visual effects like waves, environmental reflections and shadows among others. Therefore, several high level shading programming languages like *Cg* (C for graphics), *HLSL* (High Level Shading Language for DirectX) and *Renderman* were developed. Renderman was created and used by Pixar Animation Studios. More about these languages can be found in the books of Fernando and Kilgard [2003] and Luna [2006]. The GPU becomes more and more a new instrument for parallel-programmed applications for general purposes. The latest state of GPU hardware can be seen and programmed as a parallel processor using an extended programming language of *C* called *CUDA*, which is a shorten form for Compute Unified Device Architecture, developed by nVidia for their graphics cards and applicable since the nVidia GeForce 8800 series. Nowadays affordable graphics cards have many shader cores – also called CUDAcores – for parallel processing, for instance the new nVidia GT300's chip architecture will have up to 16 multiprocessors with 32 shader cores each, thus 512 shader cores.

2.3 Real deformation methods

2.3.1 Finite element method

The finite element method (FEM) is a very famous numerical approach for real physical deformations to compute and calculate stresses and strains inside a given model [Gaul, 2009]. In numerical math-

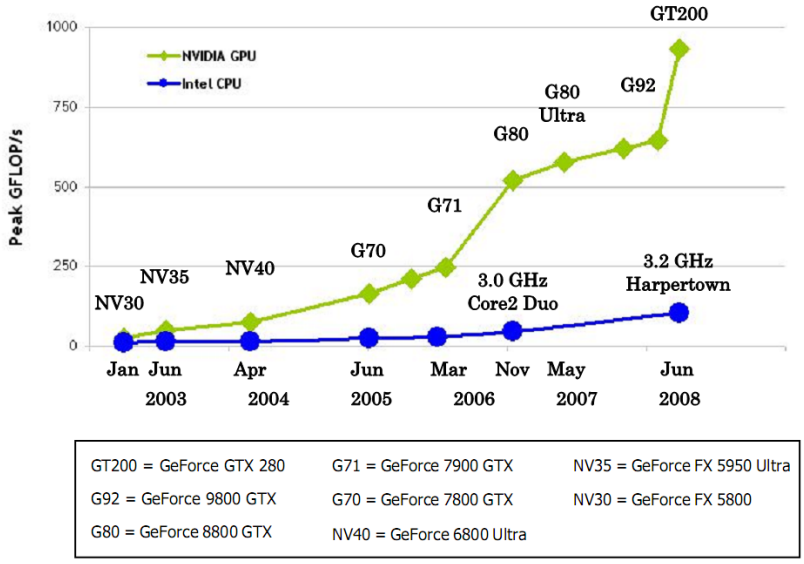


Figure 2.2: Long-term trend of the nVidia product series

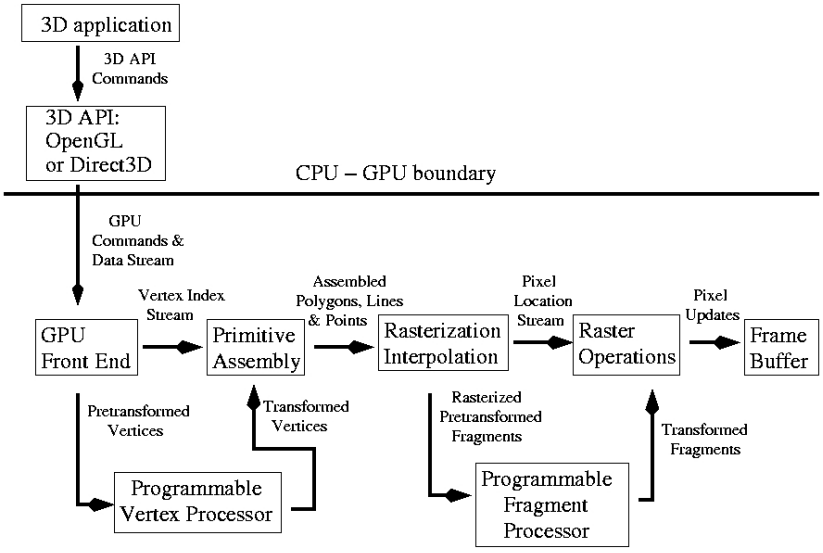


Figure 2.3: Graphics pipeline for 3-D rendering

ematics FEM has a wide range of applications, namely for stress and torsion analysis or heat transfer and fluid flow determination among others. As the name already bespeaks, this method subdivides the body into a finite number of elements and nodes to compute the strains and stresses between these nodes. These nodes are defined on the elements, with one or more nodes per element, depending on the degree of accuracy wanted. In a first process, each node can be assigned a stress or a strain, depending on the boundary conditions, while the opposite value will be computed. Relations and interaction values of these nodes are calculated and are stored in a matrix. Normally, for a static body deformation, due to an external force, all nodes inside the body are assigned no stress, while the strain or the displacement of the nodes is being computed to yield the resulting body deformation. In case of gravity all nodes inside the body are loaded with a gravity related stress and the displacement is computed. Figure 2.4 shows a 2-D example of a deformed I-beam. The interaction values stored

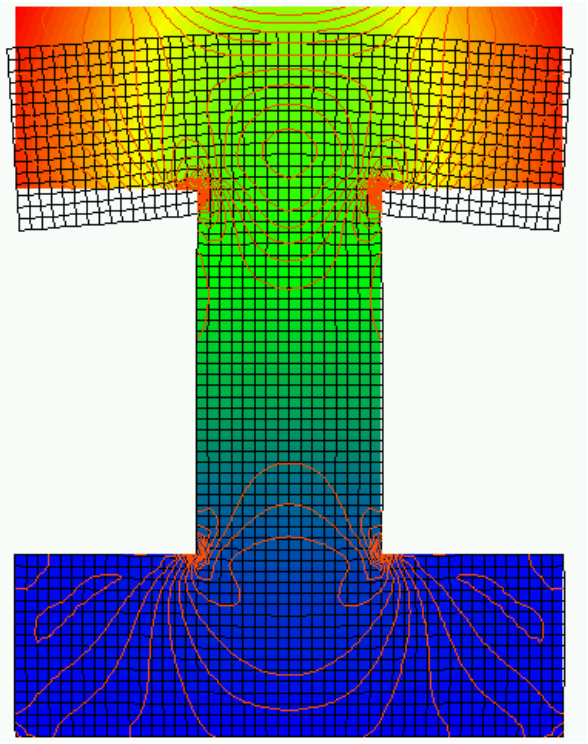
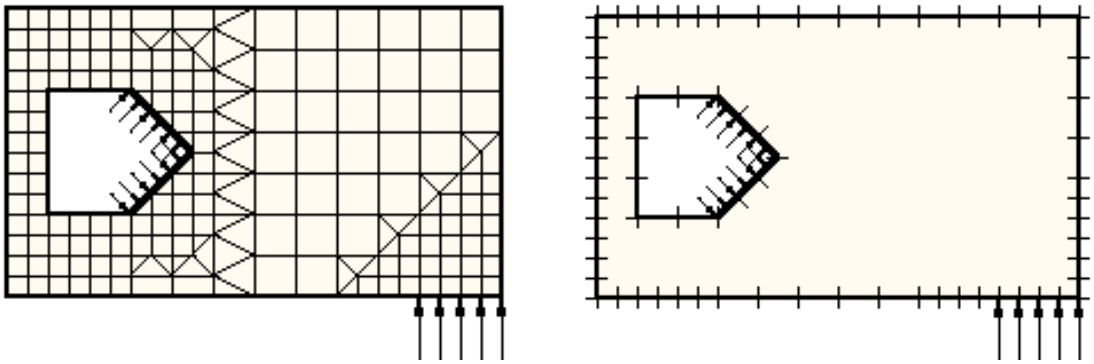


Figure 2.4: 2-D example for FEM

in the matrix, which is normally sparse for the finite element method, can be rearranged to result in a linear equation system whose solution will lead to the unknown stresses and strains at each node. Iterative linear equation solvers allow to solve those sparse matrix equation systems in a finite number of steps. More information can be found in the book of Braess [2007] and others.

2.3.2 Boundary element method

The boundary element method (BEM), as a second representative for numerical approaches to compute real physical deformations, has lived in a shadowy existence beside the finite element method, since the resulting matrix equation system is non-symmetric and fully populated, resulting in difficulties to solve the system by iterative algorithms [Rüberg, 2008]. However, the matrix is smaller and since computers became more powerful, a matrix inversion of boundary element problems with



(a) FEM discretization **(b)** BEM discretization

Figure 2.5: 2-D discretization of a model for FEM and BEM [Gaul, 2009]

a reduced number of nodes can be computed in an appropriate amount of time. The application area for BEM is similar to FEM and ranges from stress and torsion analysis to fluid flow and heat transfer determination. An important difference between the finite element method and the boundary element method, apart from the need to continuously resist against shears and torsions inside the body, is the required assumption of a homogeneous body material as it is impossible to prescribe material relations inside the body without knowing about geometrical or material differences [Wrobel, 2002]. By this reduction in the degree of freedom, the deformation problem can be defined over the boundary only. Figure 2.5a shows a discretized model for the FEM while Figure 2.5b depicts the discretization of the same model for the BEM [Gaul, 2009]. The advantages and disadvantages of BEM, referring to the lecture notes of Gaul [2009], are:

+ Discretization of the boundary only.
+ Simplified pre-processing.
+ Improved accuracy in stress concentration problems.
+ Simple and accurate modeling of problems involving infinite and semi-infinite domains.
+ Simplified treatment of symmetrical problems (no discretization needed in the plane of symmetry).

– Non-symmetric, fully populated system of equations in collocation BEM.
– Treatment of inhomogeneous and non-linear problems.
– Requires the knowledge of a suitable fundamental solution.
– Practical application relatively recent, not as well known as FEM among users.

Similar to the finite element method, the boundary of the solid body needs to be split in a finite number of elements and nodes with their desired boundary conditions. They can also be set to stresses or displacements and are defined over the elements, with one or more nodes per element, depending on the degree of accuracy wanted. One node per element requires a constant interpolation, while two or more nodes give the opportunity to interpolate linear, quadratic or higher order over the element.

Of course, a constant interpolation of the boundary element values over the elements leads to cheaper computation while its accuracy is lower than when doing linear or higher ordered interpolations of the boundary values over the elements. However, the accuracy also increases with the number of elements created out of the boundary through denser mesh representations. More detail description of the boundary element method can be found in the books of Gaul et al. [2003], Wrobel [2002] and Banerjee [1994] among others. An implementation of the boundary element method used for deformations of virtual objects is shown in their Artdefo (accurate real-time deformable objects) paper from James and Pai [1999]. The investigation uses the constant element case and applies only for smaller meshes, as the amount of data increases for large objects consisting of many triangles. A fast update results from the requirement of an only few boundary conditions change. An extension to this boundary element method can be given by using a subconnected mesh description for a given boundary. From several mesh subdivision algorithms, such as the Loop subdivision method [Loop, 1987] for triangles or the Catmull-Clark's subdivision method [Catmull and Clark, 1978] for quadrilaterals among others, a mesh with subdivision connectivity can be created. Eck et al. [1995] present methods to resample arbitrary surface meshes to achieve sub-connectivity. In combination with the usage of wavelets the number of close-to-zero values inside the matrix will be increased, which are set, by accepting this error, to zero and therefore the matrix becomes sparser. The computed matrix resulting from the boundary element method will have normally no zero entries. A constant boundary value interpolation with usage of adapted Haar wavelets was discussed and developed at the ICG at Graz University of Technology. This theses shows how this can be extended and adapted for a linear interpolation using linear or lazy wavelets and also gives a clue for the usage of higher ordered wavelets and interpolation types.

2.4 Realistic deformation methods

2.4.1 Spring-mass-damper method

The spring-mass-damper model is the most common real-time representative due to its lower computation effort. The model is commonly used to simulate deformations of human organs or muscles. The simulation can have to effect on the whole body or the boundary of the body only [Hähnke, 2004]. The spring-mass-damper method connects each vertex or node of the body to each neighboring node via a spring-mass-damper system [Yoon, 2008]. A displacement of any of the nodes will affect the neighboring nodes and so on (Figure 2.7a). Each node carries a perceptual value of the whole body mass while the spring's values between them represent the stiffness or elasticity of the body. To prevent an oscillation of the body, which would be a result of the masses and springs and is not desired, a third component, the damper value, is needed. Figure 2.6 explains the spring-mass-damper principle for a mass connected to a fixed boundary. The force acting on one node due to elasticity results from the influence of the nodes in the vicinity. Additionally, directions to neighboring nodes can be taken into account to simulate different physical properties, for instance, stretching the body in direction of x to be harder than in direction of y. For cheaper computational costs the number of nodes might be reduced to those on the surface, which also leads to a deformation, but the model does not deform like a balloon anymore, because during a deformation of this type the volume of the body is not taken into account. However, additionally each node needs also be connected via a spring and damper to it's original undeformed position in space to allow the node to return to it's initial position. This is shown in figure 2.7b giving such an example for a 2-D case. Application areas for the spring-mass-damper method ranges form clothes simulation, living tissue simulation, emotion sensitive avatars and virtual medical surgeries for training purposes or as virtual preview to the patient in case of aesthetic surgery.

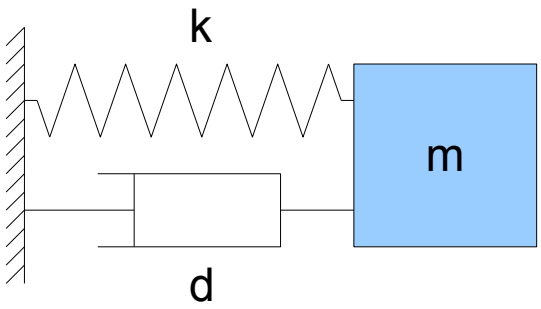
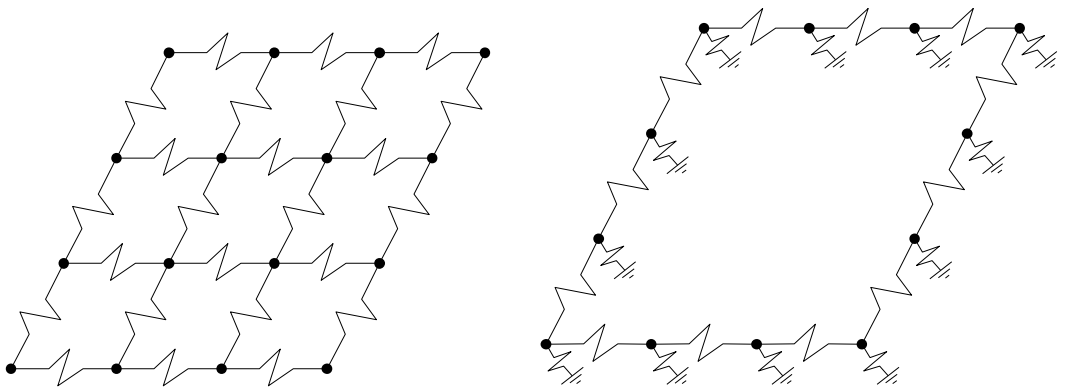


Figure 2.6: Spring-mass-damper system



(a) SMD model with inner nodes

(b) SMD model with boundary nodes only

Figure 2.7: 2-D spring-mass-damper model types

2.4.2 Cellular neural network method

Methods using a neural network as backbone of a deformation algorithm were derived and improved by [Zhong et al., 2008] and [Morooka et al., 2008]. The deformation is formulated as an dynamic cellular neural network. The energy stored in the body due to an acting external force, is propagated along mass points or nodes, resulting from a subdivision, by non-linear cellular neural network activities. The method is used for highly non-linear or plastic deformations in simulations of soft objects like human organs. In combination with the finite element method, the method tries to reduce the computation load needed to solve the linear equation system and the huge amount of data stored in the matrix, normally at the expense of the accuracy of the computed result. The linear elastic deformation problem is changed into a cellular neural network to avoid the expensive computations for linear elasticity. An advantage is the large-ranged deformation possibility, while only small-ranged deformations are computed by FEM, BEM or spring-mass-damper model. In the paper of [Hambli et al., 2006] a neural network in combination with the finite element method is described. Neural networks are employed as numerical devices which substitute the finite element computation part needed for the deformation. In offline pre-calculations the artificial neural network is trained by some randomly generated parameter sets which are suited to deform the model. As an easy example, a ball can be considered whose used neural network is trained with radial forces only to finally simulate a jumping ball with realistic physical deformation properties. This method predicts the typical behaviors of living tissues and easily accommodates isotropic, anisotropic and inhomogeneous materials next to local and large-range deformations.

Chapter 3

Boundary element method

3.1 Introduction

This part of the thesis will be concerned with related work to the boundary element method. The boundary element method, introduced in 1970, describes a way for computational solutions of engineering problems [Brebbia, 1979]. These problems can be defined as elliptic, parabolic or hyperbolic. Additionally, a boundary value problem is generally defined as static or dynamic (time-depended), but in this thesis the attention is mainly focused on the elliptic static boundary value problem. In general it is used for computation of complex problems described by differential equations, like those which come up in the electrostatic field, in fluids, in electromagnetism or, as in this context, in material deformations. The finite element method [Braess, 2007] solves these equations by subdividing the domain into a finite number of subdomains. For the boundary element method these equations will be transformed into their boundary elements for calculation. Applying the boundary element method instead of the finite element method computation gets easier and more efficient. The computation is valid as long as the applied domain is homogeneous. This section will guide through related methods and algorithms of the boundary element technique.

Initial steps in this direction were done by the Swedish mathematician Ivar Fredholm [1903] who formulated first boundary value problems and demonstrated solutions by discretizing the problem. Most of this work was done in the theoretical perspective due to scarce possibilities realm for solving large linear matrix systems. Most of the time has been spent to find evidences for the existence and uniqueness of solutions. Nowadays, high-speed computers allow doing practical steps and application, whose enable numerical solutions. The first numerical algorithms to solve Fredholm boundary integral equations were implemented by Jaswon and Symm [1977].

In general, interactions between the elements - finite elements as well as boundary elements - are stored in a matrix format. The finite element method will lead to big sparse matrices in contrast to the boundary element method which will lead to smaller dense matrices. Solving a linear equation system using these matrices results in a valid solution of the defined problem.

3.2 Linear differential operators

Linear differential operators are an essential part of the boundary element method. They are used to define the boundary integral equations introduced in section 3.8. Linear differential operators are

defined as

$$\mathcal{L}u(\boldsymbol{x}) = \sum_{i \in \mathbb{N}} \alpha_i \frac{\partial^i u}{\partial \boldsymbol{x}^i} \tag{3.2.1}$$

with $\alpha_i \in \mathbb{R}$ For example, one linear differential operator may be the Laplace operator

$$\mathcal{L}u(\boldsymbol{x}) = \Delta u(\boldsymbol{x}) = \frac{\partial^2 u(\boldsymbol{x})}{\partial \boldsymbol{x}^2} \tag{3.2.2}$$

or for three dimensions

$$\mathcal{L}u(\boldsymbol{x}) = \Delta u(\boldsymbol{x}) = \frac{\partial^2 u(\boldsymbol{x})}{\partial x_x^2} + \frac{\partial^2 u(\boldsymbol{x})}{\partial x_y^2} + \frac{\partial^2 u(\boldsymbol{x})}{\partial x_z^2} \tag{3.2.3}$$

3.3 Gauss's divergence theorem

Gauss divergence theorem states that the sum of the derivative of all sources and sinks inside a domain Ω yields the sum of all fluxes through $\Gamma = \partial\Omega$, the boundary of Ω (Figure 3.1).

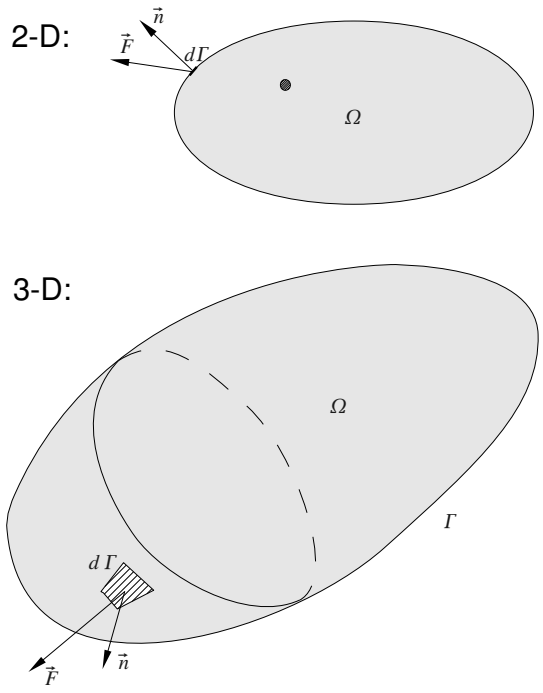


Figure 3.1: Gauss divergence theorem

$$\int_{\Omega} (\nabla \boldsymbol{F}) \mathrm{d}\Omega = \oint_{\partial\Omega} \boldsymbol{n}\boldsymbol{F} \mathrm{d}\Gamma \tag{3.3.1}$$

F denotes a field at an arbitrary surface point and the orientation of the unit normal field $\boldsymbol{n}$ is pointing outwards of the boundary Γ.

3.4 Elliptic boundary value problems

Each elliptic boundary value problem will be formulated in an open domain, denoted as Ω, where Ω is assumed to be bounded. In other words, the problem can be formulated inside a ball B_r with a finite radius r.

$$||\boldsymbol{x}|| < r \;\Leftrightarrow\; \Omega \subset B_r \tag{3.4.1}$$

$\partial\Omega$ will be defined as the surface of Ω, denoted as Γ. It is closed, since Ω is bound [Rüberg, 2008]. A unit normal vector on Γ is marked as $\boldsymbol{n}$ and is orientated outwards from Ω. A elliptic boundary value problem can be defined as a linear differential operator as introduced in section 3.2.

$$\mathcal{L}u(\boldsymbol{x}) = f(\boldsymbol{x}) \tag{3.4.2}$$

$f(\boldsymbol{x})$ is the so-called *source term* [Gaul et al., 2003] and can be a vector, in for instance elastodynamics, or a scalar, in for instance potential fields, depending on the situation. For static problems, to which this thesis is limited, this source term is equal zero, $f(\boldsymbol{x}) \equiv 0$. $\mathcal{L}$ is a linear differential operator with constant coefficients through Ω. As an example, assuming $\mathcal{L} = -\Delta$ will lead to Poisson's equation for electrostatic fields

$$-\Delta\, u(\boldsymbol{x}) = f(\boldsymbol{x}) \tag{3.4.3}$$

When setting $f(\boldsymbol{x})$ to zero the remaining equation is the so-called *Laplace equation*. A trace operator $\mathcal{T}_r$ will now be introduced to describe boundary traces u_Γ in terms of u [Rüberg, 2008; Schwab, 2004]

$$u_\Gamma(\boldsymbol{y}) = \mathcal{T}_r u(\boldsymbol{x}) = \lim_{\boldsymbol{x}\in\Omega\to\boldsymbol{y}\in\Gamma} u(\boldsymbol{x}) \tag{3.4.4}$$

and respectively, for surface forces and fluxes a traction operator $\mathcal{T}_t$ will be introduced in order to prescribe boundary tractions in terms of u

$$p_\Gamma(\boldsymbol{y}) = \mathcal{T}_t u(\boldsymbol{x}) = \lim_{\boldsymbol{x}\in\Omega\to\boldsymbol{y}\in\Gamma} p(\boldsymbol{x}) \tag{3.4.5}$$

3.4.1 Dirichlet boundary value problems

The previously defined u_Γ describes the displacement or pressure value on the surface Γ. Assuming u_Γ is known, described by a function $\bar{u}$, contrary to the unknown p_Γ, for a closed surface $\Gamma_N = \Gamma$, then a *Dirichlet boundary value* problem is posed (Figure 3.2). This condition is also called *Dirichlet boundary condition* or *essential boundary condition*.

$$\begin{aligned} \mathcal{L}u(\boldsymbol{x}) &= f(\boldsymbol{x}) \;\; \boldsymbol{x} \in \Omega \\ u_\Gamma(\boldsymbol{y}) &= \bar{u}(\boldsymbol{y}) \;\; \boldsymbol{y} \in \Gamma \\ \Gamma_D &= \Gamma \end{aligned}$$

3.4.2 Neumann boundary value problems

Defining stresses or fluxes respectively as a condition on the whole closed surface $\Gamma_N = \Gamma$ a *Neumann boundary value problem* is given (Figure 3.3). This condition is called *Neumann boundary condition* or *natural boundary condition*. The known value p_Γ is prescribed by the function $\bar{p}$.

$$\begin{aligned} \mathcal{L}u(\boldsymbol{x}) &= f(\boldsymbol{x}) \;\; \boldsymbol{x} \in \Omega \\ p_\Gamma(\boldsymbol{y}) &= \bar{p}(\boldsymbol{y}) \;\; \boldsymbol{y} \in \Gamma \\ \Gamma_N &= \Gamma \end{aligned}$$

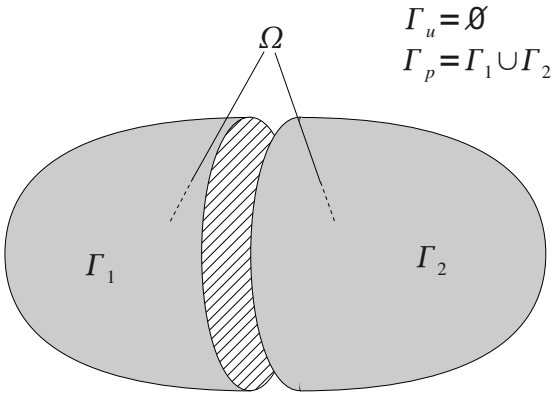


Figure 3.2: Dirichlet boundary value problem

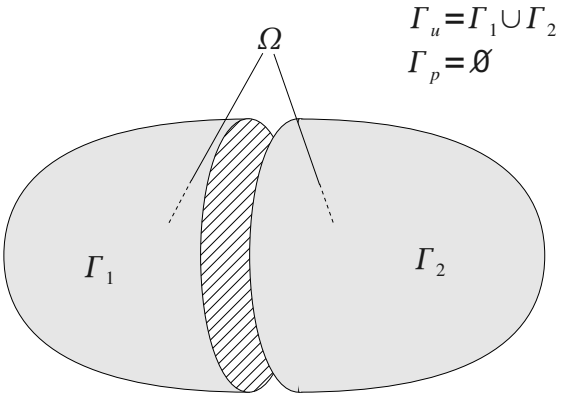


Figure 3.3: Neumann boundary value problem

3.4.3 Mixed boundary value problems

For the most commonly given situation, the mixed boundary value problem, it is assumed, that the surface Γ is split into two non-overlapping parts, a Dirichlet surface Γ_D and a Neumann surface Γ_N. On Γ_D a Dirichlet boundary condition and on Γ_N a Neumann boundary condition is applied, respectively (Figure 3.4). This problem, a so-called *Mixed boundary value problem*, is of the form

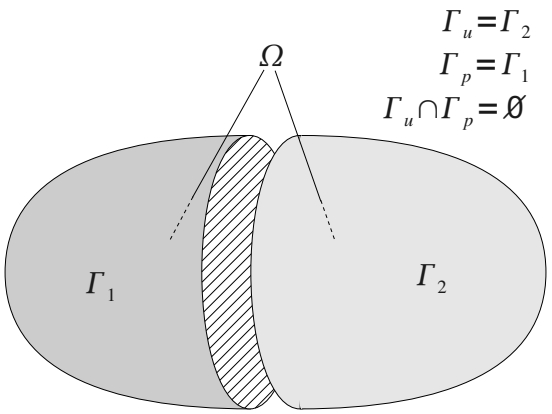


Figure 3.4: Mixed boundary value problem

$$\begin{aligned} \mathcal{L}u(\boldsymbol{x}) &= f(\boldsymbol{x}) \;\; \boldsymbol{x} \in \Omega \\ u_\Gamma(\boldsymbol{y}) &= \bar{u}(\boldsymbol{y}) \;\; \boldsymbol{y} \in \Gamma_D \\ p_\Gamma(\boldsymbol{y}) &= \bar{p}(\boldsymbol{y}) \;\; \boldsymbol{y} \in \Gamma_N \\ \Gamma_D \cup \Gamma_N &= \Gamma \\ \Gamma_D \cap \Gamma_N &= \emptyset \end{aligned}$$

Note, due to $\Gamma_D \cap \Gamma_N = \emptyset$ it is impossible to apply both, a Dirichlet and a Neumann condition, on an arbitrary surface point. Again, Γ is a closed surface.

3.5 Inverse formulation of differential equations

Consider the governing differential operator $\mathcal{L}$ weighted by a weight function w for a domain Ω is given

$$\int_\Omega \mathcal{L}(u) w \mathrm{d}\Omega = 0 \tag{3.5.1}$$

Integration by parts, recall

$$\int_\Omega f' g \mathrm{d}\Omega = \int_\Omega (fg)' \mathrm{d}\Omega - \int_\Omega f g' \mathrm{d}\Omega \tag{3.5.2}$$

yields by definition of $\mathcal{X}(u) = \int \mathcal{L}(u)$

$$\int_\Omega \mathcal{L}(u) w \mathrm{d}\Omega = \int_\Omega (\mathcal{X}(u) w)' \mathrm{d}\Omega - \int_\Omega \mathcal{X}(u) w' \mathrm{d}\Omega \tag{3.5.3}$$

Applying Gauss's divergence theorem 3.3.1 for the first integral on the right hand side leads to

$$\int_{\Omega} \mathcal{L}(u) w \mathrm{d}\Omega = \int_{\Gamma} \mathcal{X}(u) w \boldsymbol{n} \mathrm{d}\Gamma - \int_{\Omega} \mathcal{X}(u) w' \mathrm{d}\Omega \tag{3.5.4}$$

As long as the order of $\mathcal{X} > 0$ or in other words as long as $\mathcal{X}(u)$ contains derivations of u, integration by parts and Gauss's theorem have to be applied again. Equation (3.5.1), according to the order of $\mathcal{L}$ and under consideration of all derivatives of u have be eliminated in the domain integral, yielding in general [Brebbia, 1979]

$$\int_{\Omega} \mathcal{L}(u) w \mathrm{d}\Omega = \int_{\Omega} u \mathcal{L}^*(w) \mathrm{d}\Omega + \int_{\Gamma} \mathcal{G}(u) \mathcal{S}^*(w) - \mathcal{S}(u) \mathcal{G}^*(w) \mathrm{d}\Gamma \tag{3.5.5}$$

Equation (3.5.1) has been transformed, as a result of integration by parts, to a series of integrations on the boundary. $\mathcal{L}^*$ is the so-called adjoint of $\mathcal{L}$. For the case that $\mathcal{L}^* = \mathcal{L}$ the operator is called self-adjoint. By the way, when the self-adjointness occurs, then it also holds for $\mathcal{S}^* = \mathcal{S}$ and $\mathcal{G}^* = \mathcal{G}$. The operator $\mathcal{S}$ prescribes, again, the essential boundary conditions and operator $\mathcal{G}$ prescribes the natural boundary conditions, respectively. The superscript * denotes association with the w-terms, in contrast to the association with the u-terms without such a superscript. If operator $\mathcal{L}$ is assumed as self-adjoint, then equation (3.5.5) can be rewritten in the terms of $\mathcal{T}_r$ and $\mathcal{T}_t$ introduced in section 3.4 [Rüberg, 2008].

$$\int_{\Omega} \mathcal{L}(u) w \mathrm{d}\Omega + \int_{\Gamma} p_{\Gamma} \mathcal{T}_r(w) \mathrm{d}\Gamma = \int_{\Omega} u \mathcal{L}(w) \mathrm{d}\Omega + \int_{\Gamma} u_{\Gamma} \mathcal{T}_t(w) \mathrm{d}\Gamma \tag{3.5.6}$$

As an example consider the Laplace operator $\mathcal{L} = \Delta$. Following the steps described above equation (3.5.5) yields

$$\int_{\Omega} \Delta(u) w \mathrm{d}\Omega = \int_{\Omega} u \Delta(w) \mathrm{d}\Omega + \int_{\Gamma} [\nabla(u) w - u \nabla(w)] \cdot \boldsymbol{n} \, \mathrm{d}\Gamma \tag{3.5.7}$$

Due to the self-adjointness of the operator $\mathcal{L}$ the operators $\mathcal{L}^* = \mathcal{L}$, $\mathcal{T}_r = \mathcal{S} = \mathcal{S}^*$ and $\mathcal{T}_t = \mathcal{G} = \mathcal{G}^*$ can be determined following equation (3.5.6) as

$$\begin{aligned} \mathcal{L}^*(u) &= \Delta u \\ \mathcal{T}_r(u) &= u \\ \mathcal{T}_t(u) &= \nabla u \cdot \boldsymbol{n} \end{aligned} \tag{3.5.8}$$

3.6 Fundamental solutions

Each differential operator is subjected to a *fundamental solution*. Solving the equation means this fundamental solution u^* has to be found. One possibility to obtain the fundamental solution is solving the equation with a singularity of $\mathcal{L}u(x)$ at ξ

$$\mathcal{L}u^*(x) = -\delta(\xi - x) = \begin{cases} -\infty & (x = \xi) \\ 0 & (x \neq \xi) \end{cases} \tag{3.6.1}$$

by integration. $\delta(x)$ is the so-called *Dirac impulse* (Appendix B). The solution of equation 3.6.1 is assumed to be symmetric around the point ξ.
As an example consider the Laplace equation in three dimensions

$$\mathcal{L}u(\boldsymbol{x}) = \Delta u(\boldsymbol{x}) = \frac{\partial^2 u(\boldsymbol{x})}{\partial x_x^2} + \frac{\partial^2 u(\boldsymbol{x})}{\partial x_y^2} + \frac{\partial^2 u(\boldsymbol{x})}{\partial x_z^2} \tag{3.6.2}$$

A fundamental solution can be obtained by solving equation

$$\frac{\partial^2 u^*(\boldsymbol{x})}{\partial x_x^2} + \frac{\partial^2 u^*(\boldsymbol{x})}{\partial x_y^2} + \frac{\partial^2 u^*(\boldsymbol{x})}{\partial x_z^2} + \delta(\xi_x - x_x, \xi_y - x_y, \xi_z - x_z) = 0 \tag{3.6.3}$$

As long as this function is symmetric around ξ the solution is also expected to be symmetric. A transformation to the polar coordinate system where

$$\begin{aligned} x_x &= r\cos(\varphi)\sin(\vartheta) \\ x_y &= r\sin(\varphi)\sin(\vartheta) \\ x_z &= r\cos(\vartheta) \end{aligned}$$

yields, since $\delta(\xi_x - x_x, \xi_y - x_y, \xi_z - x_z) = 0$ due to $r > 0$,

$$\Delta u^*(r,\varphi,\vartheta) = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial u^*}{\partial r}\right) + \frac{1}{r^2\sin\vartheta}\frac{\partial}{\partial\vartheta}\left(\sin\vartheta\frac{\partial u^*}{\partial\vartheta}\right) + \frac{1}{r^2\sin^2\vartheta}\frac{\partial^2 u}{\partial\varphi^2} \tag{3.6.4}$$

The solution is assumed to be symmetric and therefore the second and the third term have to be zero and the solution depends on r only. It can be written as

$$\Delta u^*(r) = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial u^*}{\partial r}\right) = \frac{2u^*(r) + 4ru^{*\prime}(r) + r^2u^{*\prime\prime}(r)}{r^2} = 0 \tag{3.6.5}$$

The solution of the differential equation 3.6.5 can be obtained by integration and has the form

$$u^*(r) = \frac{C_1}{r} + \frac{C_2}{r^2} \tag{3.6.6}$$

The constants C_1 and C_2 can be calculated using equation (2.1.1) and equation (3.6.3), which yields

$$\int_\Omega \Delta u^*(r)\mathrm{d}\Omega(r) = -\int_\Omega \delta(r)\mathrm{d}\Omega(r) = -1 \tag{3.6.7}$$

Ω is the volume and must contain the singularity point ξ at $r = 0$. Using a sphere as domain Ω with the radius $\varepsilon \to 0$ around ξ and applying Gauss's divergence theorem 3.3.1 the constant C_1 can be determined

$$\begin{aligned} \int_\Omega \Delta u^*(r)\mathrm{d}\Omega(r) &= \int_{\partial\Omega} \frac{\partial u^*}{\partial n}\mathrm{d}\Gamma(r) \\ &= \int_{\partial\Omega} \frac{\partial u^*}{\partial r}\mathrm{d}\Gamma(r) \\ &= \int_{\partial\Omega} \left(-\frac{2C_2}{r^3} - \frac{C_1}{r^2}\right)\mathrm{d}\Gamma(r) \\ &= \lim_{\varepsilon\to 0}\left(-\frac{2C_2}{\varepsilon^3} - \frac{C_1}{\varepsilon^2}\right)4\pi\varepsilon^2 \\ -1 &= -4\pi C_1 \\ C_1 &= \frac{1}{4\pi} \end{aligned}$$

By now equation 3.6.6 and equation 3.6.7 yields the solution of equation 3.6.3

$$u^*(\boldsymbol{x},\boldsymbol{\xi}) = \frac{1}{4\pi||\boldsymbol{\xi} - \boldsymbol{x}||} \tag{3.6.8}$$

3.7 Representation formula

Introduced in the previous section a function $u^*(x,\xi)$ can be defined, so that

$$\mathcal{L}(u^*(x,\xi)) = \mathcal{L}^*(u^*(x,\xi)) = -\delta(\xi - x) \tag{3.7.1}$$

$\mathcal{L}$ is assumed as self-adjoint. Remember equation (3.5.6), the first domain integral, the first integral on the left hand side, becomes zero due to equation (3.5.1). For the second domain integral, the first integral on the right hand side, the achieved fundamental solution is used as the weight function $w = u^*(x,\xi)$ and can be rewritten as

$$\begin{aligned} \int_\Omega \mathcal{L}(w)u\mathrm{d}\Omega &= \int_\Omega \mathcal{L}(u^*(x,\xi))u(x)\mathrm{d}\Omega \\ &= -\int_\Omega \delta(\xi - x)u(x)\mathrm{d}\Omega \end{aligned} \tag{3.7.2}$$

$$\int_\Omega \mathcal{L}(w)u\mathrm{d}\Omega = -u(\xi) \tag{3.7.3}$$

Substituting $w = u^*(x,\xi)$ in equation (3.5.6) and using equations (3.5.1) and (3.7.3) yields

$$u(\xi) = \int_\Gamma p_\Gamma \mathcal{T}_r(u^*(x,\xi))\mathrm{d}\Gamma - \int_\Gamma u_\Gamma \mathcal{T}_t(u^*(x,\xi))\mathrm{d}\Gamma \tag{3.7.4}$$

This is the so-called *representation formula* and holds for 2D and 3D problems. This equation makes it possible to determine unknown values $u(\xi)$ inside the domain $\xi \in \Omega/\Gamma$ when the boundary values, displacements and stresses, are known.

Example: As example consider again the Laplace equation. Under consideration of the results from (3.5.8), (3.4.4) and (3.4.5) equation (3.7.4) results in

$$u(\xi) = \int_\Gamma p(x)u^*(x,\xi)\mathrm{d}\Gamma - \int_\Gamma u(x)p^*(x,\xi)\mathrm{d}\Gamma \tag{3.7.5}$$

where $p^* = \nabla u^* \cdot \boldsymbol{n}$ is the fundamental solution for the flux throughout the boundary Γ. This equation in its special case is the so-called *Green's representation formula* for boundary value problems and holds for 2D and 3D value problems in the full domain Ω.

3.8 Boundary integral equation

Consider equation (3.7.4), for all $\xi \in \Omega^*$, where $\Omega^* = \mathbb{R}^d/\Omega$ is the complement of Ω, $u(\xi)$ yields zero since equation (3.7.2) yields zero. If ξ is located on the boundary Γ, then $u(\xi)$ is undefined for the moment. A evaluation can be found in Gaul et al. [2003]. For further evaluation the so-called *jump term* or *free term* $c(\xi)$ will be introduced here and $u(\xi)$ will be replaced by $c(\xi)u(\xi)$ where

$$c(\xi) = \begin{cases} 1 & \xi \in \Omega/\Gamma \\ \in (0,1) & \xi \in \Gamma \\ 0 & \xi \in \Omega^* \end{cases} \tag{3.8.1}$$

assuming $\Gamma \subset \Omega$. $c(\xi)$ can be interpreted as the fraction of $u(\xi)$, which lies inside of Ω. For smooth surfaces $c(\xi \in \Gamma) \cong \frac{1}{2}$. Including equation (3.4.4) and (3.4.5), and allowing that x lies now on the boundary Γ, leads to the *boundary integral equation*

$$\int_\Gamma p(x)\mathcal{T}_r(u^*(x,\xi))\mathrm{d}\Gamma = c(\xi)u(\xi) + \int_\Gamma u(x)\mathcal{T}_t(u^*(x,\xi))\mathrm{d}\Gamma \tag{3.8.2}$$

which is valid in $\mathbb{R}^d$. Again, the boundary element method describes the relations of physical values like stresses, strains, pressures, fluxes and others depending on a position value. The idea is now to express the boundary values by a function depending on constant parameters which fulfills the boundary conditions defined. Boundary integral equations are commonly written as follows

$$(\mathcal{G}u)(\xi) = (\mathcal{C}u + \mathcal{H}p)(\xi) \tag{3.8.3}$$

with

$$\mathcal{G}u(\xi) = \int_\Gamma p(x)\mathcal{T}_r(u^*(x,\xi))\mathrm{d}\Gamma(x) \tag{3.8.4}$$

$$\mathcal{H}p(\xi) = \int_\Gamma u(x)\mathcal{T}_t(u^*(x,\xi))\mathrm{d}\Gamma(x) \tag{3.8.5}$$

$$\mathcal{C}u(\xi) = c(\xi)u(\xi) \tag{3.8.6}$$

where $\mathcal{G}$ is the so-called *single layer potential operator* with u as the *single layer potential value* and $\mathcal{H}$ as the *double layer potential operator* with p as the *double layer potential value*, respectively. $\mathcal{C}$ is denoted as the *jump term operator* [Gaul et al., 2003] or *free term operator* [Rüberg, 2008]. $\mathcal{T}_r$ is the trace operator, and $\mathcal{T}_t$ the traction operator. The Evaluation of them is shown in section 3.5. x refers to the position vector. At special boundary positions those relations are obvious and boundary conditions can be measured and defined there.

$\mathcal{G}u_0 = u_\Gamma = \bar{g}$ essentials on Γ_D
$\mathcal{H}p_0 = p_\Gamma = \bar{p}$ naturals on Γ_N

u_0 and p_0 represent the exact solution of the differential equation. Evaluation of $\mathcal{T}_r$ and $\mathcal{T}_t$, which is shown in section 3.5, holds for most boundary element value problems. Therefore it can be written $u^* = \mathcal{T}_r u^*$ and $p^* = \mathcal{T}_t u^*$and results finally in

$$\int_\Gamma p(x)u^*(x,\xi)\mathrm{d}\Gamma(x) = c(\xi)u(\xi) + \int_\Gamma u(x)p^*(x,\xi)\mathrm{d}\Gamma(x) \tag{3.8.7}$$

3.9 Weak and strong singular integrals

For 2D and 3D boundary value problems the fundamental solutions carry a singularity at $x = \xi$. An integration over this function causes problems, since it is necessary to determine the existence of the integral at the singularity point. These integrations can be classified as *weak singular integrals* if the integrated function has a limit at the singularity point and *strong singular integrals* if the integrated function has no limit at the singularity. This integral remains then as so-called *Cauchy principle value integral*, a definition of improper integration. For a 2D or 3D boundary value problem remember the fundamental solutions of, for instance, the Poisson equation. For 2D problems, a weak singular integral, for instance, over the fundamental solution u^* which often contains the singularity term

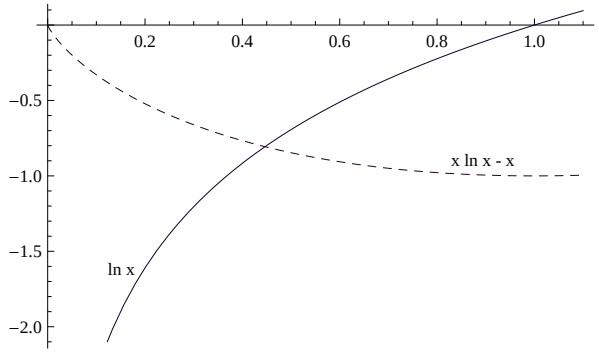


Figure 3.5: 1-D weak singular function $\ln(x)$ and its integral

$f_2(r) = \ln r$, exists (Figure 3.5)

$$f_2(r) = \ln r \tag{3.9.1}$$
$$f_2(x) = \ln x \tag{3.9.2}$$
$$F_2(x) = \int_0^1 \ln x \mathrm{d}x \tag{3.9.3}$$
$$= \left[x \ln x - x\right]_0^1 \tag{3.9.4}$$
$$= -1 \tag{3.9.5}$$

since the rule of l'Hospital applied twice yields

$$\lim_{x\to 0} x \ln x = \lim_{x\to 0} \frac{\ln x}{\frac{1}{x}} = \lim_{x\to 0} \frac{\frac{1}{x}}{-\frac{1}{x^2}} = 0 \tag{3.9.6}$$

For 3D problems the fundamental solution u^* usually contains the term $f_3(r) = 1/r$. However, this leads also to a weak singular integral in 3D since

$$f_3(r) = \frac{1}{r} \tag{3.9.7}$$
$$f_3(x,y) = \frac{1}{\sqrt{x^2+y^2}} \tag{3.9.8}$$
$$F_3(x,y) = \int_0^1 \int_0^1 \frac{1}{\sqrt{x^2+y^2}} \mathrm{d}y\mathrm{d}x \tag{3.9.9}$$
$$= \int_0^1 \left[\ln\left(y + \sqrt{x^2+y^2}\right)\right]_0^1 \mathrm{d}x \tag{3.9.10}$$
$$= \int_0^1 \left(\ln\left(1 + \sqrt{1+x^2}\right) - \ln x\right) \mathrm{d}x \tag{3.9.11}$$

The first part in the remaining 1D integral is a regular one, since $1 + \sqrt{1+x^2} > 0 | \{\forall x \geq 0\}$ and therefore the logarithm exists. The second part, $\ln x$, remains as a weak singular integral whose

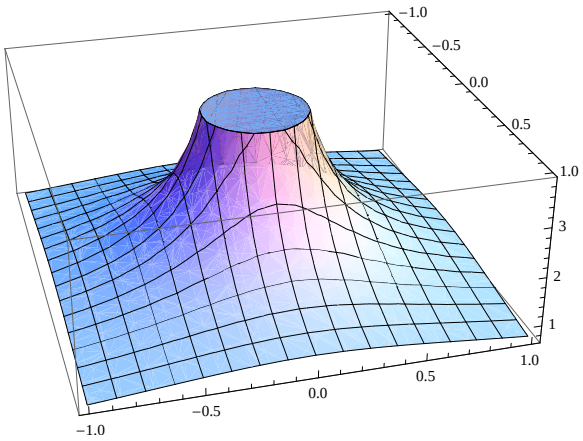


Figure 3.6: 2-D weak singular function $\frac{1}{\sqrt{x^2+y^2}}$

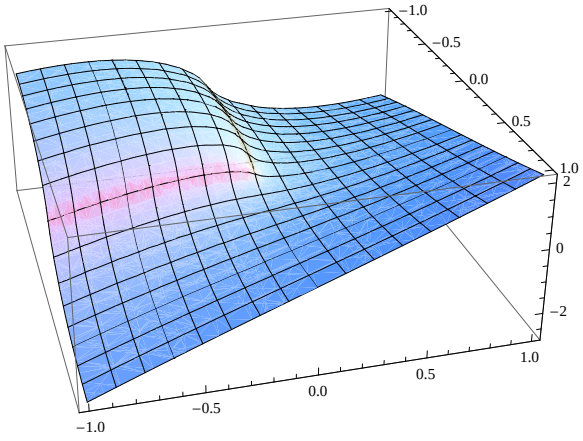


Figure 3.7: Integral of 2-D weak singular function $\frac{1}{\sqrt{x^2+y^2}}$

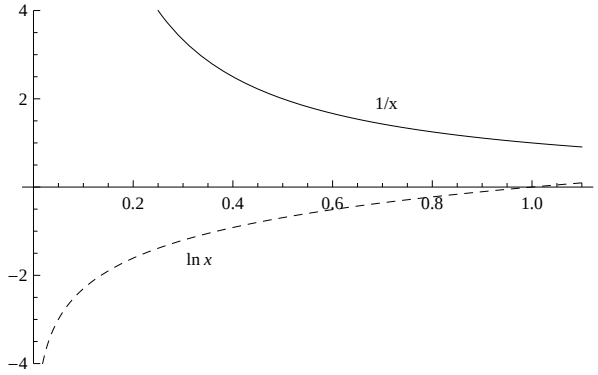


Figure 3.8: 1-D strong singular function $1/x$ and its integral

evaluation is explained above. The function and its integral are plotted in Figure 3.6 and Figure 3.7 Finally the integral can be written as

$$\int\int \frac{1}{\sqrt{x^2+y^2}}\mathrm{d}y\mathrm{d}x = y\ln\left[2\left(x+\sqrt{x^2+y^2}\right)\right] + x\ln\left[2\left(y+\sqrt{x^2+y^2}\right)\right] - x \tag{3.9.12}$$

For a strong singular integral, with an integrand for instance $g_2(r) = 1/r$ in 2-D or $g_3(r) = 1/r^2$ in 3-D problems, which occurs in the double potential layer integral, it can be rewritten as

$$g_2(r) = \frac{1}{r} \tag{3.9.13}$$

$$g_2(x) = \frac{1}{x} \tag{3.9.14}$$

$$G_2(x) = \int_{a<0}^{b>0} \frac{1}{x}\mathrm{d}r \tag{3.9.15}$$

$$= \lim_{\epsilon\to 0}\left(\int_a^{-\epsilon}\frac{1}{x}\mathrm{d}x + \int_{\epsilon}^{b}\frac{1}{x}\mathrm{d}x\right) \tag{3.9.16}$$

$$= \oint_a^b \frac{1}{x}\mathrm{d}x \tag{3.9.17}$$

where $a < \epsilon < b$. Figure 3.8 shows the strong singular function $f_2(x) = 1/x$ and its integral. It can be observed, that the integrated function contains also a singularity at $x = 0$. This is a so-called *Cauchy principle value integral*. For some of these functions an integral can be calculated for $x \geq 0$ under special conditions. Computation of those integrals can be found in Gaul et al. [2003]. Fortunately, for boundary element value problems the evaluation can be done indirectly when rigid body movement is assumed. This means the boundary conditions are Neumann ones, the forces are known and the displacements are unknown for each element, and the whole body is moved due to acting forces.

3.10 Discretization of the boundary surface

After all those mathematical aspects of the boundary element method a practical numerical implementation of the discussed equations is needed. Therefore, the domain's boundary will be subdivided

into a finite number of parts, the so-called *boundary elements* (Figure 3.9). The subscript index j

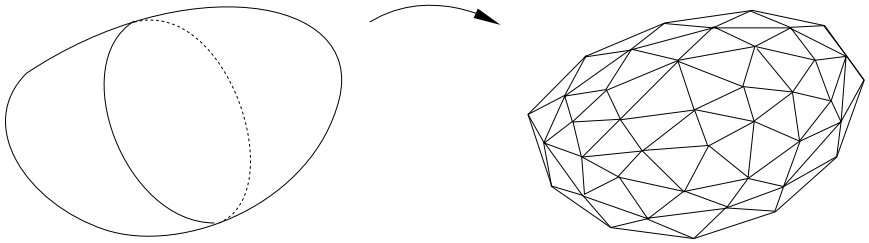


Figure 3.9: Distretized surface using triangle patches

denotes a single boundary element [Gaul et al., 2003].

$$\Gamma = \bigcup_{j=1}^{E} \Gamma_j \tag{3.10.1}$$

Remember the boundary integral equation (3.8.7) will transform after discretization to

$$\sum_j \int_{\Gamma_j} p(x)u^*(x,\xi)\mathrm{d}\Gamma_j(x) = c(\xi)u(\xi) + \sum_j \int_{\Gamma_j} u(x)p^*(x,\xi)\mathrm{d}\Gamma_j(x) \tag{3.10.2}$$

In addition the functions of the boundary values u and p have to be discretized too, since the boundary elements are now only represented by their vertices. Therefore, it is necessary to define interpolation functions between those vertices to gain values inside the element for integration. This will be explained in the next section. The dimension of the boundary is one dimension less than the domain's dimension, since the boundary is a manifold of the volume.

3.11 Interpolation and shape functions

After discretization, the boundary of an object is defined by lines in 2D and by triangles or quadrangles in 3D which represent the boundary elements. Consider one boundary element and its integral

$$\int_{\Gamma_j} u(x)p^*(x,\xi)\mathrm{d}\Gamma_j(x) \tag{3.11.1}$$

There is no interest to evaluate u at an arbitrary position on Γ_j, but instead at elected positions, so-called *nodes* on which, for instance, stresses and displacements can be applied. By the definition of interpolation functions the influence of an arbitrary position with respect to this nodes can be evaluated. An interpolation function can be supposed to be constant, linear or higher ordered enabling the influence of the element to be assigned to the nodes.

3.11.1 Order of interpolation functions

As already mentioned there are different types of how to interpolate the influence of a boundary element to the nodes. According to the interpolation function each element needs several numbers of nodes to interpolate the function finally. The number of nodes are denoted as κ for later usage. These types will be shown in the next subsections and examples are drawn in Figure 3.10, where constant interpolation for 1-D line boundary elements, constant interpolation for 2-D quadrilateral boundary elements and linear interpolation for 2-D triangular boundary elements, in this order, are shown.

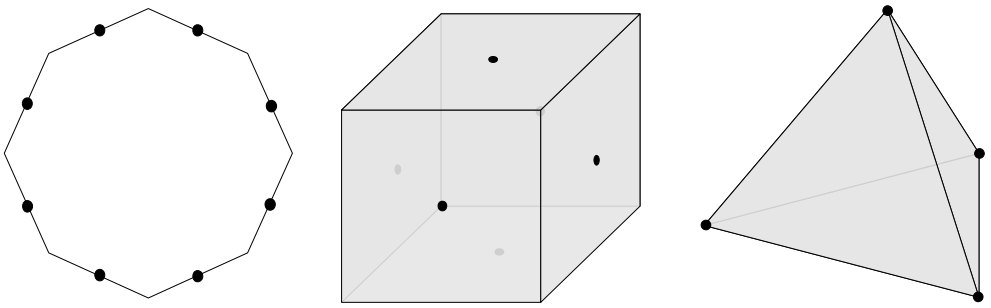


Figure 3.10: Examples for boundary nodes

Constant case

For constant boundary elements the boundary values are represented by one node located in the center of each surface element. Each element interacts with only one node, and therefore no boundary element sharing with neighboring boundary elment's nodes is needed to be considered which simplifies this case.

Linear case

For interpolations over the boundary elements in an linear way the nodes are located in each corner of each surface element. Those corners are shared by other boundary elements and needed to be taken into account. The interpolation occurs linearly between these nodes.

Quadratic case

With each level of degree in the interpolation functions the calculation becomes more an more complex, but the final result is therefore more accurate with the same number of nodes compared to cases with an lower degree.

Cubic case

For the cubic interpolation more fixed points are needed and the accuracy increases again compared to the quadratic case. However this accuracy is only needed for elements which are close to an singularity, since the boundary element values at these nodes might be strongly different.

3.11.2 Evaluation of the interpolation functions

Line boundary elements (1-D)

The following parts show how the interpolation function can be determined for 1-D boundary elements. The element will be transformed into homogeneous coordinates η where $\eta \in [-1, 1]$.

Constant case $\kappa = 1$ (Figure 3.11)

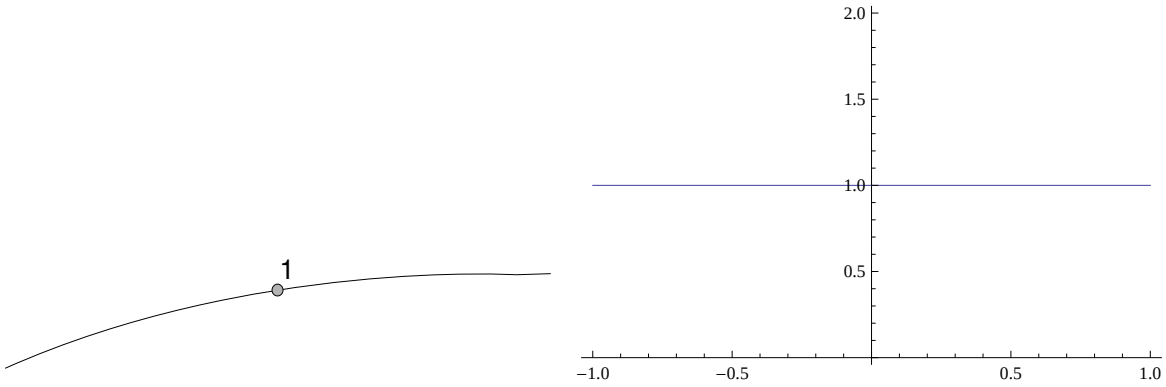


Figure 3.11: 1-D constant interpolation

$$\varphi = 1 \tag{3.11.2}$$

Linear case $\kappa = 2$ (Figure 3.12)

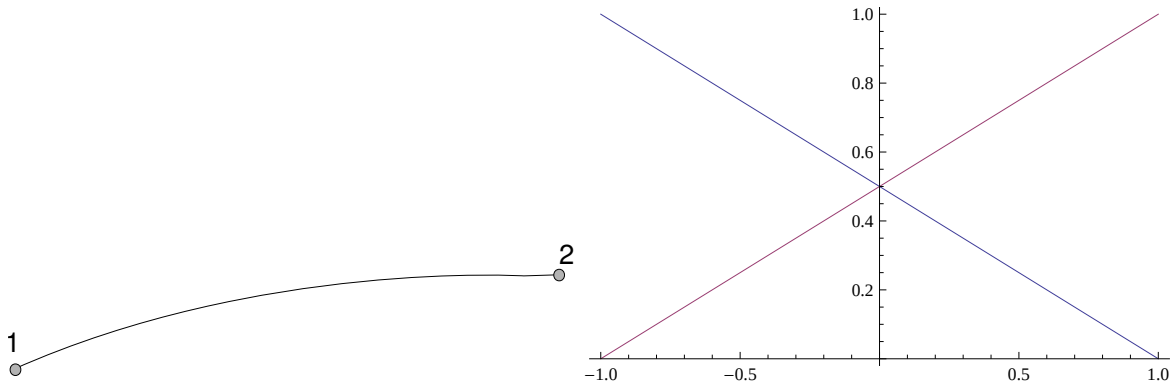


Figure 3.12: 1-D linear interpolation

$$\varphi = \begin{pmatrix} \frac{1}{2}(1-\eta) \\ \frac{1}{2}(1+\eta) \end{pmatrix} \tag{3.11.3}$$

Quadratic case $\kappa = 3$ (Figure 3.13)

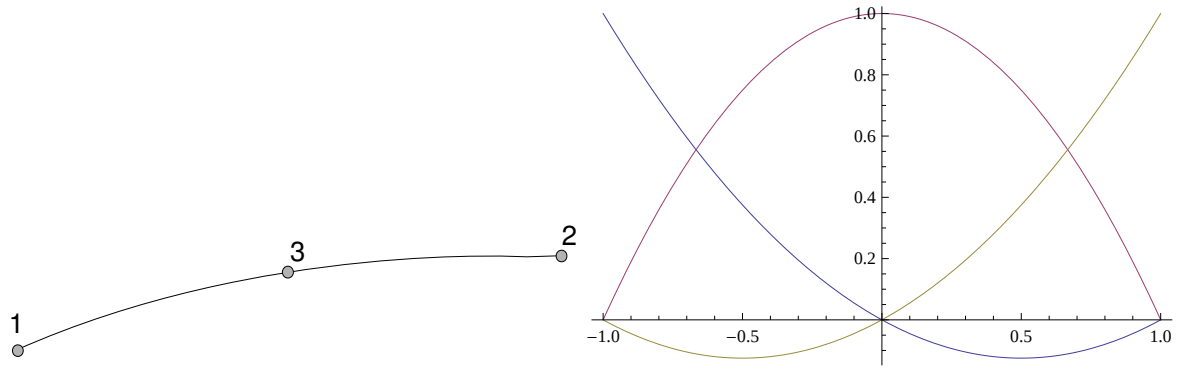


Figure 3.13: 1-D quadratic interpolation

$$\varphi = \begin{pmatrix} \frac{1}{2}\eta(\eta-1) \\ \frac{1}{2}\eta(\eta+1) \\ (1+\eta)(1-\eta) \end{pmatrix} \tag{3.11.4}$$

Cubic case $\kappa = 4$ (Figure 3.14)

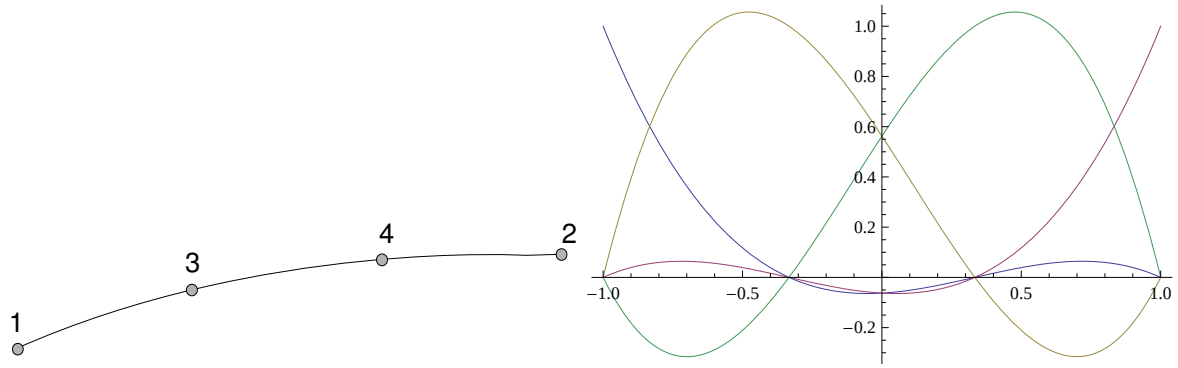


Figure 3.14: 1-D cubic interpolation

$$\varphi = \begin{pmatrix} \frac{1}{16}(1-\eta)(1-9\eta^2) \\ \frac{1}{16}(1+\eta)(1-9\eta^2) \\ \frac{9}{16}(1-\eta^2)(1-3\eta) \\ \frac{9}{16}(1-\eta^2)(1+3\eta) \end{pmatrix} \tag{3.11.5}$$

Quadrilateral boundary elements (2-D)

Here it is shown how to apply the interpolation functions for 2-D quadrilateral boundary elements. The homogeneous coordinate system for $[\eta_1, \eta_2]$ ranges inside the square $[\eta_1, \eta_2] \in [[-1, 1], [-1, 1]]$.

Constant case $\kappa = 1$ (Figure 3.15)

$$\varphi = 1 \tag{3.11.6}$$

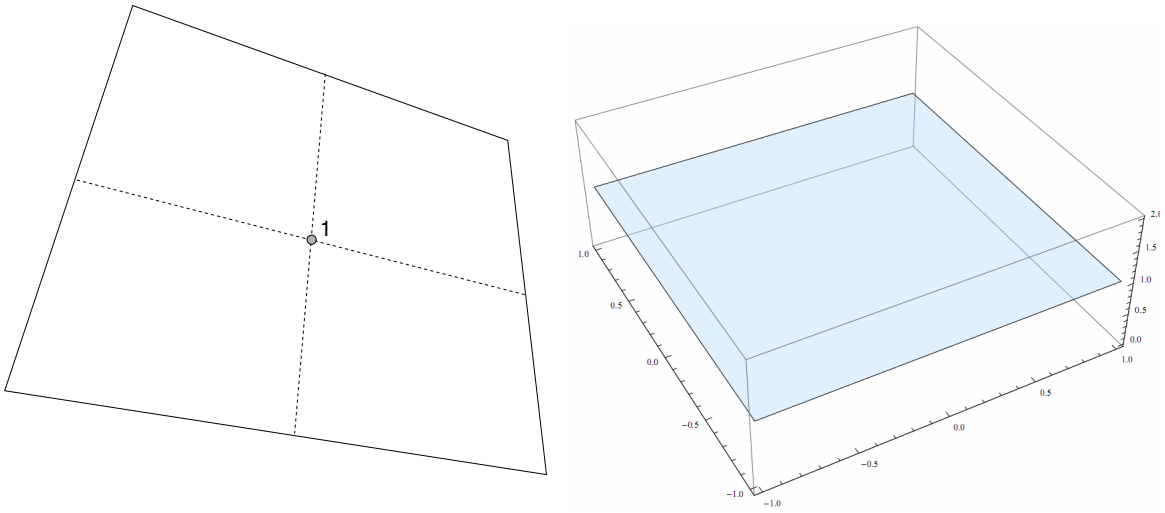


Figure 3.15: 2-D quadrilateral constant interpolation

Linear case $\kappa = 4$ (Figure 3.16)

$$\varphi = \begin{pmatrix} \frac{1}{4}(1-\eta_1)(1-\eta_2) \\ \frac{1}{4}(1-\eta_1)(1+\eta_2) \\ \frac{1}{4}(1+\eta_1)(1+\eta_2) \\ \frac{1}{4}(1+\eta_1)(1-\eta_2) \end{pmatrix} \tag{3.11.7}$$

Quadratic case $\kappa = 9$ (Figure 3.17)

$$\varphi = \begin{pmatrix} \frac{1}{4}\eta_1\eta_2(1-\eta_1)(1-\eta_2) \\ \frac{1}{4}\eta_1\eta_2(1-\eta_1)(1+\eta_2) \\ \frac{1}{4}\eta_1\eta_2(1+\eta_1)(1+\eta_2) \\ \frac{1}{4}\eta_1\eta_2(1+\eta_1)(1-\eta_2) \\ \frac{1}{2}\eta_2(\eta_2-1)(1-\eta_1^2) \\ \frac{1}{2}\eta_2(\eta_2+1)(1-\eta_1^2) \\ \frac{1}{2}\eta_1(\eta_1-1)(1-\eta_2^2) \\ \frac{1}{2}\eta_1(\eta_1+1)(1-\eta_2^2) \\ (1-\eta_1^2)(1-\eta_2^2) \end{pmatrix} \tag{3.11.8}$$

Cubic case $\kappa = 16$ (Figure 3.18)

$$\varphi = \begin{pmatrix} \frac{1}{256}(1\pm\eta_1)(1\pm\eta_2)(1-9\eta_1^2)(1-9\eta_2^2) \\ \frac{-9}{256}(1-9\eta_1^2)(1\pm\eta_1)(1-\eta_2^2)(1\pm 3\eta_2) \\ \frac{-9}{256}(1-\eta_1^2)(1\pm 3\eta_1)(1-9\eta_2^2)(1\pm\eta_2) \\ \frac{81}{256}(1-\eta_1^2)(1-\eta_2^2)(1\pm 3\eta_1)(1\pm 3\eta_2) \end{pmatrix} \tag{3.11.9}$$

Triangular boundary elements (2-D)

Finally, for triangular elements the following prescribes the interpolation functions where the homogeneous coordinate system is represented by its *barycentric coordinates* $[\lambda_1, \lambda_2, \lambda_3]$, which are

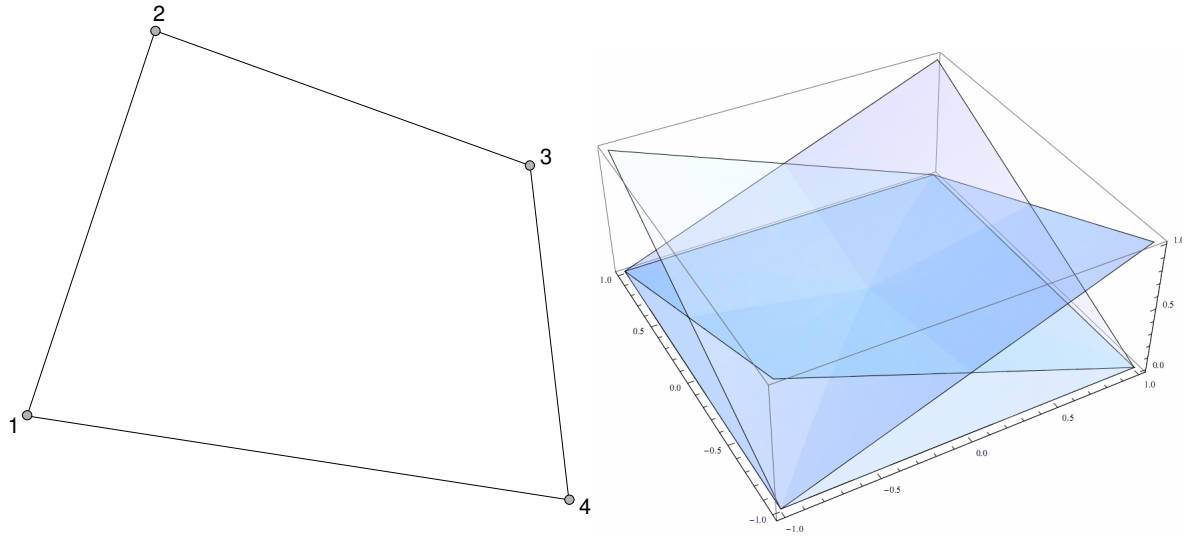


Figure 3.16: 2-D quadrilateral linear interpolation

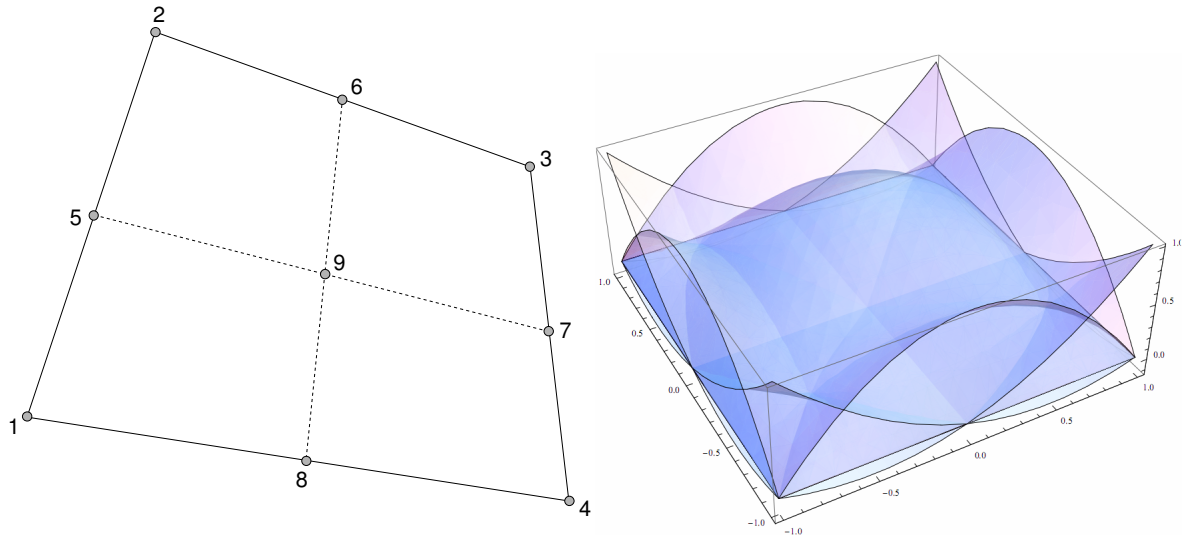


Figure 3.17: 2-D quadrilateral quadratic interpolation

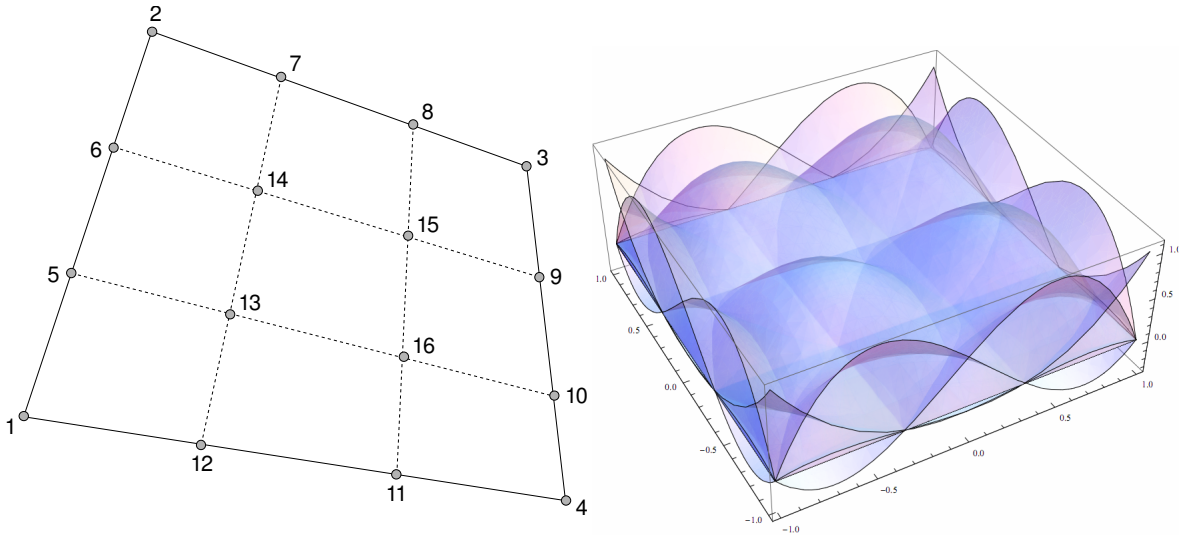


Figure 3.18: 2-D quadrilateral cubic interpolation

explained in the Appendix C of this paper

Constant case $\kappa = 1$ (Figure 3.19)

$$\varphi = 1 \tag{3.11.10}$$

Linear case $\kappa = 3$ (Figure 3.20)

$$\varphi = \begin{pmatrix} \eta_1 \\ \eta_2 \\ \eta_3 \end{pmatrix} \tag{3.11.11}$$

Quadratic case $\kappa = 6$ (Figure 3.21)

$$\varphi = \begin{pmatrix} (2\eta_1 - 1)\eta_1 \\ (2\eta_2 - 1)\eta_2 \\ (2\eta_3 - 1)\eta_3 \\ 4\eta_1\eta_2 \\ 4\eta_1\eta_3 \\ 4\eta_2\eta_3 \end{pmatrix} \tag{3.11.12}$$

Cubic case $\kappa = 10$ (Figure 3.22)

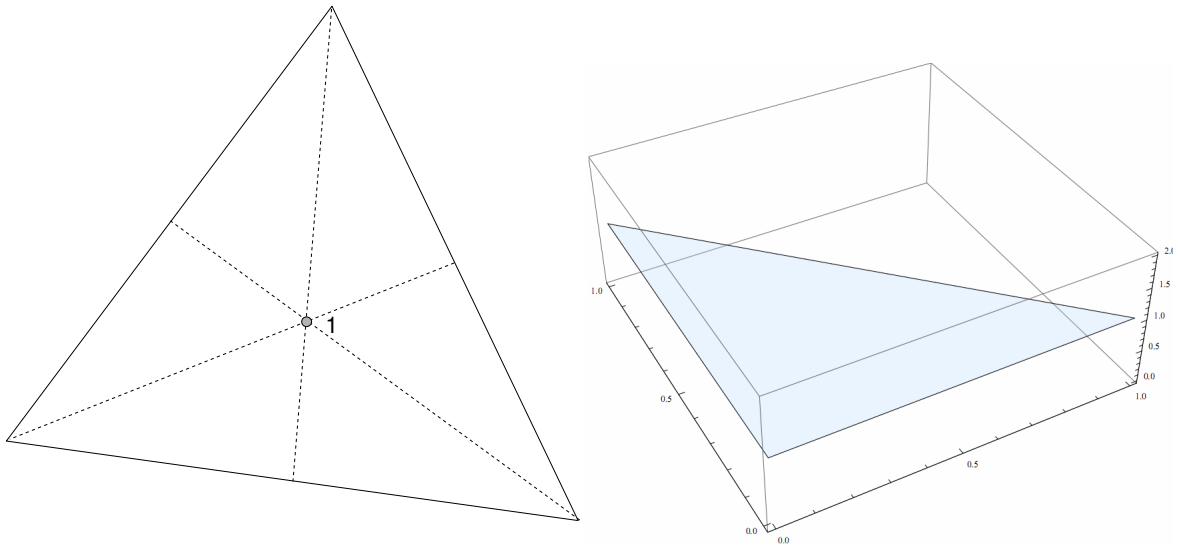


Figure 3.19: 2-D triangular constant interpolation

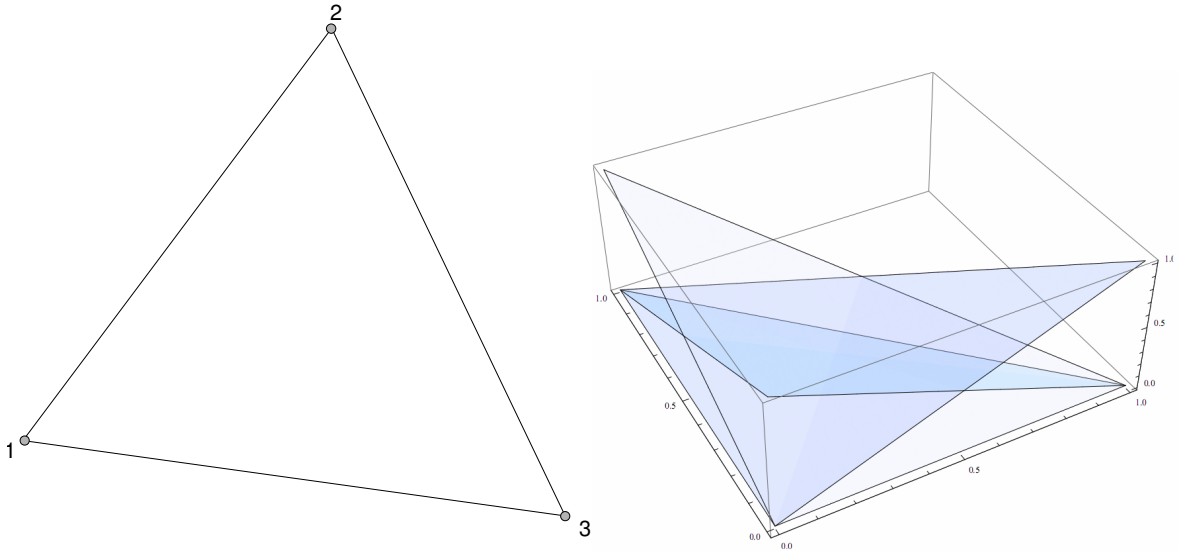


Figure 3.20: 2-D triangular linear interpolation

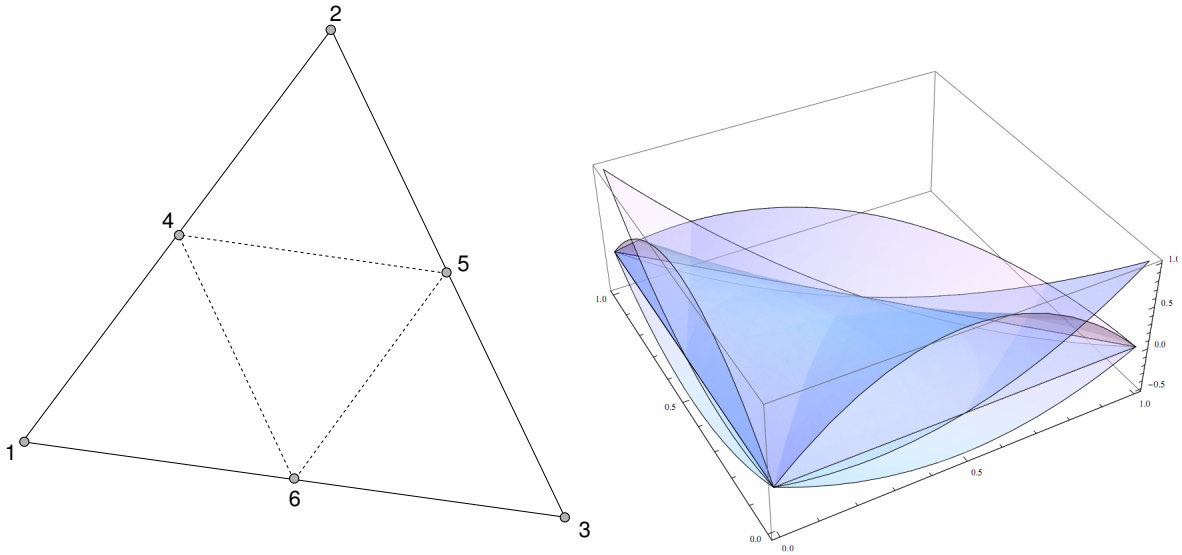


Figure 3.21: 2-D triangular quadratic interpolation

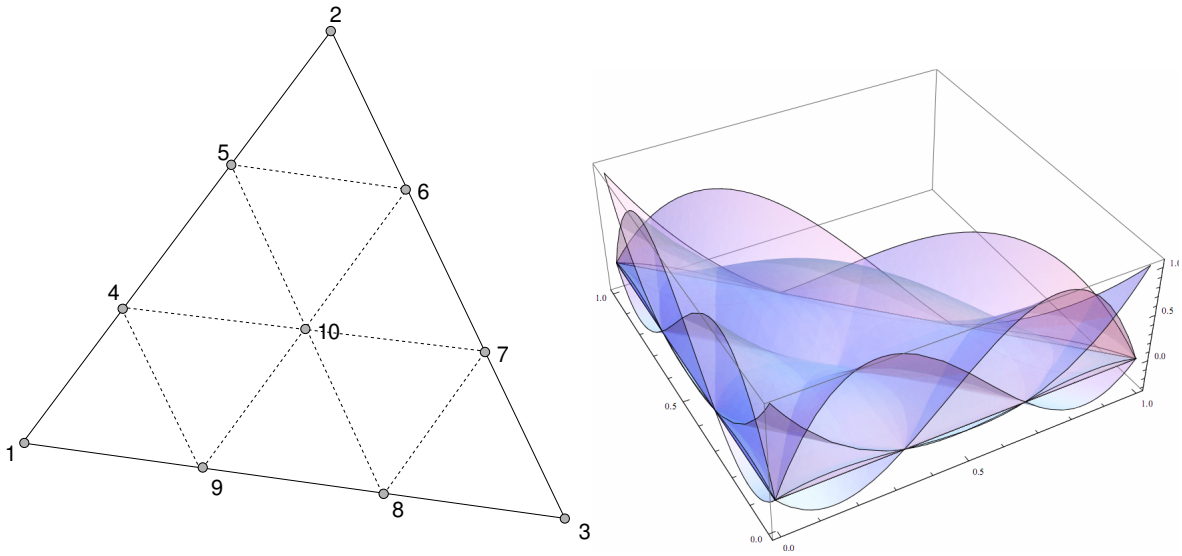


Figure 3.22: 2-D triangular cubic interpolation

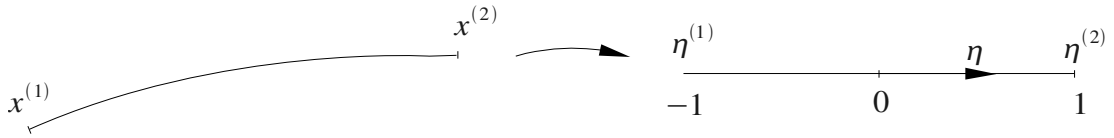


Figure 3.23: Line transformation into homogeneous coordinates

$$\varphi = \begin{pmatrix} \frac{1}{2}\eta_1(3\eta_1-2)(3\eta_1-1) \\ \frac{1}{2}\eta_2(3\eta_2-2)(3\eta_2-1) \\ \frac{1}{2}\eta_3(3\eta_3-2)(3\eta_3-1) \\ \frac{9}{2}\eta_1\eta_2(3\eta_1-1) \\ \frac{9}{2}\eta_1\eta_2(3\eta_2-1) \\ \frac{9}{2}\eta_1\eta_3(3\eta_1-1) \\ \frac{9}{2}\eta_1\eta_3(3\eta_3-1) \\ \frac{9}{2}\eta_2\eta_3(3\eta_2-1) \\ \frac{9}{2}\eta_2\eta_3(3\eta_3-1) \\ 27\eta_1\eta_2\eta_3 \end{pmatrix} \tag{3.11.13}$$

3.11.3 Transformation into homogeneous coordinates system

1D line elements

A surface point on the boundary element can be expressed in terms of η by

$$\boldsymbol{x} = \frac{1-\eta}{2}\boldsymbol{x}^{(1)} + \frac{1+\eta}{2}\boldsymbol{x}^{(2)} \tag{3.11.14}$$

The local homogeneous coordinate η can be expressed by

$$\eta = \frac{2(\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(2)}) \cdot (\boldsymbol{x}^{(1)} - \boldsymbol{x})}{||\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(2)}||^2} - 1 \tag{3.11.15}$$

where x is the global coordinate and $x^{(i)}$ the corners of the line element. (Figure 3.23)

2D quadrilateral elements

For 2D quadrilateral boundary elements a transformation to homogeneous coordinates η can be approximated by solving

$$\boldsymbol{x} = \boldsymbol{a}_1 + \boldsymbol{a}_2\eta_1 + \boldsymbol{a}_3\eta_2 + \boldsymbol{a}_4\eta_1\eta_2 \tag{3.11.16}$$

for the known corner values

$$\begin{aligned} \boldsymbol{\eta}(\boldsymbol{x}^{(1)}) &= (-1,-1)^{\mathrm{T}} \\ \boldsymbol{\eta}(\boldsymbol{x}^{(2)}) &= (-1,1)^{\mathrm{T}} \\ \boldsymbol{\eta}(\boldsymbol{x}^{(3)}) &= (1,-1)^{\mathrm{T}} \\ \boldsymbol{\eta}(\boldsymbol{x}^{(4)}) &= (1,1)^{\mathrm{T}} \end{aligned}$$

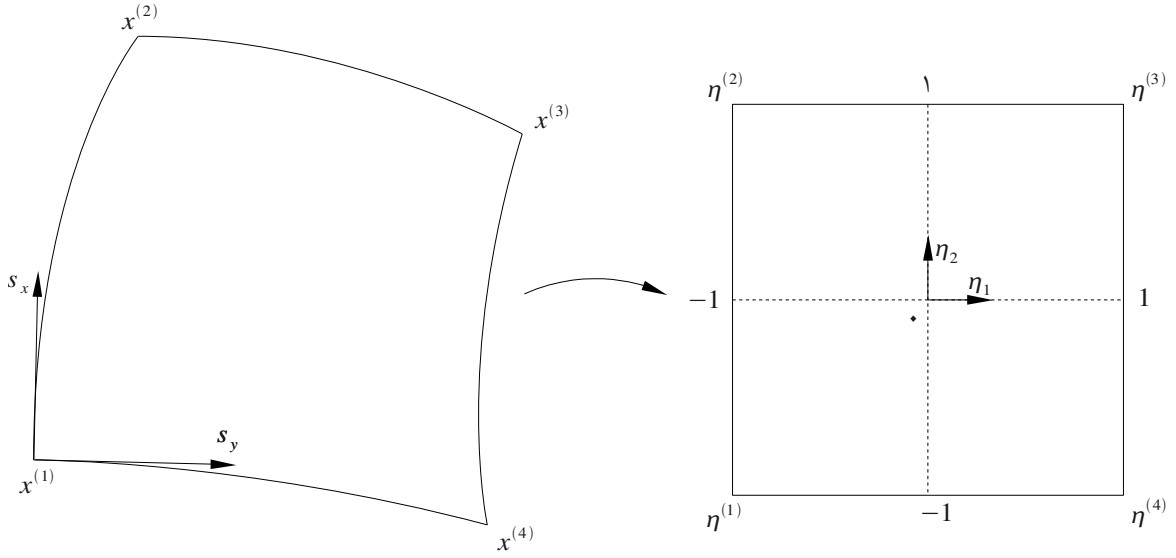


Figure 3.24: Quad transformation into homogeneous coordinates

which results in

$$\begin{aligned}
\boldsymbol{a}_1 &= \frac{1}{4}(\boldsymbol{x}^{(1)} + \boldsymbol{x}^{(2)} + \boldsymbol{x}^{(3)} + \boldsymbol{x}^{(4)}) \\
\boldsymbol{a}_2 &= \frac{1}{4}(-\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(2)} + \boldsymbol{x}^{(3)} + \boldsymbol{x}^{(4)}) \\
\boldsymbol{a}_3 &= \frac{1}{4}(-\boldsymbol{x}^{(1)} + \boldsymbol{x}^{(2)} - \boldsymbol{x}^{(3)} + \boldsymbol{x}^{(4)}) \\
\boldsymbol{a}_4 &= \frac{1}{4}(\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(2)} - \boldsymbol{x}^{(3)} + \boldsymbol{x}^{(4)})
\end{aligned}$$

2D triangular elements

Any point on a triangle can be expressed in the global coordinate system by its barycentric coordinates (Appendix C) using

$$\boldsymbol{x} = \lambda_1 \boldsymbol{x}^{(1)} + \lambda_2 \boldsymbol{x}^{(2)} + \lambda_3 \boldsymbol{x}^{(3)} \tag{3.11.17}$$

where $\boldsymbol{x}^{(i)}$ denotes the coordinates of the i^{th} corner. Choosing the commonly used local corner values $\boldsymbol{\lambda}^{(1)} = (1,0,0)$, $\boldsymbol{\lambda}^{(2)} = (0,1,0)$ and $\boldsymbol{\lambda}^{(3)} = (0,0,1)$ λ can be expressed by

$$\boldsymbol{\lambda} = \frac{1}{||\boldsymbol{n}||^2} \begin{pmatrix} (\boldsymbol{x}^{(3)} - \boldsymbol{x}) \cdot \boldsymbol{v}^{(1)} \\ (\boldsymbol{x}^{(1)} - \boldsymbol{x}) \cdot \boldsymbol{v}^{(2)} \\ (\boldsymbol{x}^{(2)} - \boldsymbol{x}) \cdot \boldsymbol{v}^{(3)} \end{pmatrix} \tag{3.11.18}$$

with the normal vector $\boldsymbol{n}$ and the additional help vectors $\boldsymbol{v}^{(i)}$

$$\boldsymbol{n} = (\boldsymbol{x}^{(2)} - \boldsymbol{x}^{(3)}) \times (\boldsymbol{x}^{(3)} - \boldsymbol{x}^{(1)}) \tag{3.11.19}$$

$$\boldsymbol{v}^{(1)} = \boldsymbol{n} \times (\boldsymbol{x}^{(2)} - \boldsymbol{x}^{(3)}) \tag{3.11.20}$$

$$\boldsymbol{v}^{(2)} = \boldsymbol{n} \times (\boldsymbol{x}^{(3)} - \boldsymbol{x}^{(1)}) \tag{3.11.21}$$

$$\boldsymbol{v}^{(3)} = \boldsymbol{n} \times (\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(2)}) \tag{3.11.22}$$

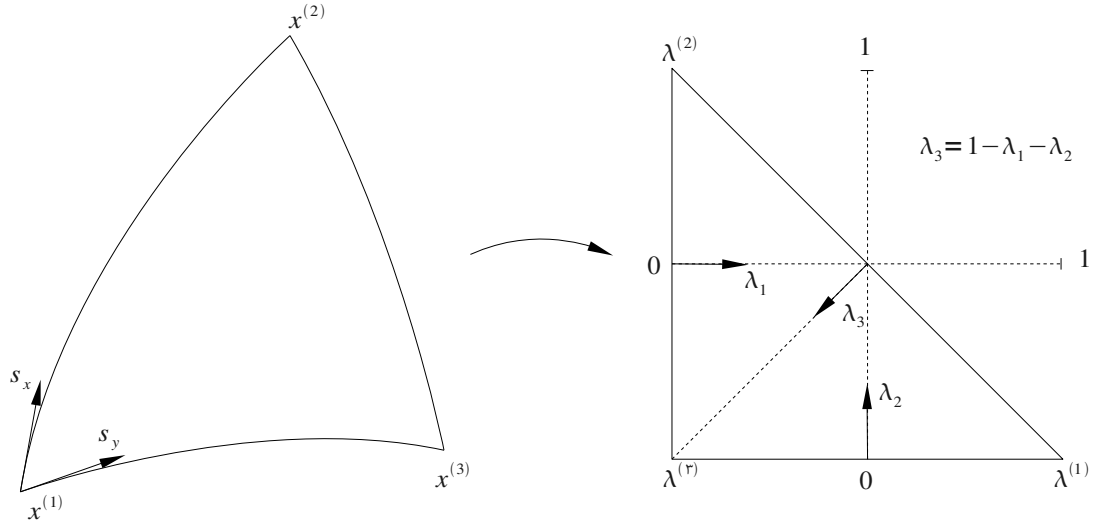


Figure 3.25: Triangle transformation into homogeneous coordinates

3.11.4 Boundary integral equation and interpolation functions

In Section 3.8 the boundary integral equation was introduced as

$$\int_{\Gamma} p(x)u^*(x,\xi)\mathrm{d}\Gamma(x) = c(\xi)u(\xi) + \int_{\Gamma} u(x)p^*(x,\xi)\mathrm{d}\Gamma(x) \tag{3.11.23}$$

The nodal values u and p can be interpolated over the boundary elements by using the interpolation functions as

$$u = \Phi^T u^n \tag{3.11.24}$$
$$p = \Phi^T p^n \tag{3.11.25}$$

where u^n and p^n denotes the nodal displacement and tractions of the surface mesh. The equation (3.11.23) can now be discretized by using this interpolation functions and can be written as

$$\sum_j \int_{\Gamma_j} u^*(x,\xi)\Phi_j^T \mathrm{d}\Gamma_j p^n = c(\xi)u(\xi) + \sum_j \int_{\Gamma_j} p^*(x,\xi)\Phi_j^T \mathrm{d}\Gamma_j u^n \tag{3.11.26}$$

The sum goes over all surface elements Γ_j and the equation is usually solved by numerical iterative solvers. Φ is expressed in one of the previously introduced homogenous coordinates systems of the surface element j.

3.12 Numerical evaluation of coefficient integrals

3.12.1 Gauss quadrature formula

The Gauss quadrature formula describes an algorithm for solving integral equations numerically by estimating the integrand with a polynomial function.

$$I = \int f(x)\mathrm{d}x \cong \int \sum_{i=1}^{n} c_i x^i \mathrm{d}x \tag{3.12.1}$$

Compared to the Simpson rule for numeric integration [Kreyszig, 2005] it will be shown, that less fixed-points are necessary to estimate the integrand. Using Simpson's rule, for a n-ordered polynomial n fixed-points are needed to approximate the function exactly. Fixing the integration domain and adding weight factors will lead to Gauss quadrature rule

$$I = \int_{-1}^{1} f(s)\mathrm{d}s \cong \sum_{i=1}^{n} f(s_i) w_i \tag{3.12.2}$$

x_i are the so-called *Gauss points* and w_i are the corresponding *weight factors*. For an arbitrary interval $[a, b]$ the integration boundaries have to be transformed first to the reference interval $[-1, 1]$. The Gauss points and weight factors can be evaluated by setting up, depending on the degree of accuracy n, $2n$ numbers of equations $f(x^j)|j \in \mathbb{N} \wedge j < 2n$. For example, this yields for $n = 2$

$$f(x)|x = 1, s, s^2, s^3 \tag{3.12.3}$$

$$\begin{aligned}
\int_{-1}^{1} \mathrm{d}s &= 2 = & w_1 + w_2 \\
\int_{-1}^{1} s\mathrm{d}s &= 0 = & w_1 s_1 + w_2 s_2 \\
\int_{-1}^{1} s^2\mathrm{d}s &= \tfrac{2}{3} = & w_1 s_1^2 + w_2 s_2^2 \\
\int_{-1}^{1} s^3\mathrm{d}s &= 0 = & w_1 s_1^3 + w_2 s_2^3
\end{aligned}$$

Solving this system of non-linear equations gives

$$s_{1,2} = \pm\frac{1}{\sqrt{3}} \text{ and } w_{1,2} = 1 \tag{3.12.4}$$

The values for the Gauss points and their weights are listed in table 3.1 for an order $N \leq 5$. For two or more dimensional integrations, this formula can be applied analogously, finding Gauss points ξ_i and η_j and corresponding weight factors w_i and w_j, so that [Gaul et al., 2003]

$$I = \int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta)\mathrm{d}\xi\mathrm{d}\eta = \sum_{i=1}^{n_1}\sum_{j=1}^{n_2} f(\xi_i,\eta_j) w_i w_j \tag{3.12.5}$$

N	s_i	w_i
2	± 0.57735	1.0
3	0.0	0.888889
	± 0.774597	0.555556
4	± 0.861136	0.347855
	± 0.339981	0.652145
5	0.0	0.568889
	± 0.538469	0.478629
	± 0.90618	0.236927

Table 3.1: Gauss points and weights values

3.12.2 Integration over boundary elements

After discretization of the surface into discrete boundary elements, which was explained in section 3.10, each element domain has to be transformed into its reference (square) coordinate system $[-1, 1]$ ($[[-1, 1], [-1, 1]]$) before applying the Gauss quadrature formula.

Boundary elements with weak singular integrals

One point needs further attention. A problem occurs if the Gauss integration algorithm for the weak and strong singular integrals is used, because it is very crude to estimate a polynomial function for functions containing a singularity. For 1-D boundaries the Gauss rule can be adapted to evaluate weak singular integrals. Consider Gaul et al. [2003] then

$$\int_0^1 g(s)\varphi(s)\mathrm{d}s \approx \sum_{i=1}^{n} g(s_i)w_i \tag{3.12.6}$$

takes already the singularity into account at $x = 0$ by choosing $\varphi(x) = \ln(x)$. Evaluation of the Gauss points works analogously to equation (3.12.3) and yields for $n = 2$

$$s_{1,2} = \frac{15 \pm \sqrt{106}}{42} \text{ and } w_{1,2} = \frac{-212 \pm 9\sqrt{106}}{424}. \tag{3.12.7}$$

For 2D boundary elements the evaluation of surfaces containing a weak singularity can be achieved by an additional special regularizing transformation into polar coordinates, which results in a degeneration of the singularity.

$$x = r\cos\vartheta, \qquad y = r\sin\vartheta, \tag{3.12.8}$$

$$r = \sqrt{x^2 + y^2}, \qquad J = \left|\frac{\partial(x, y)}{\partial(r, \vartheta)}\right| = r \tag{3.12.9}$$

The Jacobian determinant assigns by this transformation to $J = r$ and vanishes therefore at the point of singularity due to $\lim_{r\to 0} J = 0$. The integral reduces to

$$\iint \frac{f(x, y)}{r}\mathrm{d}x\mathrm{d}y = \iint f(r, \vartheta)\mathrm{d}r\mathrm{d}\vartheta \tag{3.12.10}$$

Figure 3.26 shows this transformation. Another solution can be found by the *Lachat-Watson Trans-*

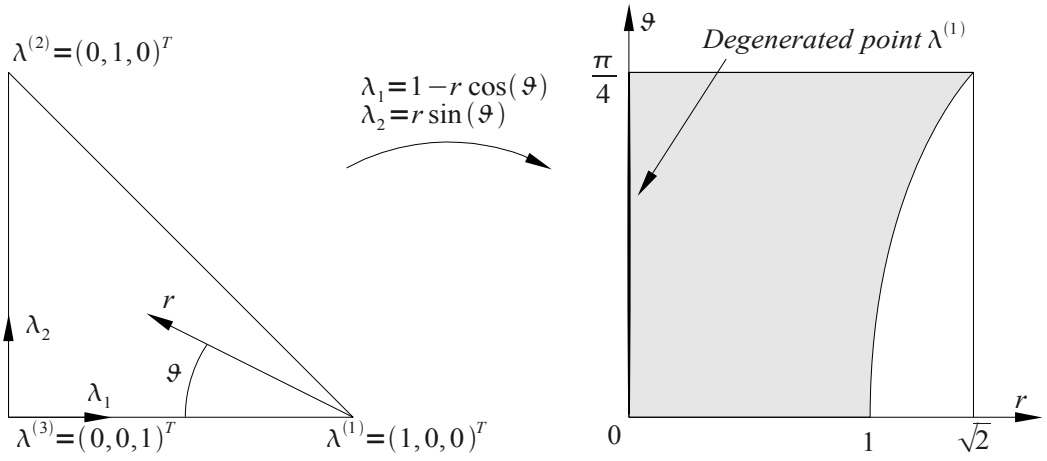


Figure 3.26: Regularization with polar coordinates

formation which regularizing transformation follows Gaul et al. [2003] using

$$x = 1 - u, \;\; y = uv \tag{3.12.11}$$

with the Jacobi determinant

$$J = \left| \frac{\partial(x, y)}{\partial(u, v)} \right| = \frac{1}{2}(1 - u) \tag{3.12.12}$$

Now, by placing the singularity point at $\lambda^{(1)}$ the Jacobi determinant disappears at $u = 1 \Leftrightarrow x = 0$ and will regularize the singularity and the Gauss quadrature formula can be applied normally. Figure 3.27 shows this regularization.

Integral domain adaptation for triangular elements

This section shows how an adaptation for triangular boundary elements can be obtained, whose integration domain is not the reference square coordinate system. For triangular elements the homogeneous coordinate system is defined by the barycentric coordinate system, which differs from the needed reference square coordinate system. Before applying the the Gauss quadrature rule this domain has to be corrected and an integral expression adaptation to the reference square coordinate system is needed. Commonly an integration over a triangular surface can be expressed as

$$\int_{\Gamma} f(x) \mathrm{d}\Gamma = \int_{0}^{1} \int_{0}^{1-\lambda_2} f[\boldsymbol{x}(\lambda_1, \lambda_2)] \left| \frac{\partial \boldsymbol{x}}{\partial \lambda_1} \times \frac{\partial \boldsymbol{x}}{\partial \lambda_2} \right| \mathrm{d}\lambda_1 \mathrm{d}\lambda_2 \tag{3.12.13}$$

A transformation of the integral boundaries to the correct reference square coordinate system can be achieved by a variation of the Lachat-Watson transformation explained above [Rathod et al., 2004],

$$\int_{\Gamma} f(x) \mathrm{d}\Gamma = \int_{-1}^{1} \int_{-1}^{1} f[\boldsymbol{x}(\eta_1, \eta_2)] \left| \frac{\partial \boldsymbol{\lambda}}{\partial \boldsymbol{\eta}} \right| \left| \frac{\partial \boldsymbol{x}}{\partial \lambda_1} \times \frac{\partial \boldsymbol{x}}{\partial \lambda_2} \right| \mathrm{d}\eta_1 \mathrm{d}\eta_2 \tag{3.12.14}$$

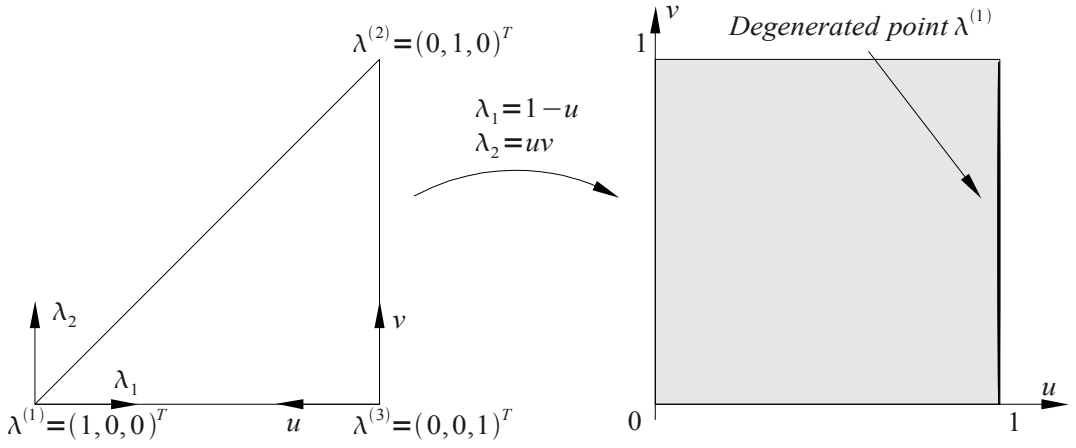


Figure 3.27: Regularization with the Lachat-Watson transformation

where

$$\boldsymbol{\eta} = \left(\frac{1+\lambda_1}{2}, \frac{(1-\lambda_1)(1+\lambda_2)}{4}\right)^{\mathrm{T}} \tag{3.12.15}$$

$$\left|\frac{\partial \boldsymbol{\lambda}}{\partial \boldsymbol{\eta}}\right| = \frac{1-\lambda_1}{8} \tag{3.12.16}$$

As mentioned above this transformation degenerates a singularity at $\boldsymbol{x}^{(1)}$. Figure 3.28 shows the changed Lachat-Watson transformation.

3.13 Collocation method and matrix assembly

Remember equation (3.11.26), the discretized form of the boundary integral equation. The inner integrals relates the node k with the element j. $x^{[k]}$ denotes the nodal position of the globally indexed node k, the source point, and $x^{[l]}$ denotes the nodal position of the globally indexed node l, the load point, respectively. These integrals will now be denoted as

$$\sum_j^E \int_{\Gamma_j} u^*(x^{[k]}, x^{[l]}) \Phi_j^T \mathrm{d}\Gamma_j = G_{kl} \tag{3.13.1}$$

$$\sum_j^E \int_{\Gamma_j} p^*(x^{[k]}, x^{[l]}) \Phi_j^T \mathrm{d}\Gamma_j = \hat{H}_{kl} \tag{3.13.2}$$

and the discretized boundary integral equation can be rewritten as

$$\sum_k^N G_{kl} p^{[k]} = c^{[l]} u^{[l]} + \sum_k^N \hat{H}_{kl} u^{[k]} \tag{3.13.3}$$

and, furthermore, by adding the jump term to $\hat{H}_{kl}$

$$H_{kl} = \begin{cases} \hat{H}_{kl} & k \neq l \\ \hat{H}_{kl} + c^{[l]} & k = l \end{cases} \tag{3.13.4}$$

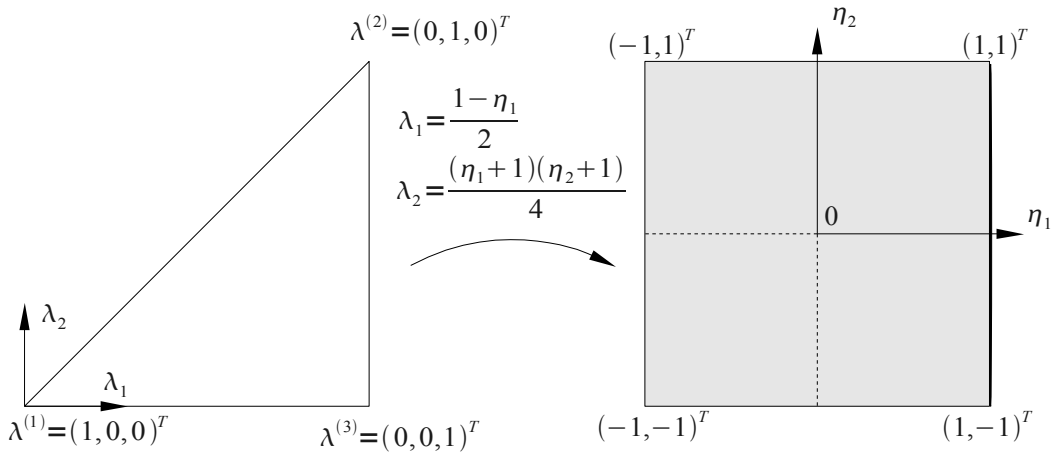


Figure 3.28: Regularization with the Lachat-Watson transformation into the reference square coordinate system

will finally lead to

$$\sum_{i}^{N} G_{kl} p^{[k]} = \sum_{i}^{N} H_{kl} u^{[k]} \tag{3.13.5}$$

A representation in matrix form will be expressed as

$$\boldsymbol{Gp} = \boldsymbol{Hu} \tag{3.13.6}$$

The diagonals of these both matrices contains the self interaction influence values and therefore the singular integral values, which can be calculated for G_{ii} containing the weak singular integrals as shown in section 3.9. The strong singular integral, stored in H_{ii}, can be determined indirectly when a rigid body movement of a bounded body, which is finally a special case of the Neumann boundary value problem, is assumed. To show this, a unit linear displacement in the same arbitrary direction for all nodes is assumed and equation (3.13.6) then becomes

$$\boldsymbol{GI_l} = \boldsymbol{H\emptyset} \tag{3.13.7}$$

where $\boldsymbol{I_l}$ denotes a common displacement vector of all nodes in direction of l. To satisfy this equation it is clear, that for the still unknown diagonal terms, $\hat{H}_{kk} + c^{[k]}$,

$$H_{kk} = -\sum_{k \neq l} H_{kl} \tag{3.13.8}$$

must be valid. Since $H_{kk} = \hat{H}_{kk} + c_k$, the strong singular integral and the jump term have been determined as one value. There is no need to calculate the jump term or the strong singular integral exactly.

Again equation (3.13.6), this matrix equation contains N unknowns and due to the boundary conditions, recall section 3.4.3, $\boldsymbol{u}$ and $\boldsymbol{p}$ consists of N_u and N_p unknowns where $N = N_u + N_p$. This matrix system can be reordered, with the unknowns stored in vector $\boldsymbol{y}$ left hand side, and the knows stored in vector $\boldsymbol{z}$ right hand side, which leads to

$$\boldsymbol{Ay} = \boldsymbol{Bz} = \boldsymbol{F} \tag{3.13.9}$$

This linear equation system, with the fully populated $N \times N$ matrix $\boldsymbol{A}$, can be solved directly or iteratively. Iterative solvers for such linear equation systems are shown in Chapter 7.

Chapter 4

Elastostatics

4.1 Introduction

Elastostatics describes deformations, relations of stress and displacements, on linear elastic bodies under consideration that all forces on the body sum up to zero and the body is at equilibrium. More important compared to the elastodynamics is the time independent view of the situated problem, where displacement and stresses are not a function of time. This chapter will guide through definitions of stresses and strains inside a body and common representations of relations between acting forces on the boundary to resulting deformations and displacements. These terms for the definitions are written in the *Cartesian tensor notation* explained in Appendix A. Most related work of this chapter can be found in the books of Aliabadi [2002], Gaul et al. [2003] and Brebbia [1979].

4.2 Hook's law

Hook's law prescribes relations of an object under deformation. Figure 4.2 shows the deformation as a function of the stress over the strain for a rod under an axial load (Figure 4.1). It can be observed, that in the beginning of deformation, strain and stress have a linear relation. As long as there is a linear relation elastic deformation occurs and the rod will return back into its initial position if it is released again. This is the region to which elastostatics is limited to and repetitions will cause no remaining damage to the rod. Respectively, if the stress becomes to high, the rod's deformation will reach the plastic region and as a result the deformation will not return back to zero if the rod is released. As a measurement to define the ending of elastic and the beginning of plastic deformation the value R_e is used and stands for *Resistance elastic*. For this simple 1-D example the relation of stress and strain,

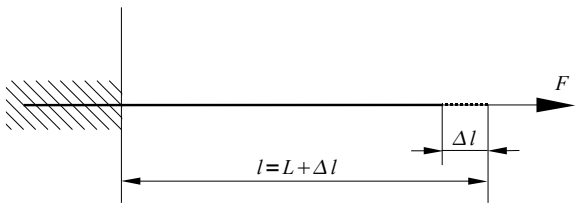


Figure 4.1: Axial loaded rod

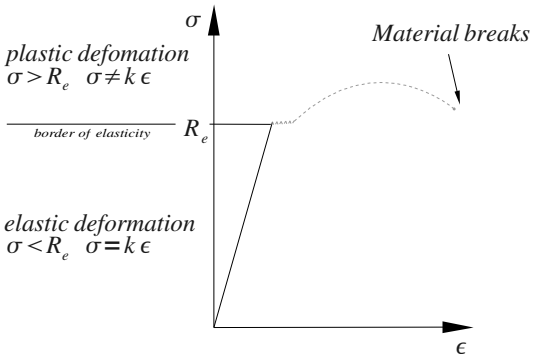


Figure 4.2: Hook's law

assuming elastic deformation, is defined as

$$\sigma = E\varepsilon \tag{4.2.1}$$

where σ is an averaged stress over the cross-section of the rod

$$\sigma = \frac{F}{A} \tag{4.2.2}$$

and ε stands for the strain which is defined as

$$\varepsilon = \frac{l - L}{L} \tag{4.2.3}$$

where l assigns the length of the rod in tension and L its initial length before deformation starts. The scaling factor E is a material constant, the so-called *elasticity factor* or *stiffness factor*. Furthermore, to define Hook's law for higher dimensions the next sections will first define some definitions to finally prescribe Hook's law in those higher dimensions.

4.3 Definition of displacement

The displacement of a particle inside a body is the difference of its initial state and its final state defined as the displacement vector $\boldsymbol{u}$

$$u_i = x_i - X_i \tag{4.3.1}$$

where $\boldsymbol{x}$ is its actual position and $\boldsymbol{X}$ its initial position.

4.4 Definition of strain

As mentioned already above, elastostatics follows *Hook's law* for linear elasticity. For higher ordered dimensions two types of strains must be considered, first the *normal strain* or *direct strain* where the strain and the displacement points towards the same direction and the *shear strain* where the strain is evaluated in an orthogonal direction of the displacement's direction (Figure 4.3). The shear strain can be seen as the difference of the angle between two originally orthogonal vicinal particles of x

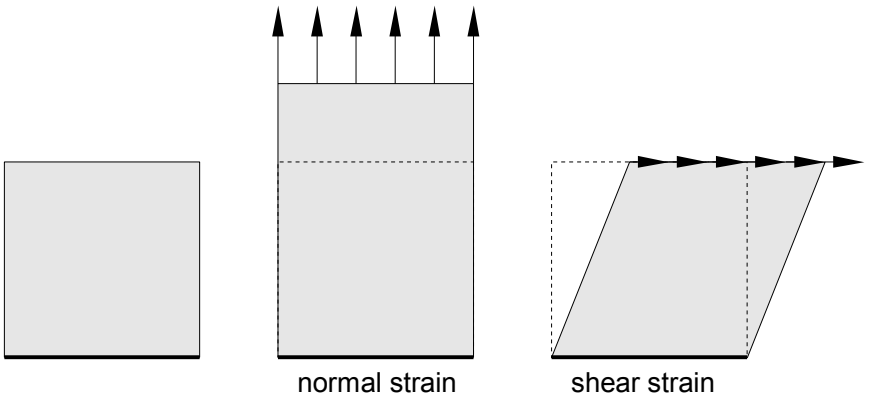


Figure 4.3: Simplified view of normal and shear strain

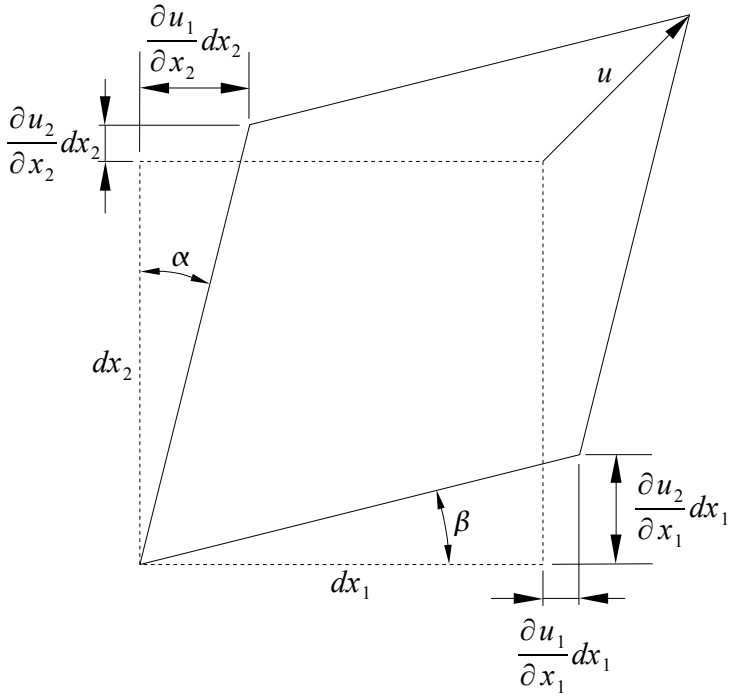


Figure 4.4: Deformation of an infinitesimal particle

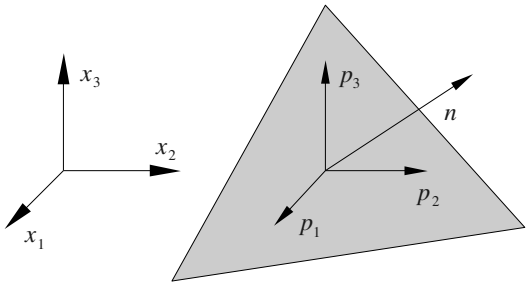


Figure 4.5: Traction p on a surface

(Figure 4.4), namely $\frac{\partial(x+\mathrm{d}x)}{\partial x_i}$ and $\frac{\partial(x+\mathrm{d}x)}{\partial x_j}$. The *Cauchy symmetric linear strain tensor* ε_{ij} is defined, under consideration of small displacements $u \ll 1$ (!), as

$$\varepsilon_{ij} = \alpha + \beta \approx \frac{1}{2}(u_{i,j} + u_{j,i}) \tag{4.4.1}$$

$$\varepsilon_{ij} = \varepsilon_{ji} \tag{4.4.2}$$

$$\boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_{11} & \varepsilon_{12} & \varepsilon_{13} \\ \varepsilon_{12} & \varepsilon_{22} & \varepsilon_{23} \\ \varepsilon_{13} & \varepsilon_{23} & \varepsilon_{33} \end{pmatrix} \approx \frac{1}{2} \begin{pmatrix} 2\frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1} & \frac{\partial u_1}{\partial x_3} + \frac{\partial u_3}{\partial x_1} \\ \frac{\partial u_2}{\partial x_1} + \frac{\partial u_1}{\partial x_2} & 2\frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} + \frac{\partial u_3}{\partial x_2} \\ \frac{\partial u_3}{\partial x_1} + \frac{\partial u_1}{\partial x_3} & \frac{\partial u_3}{\partial x_2} + \frac{\partial u_2}{\partial x_3} & 2\frac{\partial u_3}{\partial x_3} \end{pmatrix} \tag{4.4.3}$$

Since this is a definition in higher dimensions the strain is no longer a single value, more than a second-ordered tensor.
Primely the strain is defined as above in equation (4.2.3) which becomes for the Cauchy strain tensor to

$$\varepsilon = \lim_{\boldsymbol{L} \to \emptyset} \frac{\Delta \boldsymbol{l}}{\boldsymbol{L}} \tag{4.4.4}$$

For the shear strain it is clear, that it vanishes in an 1-D case, but for higher dimensions it might be unequal zero. Additionally, it can be seen, that in the case of a rigid body motion this strain tensor yields zero, which is desired, since no deformation should occur as long the whole body is moved.

4.5 Definition of stress

Stress according to the mechanics of materials is defined as the force F acting on an area A, the stress vector

$$\boldsymbol{p} = \frac{\boldsymbol{F} \cdot \boldsymbol{n}_A}{||\boldsymbol{A}||} \tag{4.5.1}$$

with $\boldsymbol{n}_A$ as the outwards pointing normalized normal vector of $\boldsymbol{A}$ (Figure 4.5). Since the vector depends on the orientation of the surface, this representation of the stress vector for a material point, which is given by the pair $\boldsymbol{F}$ and $\boldsymbol{A}$ is not suited. By introducing the second-ordered *Cauchy stress tensor* σ_{ij} the state of stress for a material point is represented by the three orthogonal planes e_i, e_j, e_k (Figure 4.6). The Cauchy stress principle states, that the stress due to the force $\boldsymbol{F}$ acting on an area $\boldsymbol{A}$ becomes

$$\boldsymbol{p} = \lim_{\Delta \boldsymbol{A} \to \emptyset} \frac{\Delta \boldsymbol{F}}{||\Delta \boldsymbol{A}||} \boldsymbol{n}_A = \frac{\mathrm{d}\boldsymbol{F}}{\mathrm{d}\boldsymbol{A}} \boldsymbol{n}_A \tag{4.5.2}$$

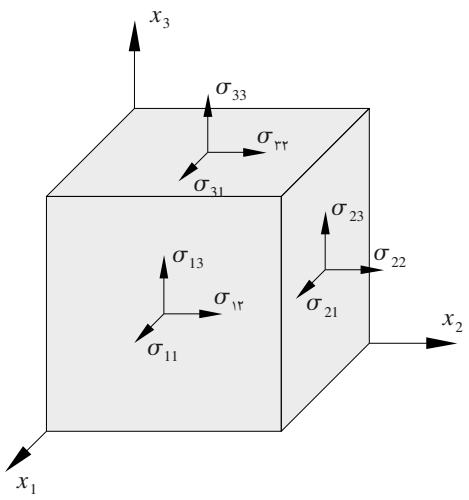


Figure 4.6: Internal stress representation on an infinitesimal particle

while $||A||$ tends to zero, and it can be written instead

$$p_i = \sigma_{ij} n_j \tag{4.5.3}$$

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{pmatrix} \tag{4.5.4}$$

where $\boldsymbol{n}$ represents the surface's normal. Similar to the strain stress can be classified into to different types, the *normal stress*, where the force points in the same direction as the surface normal and *shear stress*, where the force points in an orthogonal direction of the surface normal. Of course is normal strain caused by normal stress and shear strain caused by shear stress.

4.6 Constitutive equation

For elastic materials the state variables, the stress tensor σ_{ij} and the strain tensor ε_{kl}, depend both on material properties and therefore depends the stress on the strain and vice verse. Now under assumption, that the strain tensor is the state independent one, and the stress tensor depends on the strain, the *generalized Hooke's law*, which represents the material behavior and describes the stress as a function of the strain, can be formulated as

$$\sigma_{ij} = C_{ijkl} \varepsilon_{kl} \tag{4.6.1}$$

The so-called *elasticity tensor* or *stiffness tensor* C_{ijkl} is a fourth-order tensor, a $(3 \times 3 \times 3 \times 3)$ hypercube, containing $4^3 = 81$ material constants and prescribes the material behavior for an anisotropic, linear-elastic material. Fortunately, due to symmetries, assumption of isotropy, and independence of direction [Gaul et al., 2003], the elasticity tensor can be reduced to its simplest form for a homogeneous and isotropic material depending further more only on two independent constants, the *Lamé's moduli* λ and μ

$$C_{ijkl} = \lambda \delta_{ij} \delta_{kl} + \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) \tag{4.6.2}$$

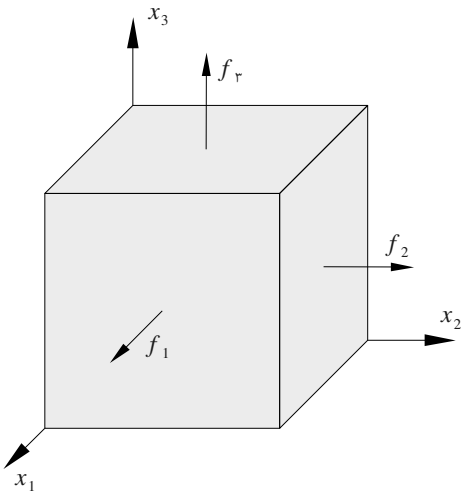


Figure 4.7: Internal body forces on an infinitesimal particle

The generalized Hook's law yields for this elasticity tensor

$$\sigma_{ij} = \lambda \delta_{ij} \varepsilon_{kk} + 2\mu \varepsilon_{ij}. \tag{4.6.3}$$

Other representation formulas using the the more common material constants E (Young's modulus), G (Shear modulus) and ν (Poisson's ratio), which can be calculated out of λ and μ

$$\begin{aligned} E &= \frac{\mu(3\lambda + 2\mu)}{\lambda + \mu} \\ G &= \mu \\ \nu &= \frac{\lambda}{2(\lambda + \mu)} \end{aligned}$$

4.7 Equilibrium equations

If the body is in equilibrium it follows

$$\sigma_{ij,j} + f_i = 0 \tag{4.7.1}$$

where f_i denotes internal body forces (Figure 4.7). This are two (three) equations with four (six) unknowns for two (three) dimensional plane strain problems. Furthermore, if no body moments are applied this equation leads also to

$$\sigma_{ij} = \sigma_{ji} \tag{4.7.2}$$

4.8 Boundary integral equation

Recall, the boundary conditions for mixed boundary value problems, introduced in Section 3.4.3, are $u = \bar{u}$ on Γ_D and $p = \bar{p}$ on Γ_N. Applying the steps explained in the previous chapter in Section 3.5

and 3.7 the weighted residual integral can be rewritten, by choosing a weighting function that follows

$$\sigma^*_{ij,j} + e_i\delta(x-\xi) = 0, \tag{4.8.1}$$

as

$$\int_\Omega u^*_{ij}(\sigma^*_{ij,j} + f_i)\mathrm{d}\Omega = \int_\Gamma p^*_{ij}u_i\mathrm{d}\Gamma + \int_\Gamma u^*_{ij}p_i\mathrm{d}\Gamma \tag{4.8.2}$$

where $u^*_{ij}(\xi,x)$ and $p^*_{ij}(\xi,x)$ are the displacements and tractions to the weighting field with $p^*_{ij} = \sigma^*_{ij}n_j$ according to equation (4.5.3).

4.9 Fundamental solutions

The fundamental solutions, u^*_{ij} and p^*_{ij}, of equation (4.8.2) for linear elasticity are also known as *Kelvin's fundamental solutions* and are given by [Brebbia, 1979; Hunter and Pullan, 2001]

$$u^*_{ij}(\xi,x) = \frac{-1}{8\pi(1-\nu)G}\left[(3-4\nu)\ln(||r||)\delta_{ij} + r_{,i}r_{,j}\right] \tag{4.9.1}$$

$$p^*_{ij}(\xi,x) = \frac{-1}{4\pi(1-\nu)||r||}\left\{\left[(1-2\nu)\delta_{ij} + 2r_{,i}r_{,j}\right]\frac{\partial r}{\partial n} - (1-2\nu)(r_{,i}n_j - r_{,j}n_i)\right\} \tag{4.9.2}$$

for two dimensional and

$$u^*_{ij}(\xi,x) = \frac{1}{16\pi(1-\nu)G||r||}\left[(3-4\nu)\delta_{ij} + r_{,i}r_{,j}\right] \tag{4.9.3}$$

$$p^*_{ij}(\xi,x) = \frac{-1}{8\pi(1-\nu)||r||^2}\left\{\left[(1-2\nu)\delta_{ij} + 3r_{,i}r_{,j}\right]\frac{\partial||r||}{\partial n} - (1-2\nu)(r_{,i}n_j - r_{,j}n_i)\right\} \tag{4.9.4}$$

for three dimensional plane strain problems. $r = r(\xi,x) = x - \xi$ represents the distance between the field point x and the source point ξ as a vector. n is the outward pointing normal vector of Γ at the specified boundary position x and yields for [James and Pai, 1999]

$$\frac{\partial||r||}{\partial n} = \frac{r\cdot n}{||r||} \tag{4.9.5}$$

Chapter 5

3-D multiresolution surface

5.1 Introduction

This chapter will give a short preview of subdivision methods for 3D meshes and multiresolution surface representation. Subdivision of a surface means splitting the surface elements into smaller parts, called child elements or child faces, to approximate a smoother surface in a recursive way [Peters and Reif, 2008; Akenine-Moller and Haines, 2002]. Based on the positions of nearby known vertices a refinement scheme computes new vertices for new child faces which finally gives rise to a denser mesh. This process can be applied again on the subdivided surface elements to recursively improve the final result which will lead to a hierarchical representation of the mesh. In the case of the boundary element method this subdivision algorithms are used to gain more nodal points for a more accurate solution. However the computing time increases dramatically with the number of nodes. Therefore, an intelligent representation for the surface is required. Commonly a surface consists of triangular elements, quadrilateral elements or both.

5.2 Recursive refinement schemes

Refinement schemes can be differentiated into interpolating or approximating techniques. Interpolating schemes require the original vertices positions being part of the new subdivided mesh, which means that they must not be displaced. They commonly converge to a similar size compared with the original mesh. These schemes are implemented in the Butterfly or Kobbelt subdivision algorithm. In contrast, approximating schemes allow to adjust the original mesh vertices as needed to approximate a smooth surface. Representatives for interpolating schemes are the Loop, Catmull-Clark or Doo-Sabin subdivision algorithm. The subdivided mesh is then often smaller than before. Some important terms in the context of subdivision algorithms are [Pharr and Fernando, 2005; Akenine-Moller and Haines, 2002]:

- Limit surface: the theoretical surface computed after an infinite number of subdivision steps.
- Valence (of a vertex): the number of edges connected to a vertex.
- Extraordinary point: a vertex with a valence differing from the algorithm's predicted valence (Triangular six, Quadrilateral four)
- Corner: the number of faces connected to a vertex with different surface normals at this vertex.

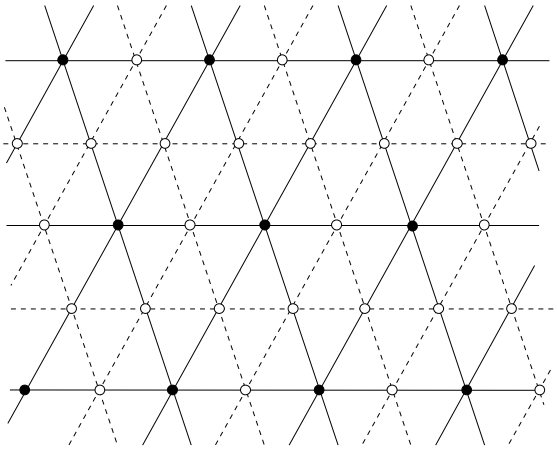


Figure 5.1: Triangular subdivision

5.3 3D surface subdivision methods

This section gives a short overview of common subdivision methods for 3-D meshes. It focuses primarily on how this algorithm refines a mesh, rather than on an analysis of the convergence rate to the limit surface, on convex hulls or on other topics in this context.

5.3.1 Triangular elements

Triangles are the simplest and most frequently used face type for surface elements in the context of 3D meshes. Triangles are the sole polygons which are always planar.

1-to-4 subdivision

This algorithm subdivides one triangular surface patch into four parts by creating new vertices in the middle of each edge and connect them to a new triangle as shown in Figure 5.1 where black dots represent still existing vertices from the original mesh before subdivision and white dots stand for newly added vertices. The dashed lines represent newly added edges. This subdivision does not smooth the surface, which means each edge of the original mesh is viewed as a crease and is normally used if a higher mesh density is needed.

Loop subdivision

Loop [1987] first prescribed this subdivision algorithm in 1987. It is a very common algorithm for triangular surface subdivision. In a first step the triangle is subdivided by the 1-to-4 subdivision algorithm explained in the previous subsection. In a second step the Loop subdivision applies weight masks as depicted in Figure 5.2 on the vertices resulting from bi-cubic uniformed B-splines, to guarantee that new vertices have C^2 continuity, and recalculates positions of new and old vertices. For

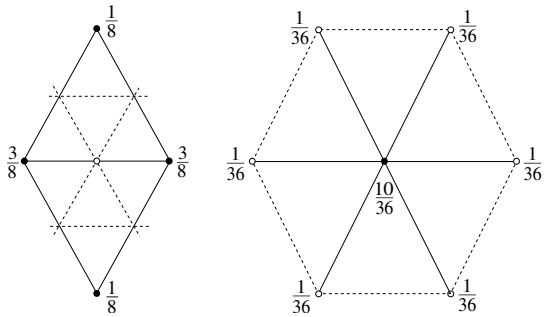


Figure 5.2: Weight masks for Loop subdivision (Valence 6)

extraordinary vertices this weight masks must be adapted, as shown in Figure 5.3, where

$$\beta = \frac{1}{n}(\frac{5}{8} - \frac{3}{8}(\frac{1}{4} + \cos\frac{2\pi}{n})^2) \tag{5.3.1}$$

if the *Original loop* is used while

$$\beta = \begin{cases} \frac{3}{8n} & n > 3 \\ \frac{3}{16} & n = 3 \end{cases} \tag{5.3.2}$$

if the *Warren* refinement method is used.

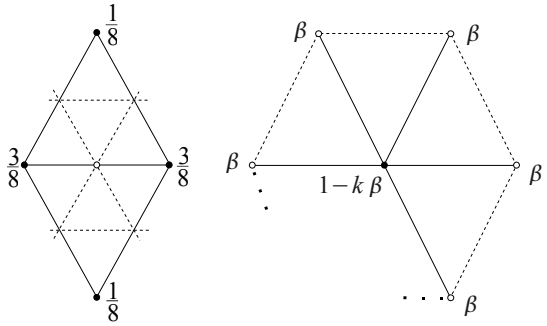


Figure 5.3: Weight masks for Loop subdivision (Valence k)

$\sqrt{3}$ subdivision

This subdivision method for triangular surface patches introduced by Kobbelt [2000] generates subdivided meshes, where the number of triangles increases by a factor of 3 instead of 4 for each subdivision step. First, this method adds a new vertex in the center of each triangle and connects this new vertex to the corners of the triangle. Afterwards the old edges are flipped to avoid degenerated triangles. This is explained in Figure 5.4.

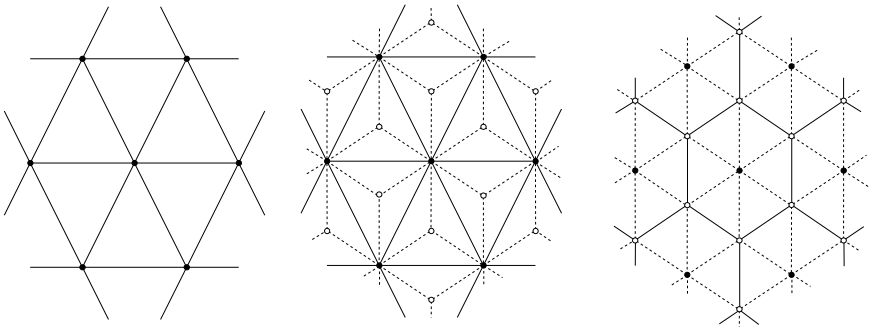


Figure 5.4: $\sqrt{3}$ subdivision method

Butterfly subdivision

This method is an interpolating subdivision scheme for triangles and was introduced by Dyn et al. [1990]. The algorithm inserts vertices for each edge following the rule

$$P^{k+1} = \frac{1}{16}[8(P_1^k + P_2^k) + 2(P_3^k + P_4^k) - (P_5^k + P_6^k + P_7^k + P_8^k)] \tag{5.3.3}$$

as shown in Figure 5.5. As required for interpolating schemes, this subdivision method keeps the original vertices in place.

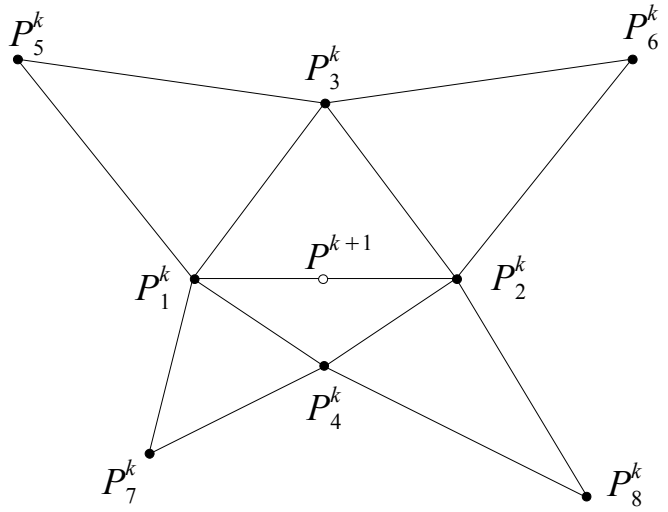


Figure 5.5: Butterfly subdivision scheme

5.3.2 Quadrilateral elements

Quadrilaterals, a second often used surface element, have also an important relevance in the context of subdivision schemes and figure commonly as the opposites to the triangles while quadrilateral elements are not always planar.

1-to-4 subdivision

This algorithm splits each face into four parts by adding new vertices in the center of each face and edge. This algorithm does not smooth the surface anyway, but it can be used if a higher resolution is needed (Figure 5.6).

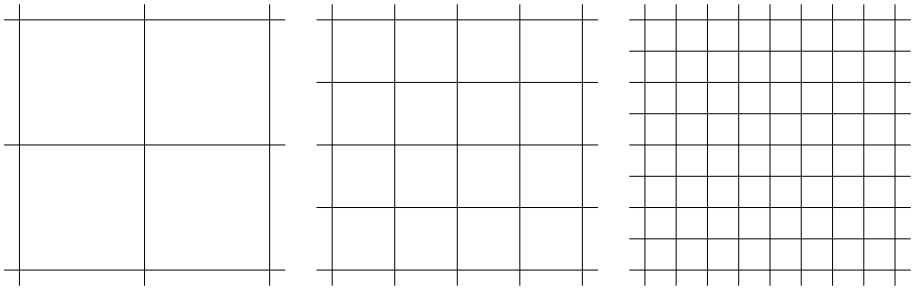


Figure 5.6: Simple 1-to-4 subdivision

Midpoint subdivision

The midpoint algorithm represents the simplest subdivision method to create smoother surfaces. It creates new vertices in the middle of each edge and connects tbe new middle points of the edges enclosing one patch. This algorithm smooths a surface in a straightforward way (Figure 5.7).

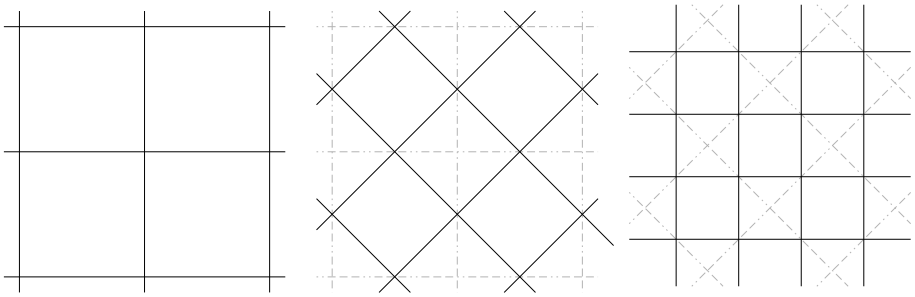


Figure 5.7: Midpoint subdivision algorithm

Catmull-Clark subdivision

This subdivision algorithm was introduced by Edwin Catmull and Jim Clark in 1978 [Catmull and Clark, 1978]. The Catmull-Clark technique uses bi-cubic uniformed B-splines to guarantee a smooth surface after the refinement step with C^1 continuity at the original mesh vertices and C^2 continuity elsewhere. This refinement works in a recursive manner and for different types of meshes but yields best results for quad meshes. The use of the Catmull-Clark algorithm to subdivide a control mesh consisting only of quad meshes is straightforward. First, each face has to be split into four faces by adding a new vertex in the center of the face and in the middle of each edge. So-called weight masks

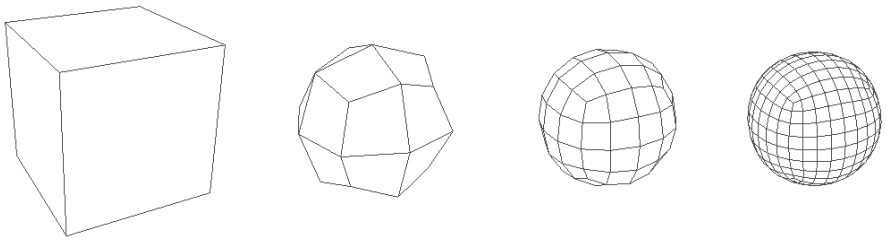


Figure 5.8: Catmull Clark subdivision of a cube

are used to recompute the positions of these new vertices and the new positions of the original vertices from the control mesh. These weight masks for a valence of 3, 4 and 5 are shown in Figure 5.9 and for a valence of k in Figure 5.10.

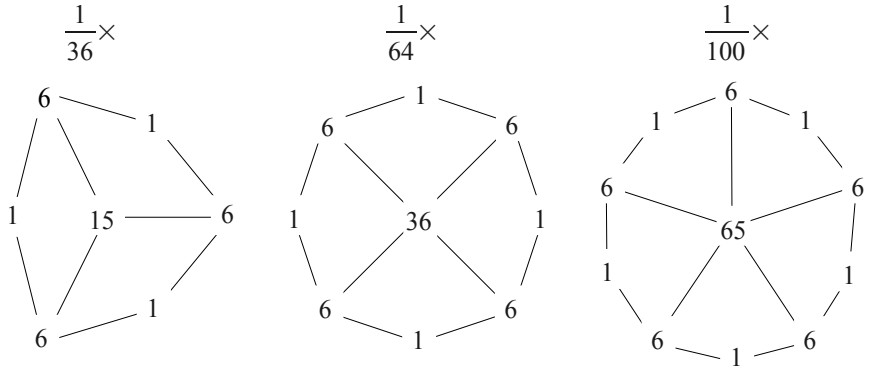


Figure 5.9: Catmull-Clark weight masks for a valence of 3, 4 and 5 [Pharr and Fernando, 2005]

Doo-Sabin subdivision

Similarly to the idea of Catmull-Clark, this subdivision method devised in 1978 by Doo [1978] and improved by Doo and Sabin [1978] uses bi-quadratic uniformed B-splines to guarantee a smooth surface. It shrinks each face by a percentage value and connects the corners of the shrunken faces to create new faces (Figure 5.11).

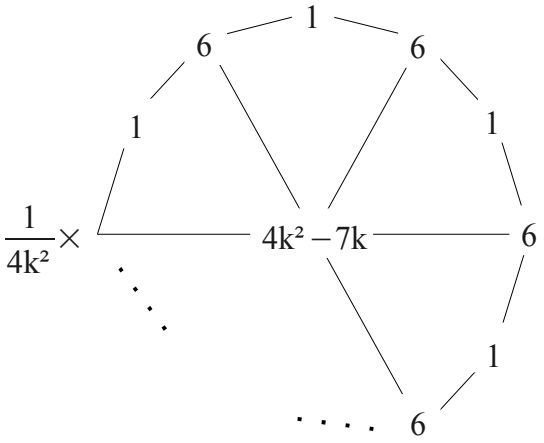


Figure 5.10: Catmull-Clark weight masks for a valence of k [Pharr and Fernando, 2005]

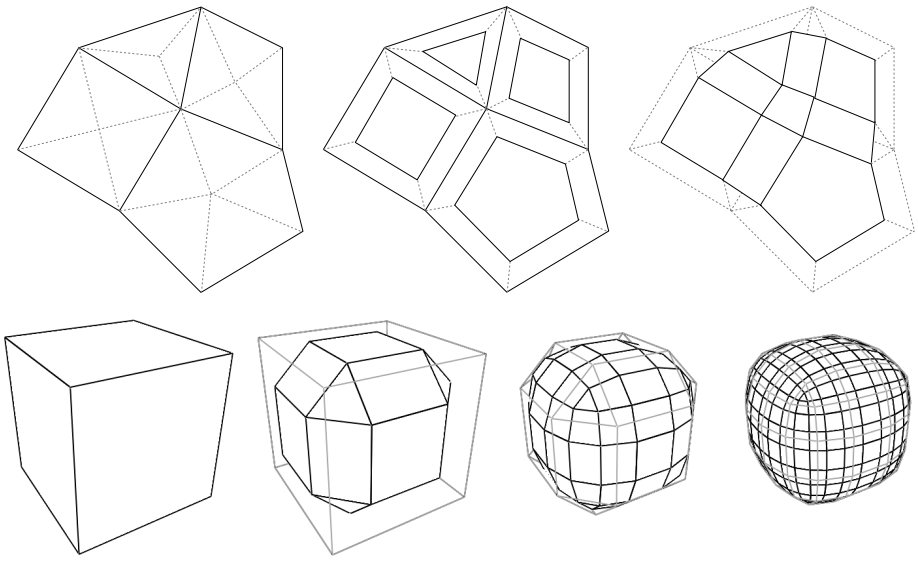


Figure 5.11: Doo-Sabin subdivision on a cube

5.4 Wavelet representation for multiresolution surfaces

This section shows the representation for 3-D multiresolution surfaces using the technique of wavelets. Similar to the Fourier transformation, where an arbitrary signal is reconstructed by the sum of different sinus and cosines functions transforming the signal into the time domain, the signal is reconstructed by a sum of scaled and translated copies of one so-called *mother wavelets* transforming the signal into the wavelet domain. Mallat [1989], Daubechies [1992] and Chui [1992] introduced wavelets theoretically and used them from a signal processing point of view. The great benefit of wavelets lies in the fact that only a small number of coefficients are required to represent general functions or large datasets accurately. It results in a compression of the used data and in more efficient computations. As a simple example of wavelets construction the Haar wavelet will be introduced next.

5.4.1 Haar wavelet

The Haar wavelet, as show in Figure 5.12, is the simplest mother wavelet for wavelet transformations and was proposed in 1909 by Alfréd Haar [Daubechies, 1992]. For an arbitrary discretized signal

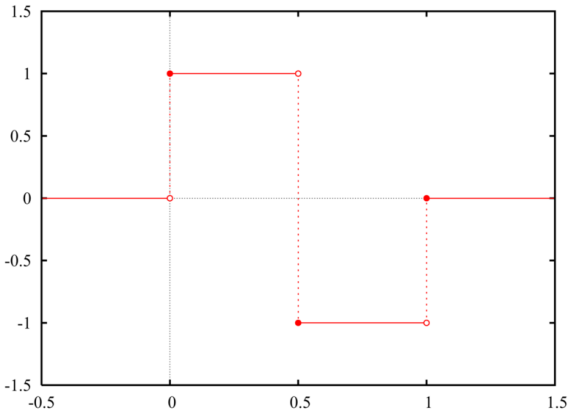


Figure 5.12: Haar wavelet

x, one can assume that two neighboring samples of the signal have a high correlation. The Haar wavelet tries to take a benefit from this and transforms any two neighboring samples into the average and the difference of both, thus generating two new sequences. A given sequence of data x can be transformed to v and w, namely

$$v_i = \frac{x_{2i} + x_{2i+1}}{2} \tag{5.4.4}$$

$$w_i = x_{2i+1} - v_i = \frac{x_{2i+1} - x_{2i}}{2} \tag{5.4.5}$$

where v denotes the average values and w represents the differences. If the neighboring samples are highly correlated the difference value will yield to zero and this will save storage memory. For this case the function prescribed by the average values v_i is very close to the original signal. The average values can be viewed as a coarser representation of the original signal containing the low frequencies. Likewise, the difference values represent the details and are the so-called *wavelet coefficients*

containing the high frequencies. The procedure can be reversed to restore the original signal

$$x_{2i} = x_i - z_i \tag{5.4.6}$$

$$x_{2i+1} = x_i + z_i \tag{5.4.7}$$

After a second wavelet transformation, applying equations (5.4.4) and (5.4.5) once again, and using the coarse signal v as the new input signal the wavelet coefficients for the next level can be computed. This can be repeated until only one value remains, which is the average of a continuous signals.

5.4.2 Wavelets for multiresolution surfaces

The idea of wavelets, like the Haar wavelet, processed on 1-D signals can also be applied to subdivided surfaces to generate multiresolution surfaces using wavelets in 3 dimensions as first introduced by Lounsbery et al. [1997] and Stollnitz et al. [1996]. After subdividing the surface with a subdivision algorithm, the surface can be transformed into wavelets using proper mother wavelets. For each subdivision transformation level the wavelet coefficients are computed (Figure 5.13). Bertram et al. [2004] present in their paper general B-spline subdivision-surface wavelets for geometry compression. For an arbitrary model the surface mesh points must be re-sampled to gain sub-connectivity as shown in the article of Eck et al. [1995].

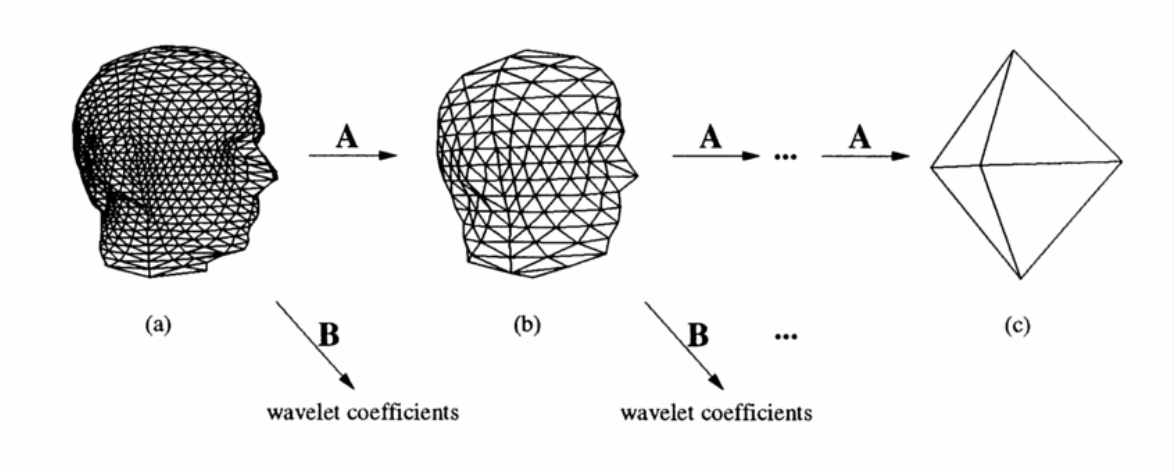


Figure 5.13: Multiresolution surface representation [Stollnitz et al., 1996]

Chapter 6

Matrix representations and storage

6.1 Introduction

Matrices are used to store the nodes-to-nodes interaction values. Their size increases normally by four for each level of depth added and in addition also by the number of nodes needed according to the interpolation function. The size of the matrix and due to this the computing time may rise dramatically. Wavelets technique helps to reduce this expense.

6.2 Dense matrix representation

Dense matrices store all values in the matrix - the non-zero values and the zero entries as well. The data are stored in a two dimensional array, for example as vectors of vectors and requires for a $n \times m$ matrix a memory allocation of

$$B = n \cdot m \cdot \alpha; \tag{6.2.1}$$

bytes, where α denotes the number of bytes needed to represent one matrix entry, for instance, four bytes for a float value. This will explode for large systems since the memory requirements increasing linear to$n \times m$. However, if the matrix is sparse, other representation types will save memory, since zero entries will not be saved.

6.3 Sparse matrix representation

More interesting are matrix representations which take into account the large number of zero entries in a sparse matrix. A matrix is known as sparse, if just a fraction are non-zero values compared to the whole number of entries. There are several techniques and formats to store sparse matrices for iterative methods. The memory requirements for these matrix representations depend mainly on the number of non-zero entries. The following representation types are mainly taken from the book of [Saad, 2003, Chapter 3.4].

6.3.1 Coordinate format (COO)

[Saad, 2003, ch. 15] This is the simplest format, which stores three values for each matrix entry, the *row-index*, the *column-index* and the *matrix entry value* itself in an arbitrary order.

$$\boldsymbol{A} = \begin{pmatrix} 11 & 12 & 0 & 14 & 0 & 0 \\ 0 & 22 & 23 & 0 & 0 & 0 \\ 31 & 0 & 33 & 34 & 0 & 0 \\ 0 & 42 & 0 & 44 & 45 & 46 \\ 0 & 0 & 0 & 0 & 55 & 56 \\ 0 & 0 & 0 & 0 & 65 & 66 \end{pmatrix} \tag{6.3.2}$$

will be stored as

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
value	65	66	55	56	42	44	45	46	11	12	14	22	23	31	33	34
row-index	5	5	4	4	3	3	3	3	0	0	0	1	1	2	2	2
column-index	4	5	4	5	1	3	4	5	0	1	3	1	2	0	2	3

A benefit of this storage type is that additional matrix entries can be easily added at the end of the arrays. However, this matrix storage format requires quite a large amount of memory, since two additional arrays must be stored, and therefore it is seldom used.

6.3.2 Modified COO format (MCOO)

Modified COO uses instead of the two index arrays, row-index and column-index, only one index array to store the values $i \cdot n + j$, where i represents the row-index and j the column-index. n denotes the number of columns. This index representation is unique. The matrix (6.3.2) will be stored with the modified COO format as

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
value	65	66	55	56	42	44	45	46	11	12	14	22	23	31	33	34
index	34	35	28	29	19	21	22	23	0	1	3	7	8	12	14	15

This modification needs only one index vector, but is only usable for smaller matrices, since the bits needed to store an index value increases with the matrix size by the power of two. For matrices larger than 256×256 more than 16 bits are needed to store a index value and the algorithm becomes inefficient. Another disadvantage are the additional calculation steps which are needed to express the index value.

6.3.3 Compressed row storage format (CRS)

In this storage format the values are ordered in an ascending direction and the row-index vector is used to store the start index in the column-index vector where the row starts, as shown below. The row-index of a value itself is stored by the index of the row-index vector.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
value	11	12	14	22	23	31	33	34	42	44	45	46	55	56	65	66
column-index	0	1	3	1	2	0	2	3	1	3	4	5	4	5	4	5

	0	1	2	3	4	5
row-index	0	3	5	8	12	14

As an example an access at index $(\text{row} = 3, \text{column} = 4)$ is considered. In this case, the row-index vector at index 3 tells that the row starts at index 8 in the column-index vector, and will end at 11, since the next row starts at 12. The next step is to search for the column-index 4 between indexes 8 and 11. Once the corresponding value of the value vector at this column-index is found, it will be returned otherwise it will be 0.
This format needs less memory than the COO-format, since the row-indexes are collected now. However, to access an arbitrary index it might be needed to check the whole row to figure out that the index even does not exist and therefore the return value yields zero. For normal matrix-vector multiplications (MVs) this is not a drawback, since each element inside a row must be accessed anyway. Note, however, that pure column accesses necessary to access the transposed matrix are not possible anymore. To do so, another storage format is needed or the matrix has to be stored twice in the memory, which is only useful as long the matrix is not too large. Nevertheless the algorithm can be reordered to collect the columns and to gain pure column access at the expense of the row accesses.

6.3.4 Modified CRS Format (MRS)

This format is similar to the CRS format and assumes that the main diagonal of the matrix includes values unequal zero, which becomes true for most matrices. Instead of using a separate row-index vector, this algorithm stores at first the main diagonal entries into the value vector and then uses the column-index vector as the row-index vector. For the first diagonal entries its not needed to store a column-index separately, since it is clear that the first entry in the diagonal vector is in the first row and the first column of the matrix, the second in the second row and the second column and so on. The values of matrix (6.3.2) will be stored as

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
value	11	22	33	44	55	66	12	14	23	31	34	42	45	46	56	65
row/column	6	8	9	11	14	15	1	3	2	0	3	1	4	5	5	4

The advantages and disadvantages are similar to the ones in the CSR format, while less memory is needed. This format is very common.

Chapter 7

Iterative numerical solver for linear equation systems

7.1 Introduction

A linear equation system is commonly expressed by a matrix $\boldsymbol{A}$, a solution vector $\boldsymbol{x}$ and a non-zero right hand vector $\boldsymbol{b}$. $\boldsymbol{x}_*$ expresses the exact solution.

$$\boldsymbol{A}\boldsymbol{x}_* = \boldsymbol{b} \tag{7.1.1}$$

$$\boldsymbol{x}_* = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{7.1.2}$$

Solving a huge linear system by computing the inversion of $\boldsymbol{A}$ takes a long time. Iterative solvers estimate a solution and try to improve this solution with each step by evaluating a cost-function to find a minimum for the residual of the current solution which is defined as

$$\boldsymbol{r}_n = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_n \tag{7.1.3}$$

For the exact solution $\boldsymbol{x}_*$ the residual is $\emptyset$. If the algorithm converges, an acceptable solution is calculated after a finite number of steps. An iterative algorithm starts with an initial solution $\boldsymbol{x}_0$, typically choosing $\boldsymbol{x}_0 = \emptyset$, and calculates new solution vectors $\boldsymbol{x}_n | n > 0$ iteratively. The next sections introduce iterative algorithms to calculate such solutions.

7.2 Preconditioners

Before beginning with iterative solver, preconditioners and their usage will be explained. First of all, preconditioners may help to find an acceptable solution faster. A matrix with a high condition number needs more steps to find an appropriate solution, since the incremental steps to the direction of the exact solution are small. The condition number of a matrix can be measured by its minimum and maximum singular value

$$\kappa(\boldsymbol{A}) = \frac{\sigma_{\max}}{\sigma_{\min}} \tag{7.2.4}$$

and a preconditioner will change these singular values. Applying preconditioners will change a linear equation system to

$$\begin{array}{rcll} (\boldsymbol{C}_L\boldsymbol{A})\boldsymbol{x} & = & \boldsymbol{C}_L\boldsymbol{b} & \text{left preconditioning} \\ (\boldsymbol{A}\boldsymbol{C}_R)(\boldsymbol{C}_R^{-1}\boldsymbol{x}) & = & \boldsymbol{b} & \text{right preconditioning} \\ (\boldsymbol{C}_L\boldsymbol{A}\boldsymbol{C}_R)(\boldsymbol{C}_R^{-1}\boldsymbol{x}) & = & \boldsymbol{C}_L\boldsymbol{b} & \text{left and right preconditioning} \end{array} \tag{7.2.5}$$

This has no influence on the final solution of $\boldsymbol{x}$. Generally, good preconditioners are good estimators of the inverse matrix of $\boldsymbol{A}$, where

$$\boldsymbol{C} \approx \boldsymbol{A}^{-1} \tag{7.2.6}$$

7.3 Error vectors and residuals

The error vectors and residuals are essential values in the estimating process for accurate solutions of a linear equation system in an iterative way. They are the measurement for an evaluation of computed solutions. In a first approach an error vector, representing the exactness of the actual solution, is defined as

$$\boldsymbol{d}_n = \boldsymbol{x}_n - \boldsymbol{A}^{-1}\boldsymbol{b} = \boldsymbol{x}_n - \boldsymbol{x}_* \tag{7.3.7}$$

but not determinable because the actual solution $\boldsymbol{x}_*$ is normally unknown while solving. However, a convergence can be estimated from the residual vector or from the residual of an actual solution which is defined as

$$\begin{aligned} \boldsymbol{r}_n &= \boldsymbol{b} - \boldsymbol{A}x_n & (7.3.8)\\ \|\boldsymbol{r}\| &= \langle \boldsymbol{r}, \boldsymbol{r} \rangle = \sqrt{\boldsymbol{r}^T \boldsymbol{r}} & (7.3.9) \end{aligned}$$

and converges to $\emptyset$, if $\boldsymbol{x}_n$ converges to $\boldsymbol{x}_*$.

7.4 Inner products and vector norms

Inner products of vector pairs and vector norms of vectors can be computed in several ways and find a large usage in the algorithms of iterative solvers. First of all, an inner product of a vector pair is defined as

$$\langle \boldsymbol{v}, \boldsymbol{w} \rangle \equiv \boldsymbol{w}^{\mathrm{H}}\boldsymbol{v} = \sum_i v_i \overline{w_i} \quad \{v_i, w_i | \forall i \in \mathbb{C}\} \langle \boldsymbol{v}, \boldsymbol{w} \rangle \equiv \boldsymbol{w}^{\mathrm{T}}\boldsymbol{v} = \sum_i v_i w_i \quad \{v_i, w_i | \forall i \in \mathbb{R}\} \tag{7.4.10}$$

$\overline{w_i}$ denotes the conjugate complex of w_i and the superscript $\boldsymbol{w}^{\mathrm{H}}$ denotes the Hermitian of $\boldsymbol{w}$, the conjugate complex transposed. From here on, only real matrix and vector entries are assumed. Vector norms are functions that assign strictly positive values to a vector except to the zero vector. There are several types of vector norms, for example [Saad, 2003]:

$$\begin{aligned} ||\boldsymbol{v}||_1 &= \sum_i |v_i| & (7.4.11)\\ ||\boldsymbol{v}||_2 &= \sqrt{\sum_i |v_i|^2} & (7.4.12)\\ ||\boldsymbol{v}||_p &= \sqrt[p]{\sum_i |v_i|^p} & (7.4.13)\\ ||\boldsymbol{v}||_\infty &= \max_i |v_i| & (7.4.14) \end{aligned}$$

The most common norm is the 2-norm, $||\cdot||_2$, also known as the *Euclidean norm*, representing the length of the vector. From here on, if not denoted separately, the norm $||\cdot||$ without any subscript is always meant as the 2-norm $||\cdot||_2$. The euclidean norm can be written as an inner product

$$||\boldsymbol{v}|| = \sqrt{\langle \boldsymbol{v}, \boldsymbol{v} \rangle} \tag{7.4.15}$$

instead. Furthermore, the norm over the matrix-vector product $\boldsymbol{Gv}$ is also defined as a vector norm, the so called $\boldsymbol{G}^{\mathrm{T}}\boldsymbol{G}$-norm, and one can say

$$||\boldsymbol{v}||_{\boldsymbol{G}^{\mathrm{T}}\boldsymbol{G}} = ||\boldsymbol{Gv}|| = \sqrt{\langle \boldsymbol{Gv}, \boldsymbol{Gv} \rangle} = \sqrt{\langle \boldsymbol{v}, \boldsymbol{G}^{\mathrm{T}}\boldsymbol{Gv} \rangle} \tag{7.4.16}$$

The last type of vector norm which is to be mentioned here is the so-called *energy norm* or $\boldsymbol{A}$-norm, where compared to the $\boldsymbol{G}^{\mathrm{T}}\boldsymbol{G}$-norm, $\boldsymbol{G}^{\mathrm{T}}\boldsymbol{G}$ can be thought as $\boldsymbol{A}$, finally written as

$$||\boldsymbol{v}||_{\boldsymbol{A}} = \sqrt{\langle \boldsymbol{v}, \boldsymbol{Av} \rangle} \tag{7.4.17}$$

7.5 Orthogonality

The next term is the so-called *orthogonality* of two vectors and is an important term in the sense of iterative solver since most algorithms use projections onto the previously computed solution space. Two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ are defined to be *orthogonal* if their inner product yields zero

$$\boldsymbol{v} \perp \boldsymbol{w} \iff \langle \boldsymbol{v}, \boldsymbol{w} \rangle = 0 \tag{7.5.18}$$

Orthogonal vectors are normal to each other and additionally, if the 2-norm of both vectors satisfies $||\boldsymbol{v}|| = ||\boldsymbol{w}|| = 1$ they are also defined as *orthonormal*. The inner product of two vectors can also be viewed as a projection onto each other. Actually an inner product can also be computed by

$$\langle \boldsymbol{v}, \boldsymbol{w} \rangle = ||\boldsymbol{v}||\, ||\boldsymbol{w}|| \cos\alpha \tag{7.5.19}$$

where α represents the angle included between the vectors $\boldsymbol{v}$ and $\boldsymbol{w}$, with $\cos\alpha = 0$ if $\boldsymbol{v}$ is normal onto $\boldsymbol{w}$. A projection of the vector $\boldsymbol{v}$ onto the vector $\boldsymbol{w}$ can be computed by normalizing the vector $\boldsymbol{w}$ to $\boldsymbol{e}_w = \boldsymbol{w}/||\boldsymbol{w}||$ before building the inner product (Figure 7.1).

$$\boldsymbol{v}_w = \langle \boldsymbol{v}, \boldsymbol{e}_w \rangle \boldsymbol{e}_w = \frac{\langle \boldsymbol{v}, \boldsymbol{w} \rangle}{||\boldsymbol{w}||^2} \boldsymbol{w} = \frac{\langle \boldsymbol{v}, \boldsymbol{w} \rangle}{\langle \boldsymbol{w}, \boldsymbol{w} \rangle} \boldsymbol{w} \tag{7.5.20}$$

Subtracting the projected vector $\boldsymbol{v}_w$ from $\boldsymbol{v}$ yields a vector $\boldsymbol{u}$ which is orthogonal to $\boldsymbol{w}$ and satisfies

$$\langle \boldsymbol{u}, \boldsymbol{w} \rangle = \langle \boldsymbol{v} - \boldsymbol{v}_w, \boldsymbol{w} \rangle = 0 \tag{7.5.21}$$

A set of orthonormal vectors $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n\}$ can be constructed out of a set of linear independent vectors $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ by applying the *Gram-Schmidt algorithm* such that

$$\mathrm{span}\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n\} = \mathrm{span}\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\} \tag{7.5.22}$$

A matrix consisting of orthonormal column vectors $\boldsymbol{Q} = (\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n)$ is called *orthogonal matrix*. Analogously, the vectors are said to be $\boldsymbol{A}$-orthogonal to each other if $\langle \boldsymbol{v}, \boldsymbol{Aw} \rangle = 0$. The $\boldsymbol{A}$-projection of two vectors can be obtained similarly by

$$\boldsymbol{v}_w = \frac{\langle \boldsymbol{v}, \boldsymbol{Aw} \rangle}{||\boldsymbol{w}||_{\boldsymbol{A}}^2} \boldsymbol{w} = \frac{\langle \boldsymbol{v}, \boldsymbol{Aw} \rangle}{\langle \boldsymbol{w}, \boldsymbol{Aw} \rangle} \boldsymbol{w} \tag{7.5.23}$$

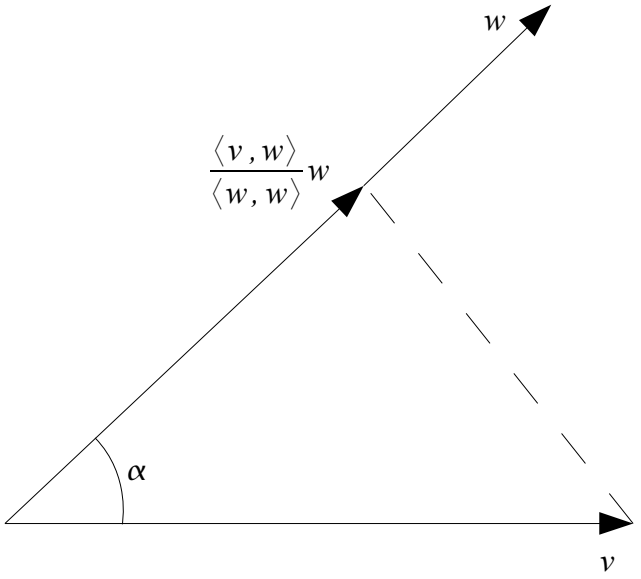


Figure 7.1: Vector projection of $\boldsymbol{v}$ onto $\boldsymbol{w}$

7.6 The projection technique

The projection technique attempts to find a solution for a given linear set of equations as a part of a *search subspace* [Saad, 2003] $\mathcal{K}_m$, where m denotes the dimension of this space. To constrict the exact solution, a set of constraints must be imposed and conditions must be hold. Commonly used techniques try to find a set of m orthogonal conditions, more precisely a set of m linearly independent vectors, which span a new subspace $\mathcal{L}$. This subspace is commonly called *subspace of constraints* or *left subspace* [Saad, 2003]. Techniques using the left subspace $\mathcal{L}$ as the search subspace $\mathcal{K}$ are called *orthogonal projection techniques*. Those techniques, where $\mathcal{L}$ differs from $\mathcal{K}$ are called *oblique projection methods*. For most techniques the residual vector $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$ is used to form the left subspace $\mathcal{L}$, being orthogonal to a set of m linearly independent vectors. If a $n \times n$ matrix $\boldsymbol{A}$, as a part of the linear equation system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, and two m-dimensional subspaces $\mathcal{K}$ and $\mathcal{L}$ are considered, then the projection techniques finds an approximated solution $\tilde{\boldsymbol{x}} = \boldsymbol{x}_0 + \boldsymbol{\delta}$, which belongs to the affine subspace $\boldsymbol{x}_0 + \mathcal{K}$ and whose residual vector is orthogonal to the subspace $\mathcal{L}$. $\boldsymbol{x}_0$ defines the initial guess (Figure 7.2). One can say

$$\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{x}} \perp \mathcal{L} \Longleftrightarrow \boldsymbol{r}_0 - \boldsymbol{A}\boldsymbol{\delta} \perp \mathcal{L} \tag{7.6.24}$$

where $\boldsymbol{\delta}$ belongs to $\mathcal{K}$. Each vector $\boldsymbol{w}$ as part of $\mathcal{L}$ satisfies the relation

$$\langle \boldsymbol{r}_0 - \boldsymbol{A}\boldsymbol{\delta}, \boldsymbol{w} \rangle = \langle \tilde{\boldsymbol{r}}, \boldsymbol{w} \rangle = 0 \tag{7.6.25}$$

The update of the approximated solution $\boldsymbol{x}_{j+1}$ and the residual $\boldsymbol{r}_{j+1}$ can be formulated as

$$\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \boldsymbol{\delta} \tag{7.6.26}$$

$$\boldsymbol{r}_{j+1} = \boldsymbol{r}_j - \boldsymbol{A}\boldsymbol{\delta} \tag{7.6.27}$$

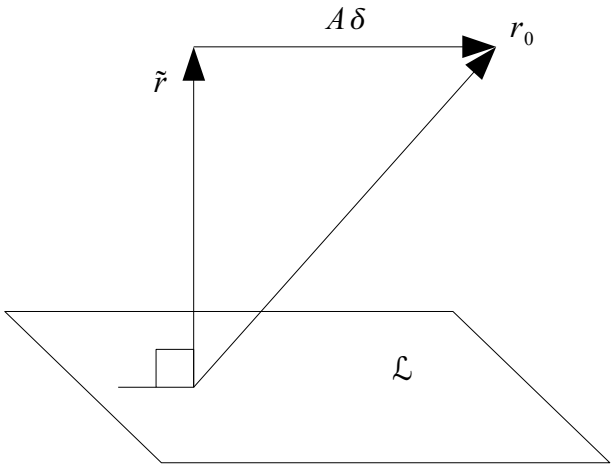


Figure 7.2: Orthogonal projection condition [Saad, 2003]

7.7 Krylov subspace methods

For an estimated solution $\boldsymbol{x}_0$ of a linear equation system the residual $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$ spans the so-called *Krylov (sub)space* for a matrix $\boldsymbol{A}$ with the base $\boldsymbol{r}_0$, given by

$$\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0) = \text{span}\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \boldsymbol{A}^2\boldsymbol{r}_0, \ldots, \boldsymbol{A}^{m-1}\boldsymbol{r}_0\} = \phi_{m-1}(\boldsymbol{A})\boldsymbol{r}_0 \tag{7.7.28}$$

where ϕ_m denotes a certain polynomial of degree m. An improved solution $\boldsymbol{x}_m$ for the initial guess $\boldsymbol{x}_0$ can be obtained as a part of the set $\boldsymbol{x}_0 + \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$

$$\boldsymbol{x}_m \in \boldsymbol{x}_0 + \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0) = \boldsymbol{x}_0 + \sum_{i=0}^{m-1} \boldsymbol{A}^i \boldsymbol{r}_0 \tag{7.7.29}$$

when the so-called *Galerkin condition* ensures that the m-residual, $\boldsymbol{r}_m = \boldsymbol{A}^m \boldsymbol{r}_0$, is an orthogonal projection onto all vectors in the left subspace $\mathcal{L}_m$ (Figure 7.2).

$$\boldsymbol{r}_m \perp \mathcal{L}_m \tag{7.7.30}$$

After each step, $\boldsymbol{x}_m$ gets closer to the exact solution $\boldsymbol{x}_*$ for a concrete chosen subspace $\mathcal{K}_m$. From this idea several methods have been born out to approach the exact solution iteratively. Very common methods, using the Krylov subspace, will be shown in the next sections.

7.7.1 Jacobi iteration method

One of the simplest iterative solver is the *Jacobi iteration solver*, which is a diagonal preconditioned fixed point iteration, trying to approach a fixed point, here the final solution of the matrix system, by an iterative function. An equation system given as (7.1.2) can be transformed, as explained by Gutknecht [2007], to

$$\boldsymbol{x} = \hat{\boldsymbol{B}}\boldsymbol{x} + \hat{\boldsymbol{b}} \tag{7.7.31}$$

where

$$\begin{aligned}\hat{\boldsymbol{B}} &= \boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{A} \\ \hat{\boldsymbol{b}} &= \boldsymbol{D}^{-1}\boldsymbol{b}\end{aligned}$$

and $\boldsymbol{D}$ is the diagonal matrix of $\boldsymbol{A}$. The fixed point iteration $x_{n+1} = \hat{\boldsymbol{B}}x_n + \hat{b}$ will converge to the exact solution, $x_* = \boldsymbol{A}^{-1}b$, for each start vector x_0 chosen, as long as $\rho(\hat{\boldsymbol{B}}) < 1$, which is the *spectral radius* of $\hat{\boldsymbol{B}}$ and is defined as

$$\rho(\hat{\boldsymbol{B}}) = \max\{|\lambda| \mid \lambda \text{ eigenvalue of } \hat{\boldsymbol{B}}\} \tag{7.7.32}$$

7.7.2 One-dimensional projection processes

In general one-dimensional projection processes are given if the update of the approximated solution is done with respect to only one dimension. In other words an update is of the form

$$\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \alpha_j \boldsymbol{v}_j \tag{7.7.33}$$

where the vectors $\boldsymbol{v}_i$ form the base of the search subspace $\mathcal{K}_m = \text{span}\{\boldsymbol{v}_0, \ldots, \boldsymbol{v}_{m-1}\}$. The left subspace is spanned by the vectors $\boldsymbol{w}_i$, $\mathcal{L}_m = \text{span}\{\boldsymbol{w}_0, \ldots, \boldsymbol{w}_{m-1}\}$, respectively. The Petrov-Galerkin condition, where

$$\boldsymbol{r} - \boldsymbol{A}\delta \perp \mathcal{L} \iff \boldsymbol{r}_j - \boldsymbol{A}\delta_j \perp \boldsymbol{w}_j \iff \boldsymbol{r}_j - \alpha\boldsymbol{A}\boldsymbol{v}_j \perp \boldsymbol{w}_j, \tag{7.7.34}$$

yields the optimal value for the scalar value α,

$$\alpha = \frac{\langle \boldsymbol{r}_j, \boldsymbol{w}_j \rangle}{\langle \boldsymbol{A}\boldsymbol{v}_j, \boldsymbol{w}_j \rangle} \tag{7.7.35}$$

The reoccurrence of $\boldsymbol{r}_{j+1}$ can be expressed by

$$\begin{aligned}\boldsymbol{r}_{j+1} &= \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{j+1} && (7.7.36)\\ &= \boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x}_j + \alpha_j \boldsymbol{v}_j) && (7.7.37)\\ &= \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_j - \alpha_j \boldsymbol{A}\boldsymbol{v}_j && (7.7.38)\\ &= \boldsymbol{r}_j - \alpha_j \boldsymbol{A}\boldsymbol{v}_j && (7.7.39)\end{aligned}$$

Steepest descent method

This orthogonal projection method, one of the one-dimensional projection processes, using $\boldsymbol{v} = \boldsymbol{w} = \boldsymbol{r}$, tries to improve a foregoing solution by

$$\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \alpha_j \boldsymbol{v}_j = \boldsymbol{x}_j + \alpha_j \boldsymbol{r}_j \tag{7.7.40}$$

where α_j denotes a new parameter which needs to be found for the best improvement. The residual for this approach can be computed by

$$\boldsymbol{r}_{j+1} = \boldsymbol{r}_j - \alpha_j \boldsymbol{A}\boldsymbol{r}_j \tag{7.7.41}$$

For an optimal step size α_j the 2-norm of $\boldsymbol{r}_{j+1}$ has to be minimized. For symmetric positive definite matrices $\boldsymbol{r}_{j+1}$ can be minimized by minimizing the $\boldsymbol{A}$-norm of the error vector $||\boldsymbol{d}_{j+1}||_{\boldsymbol{A}} = ||\boldsymbol{x}_{j+1} - \boldsymbol{x}_*||_{\boldsymbol{A}}$ and the optimal value for α_j can be obtained by

$$\alpha_j = \frac{\langle \boldsymbol{r}_j, \boldsymbol{r}_j \rangle}{\langle \boldsymbol{r}_j, \boldsymbol{A}\boldsymbol{r}_j \rangle} \tag{7.7.42}$$

More illustrative is a graphical representation of this algorithm (Figure 7.3), where in this case for convenience a 2×2-dimensional equation system given by

$$\begin{pmatrix} 7 & -2 \\ -2 & 6 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 6 \\ 2 \end{pmatrix} \text{ and } \boldsymbol{x}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{7.7.43}$$

is solved.

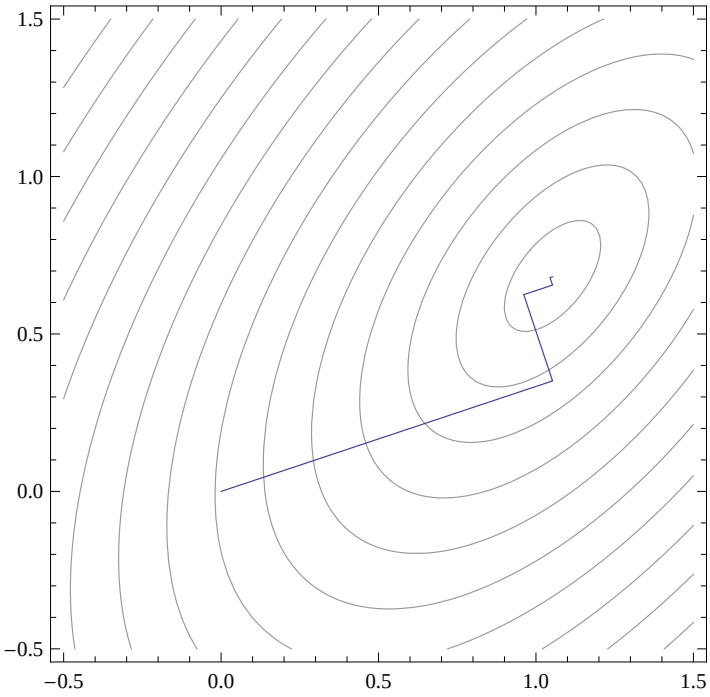


Figure 7.3: Steepest descent algorithm

By simplification of a $N \times N$ matrix to a 2×2 matrix it is easily seen how this algorithm works. For matrices of size $N = 2$, the length of the residual is building concentric ellipses, concentric ellipsoids for $N = 3$ and higher dimensional objects for larger N.

MINRES Iteration

Similar to the steepest descent, this one-dimensional projection process uses $\boldsymbol{v} = \boldsymbol{r}$ and $\boldsymbol{w} = \boldsymbol{A}\boldsymbol{r}$. This method is also commonly called *Orthomin(1)* and relaxes the symmetric property assumption of the matrix needed by the steepest descent. However, the matrix has to be positive definite. The scalar value α is minimized in the sense of the $\boldsymbol{A}$-norm of the residual $||\boldsymbol{r}_{j+1}||_{\boldsymbol{A}} = ||\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{j+1}||_{\boldsymbol{A}}$ in the direction of the $\boldsymbol{r}_{j+1}$ and one gets

$$\alpha_j = \frac{\langle \boldsymbol{r}_j, \boldsymbol{A}\boldsymbol{r}_j \rangle}{\langle \boldsymbol{A}\boldsymbol{r}_j, \boldsymbol{A}\boldsymbol{r}_j \rangle} \tag{7.7.44}$$

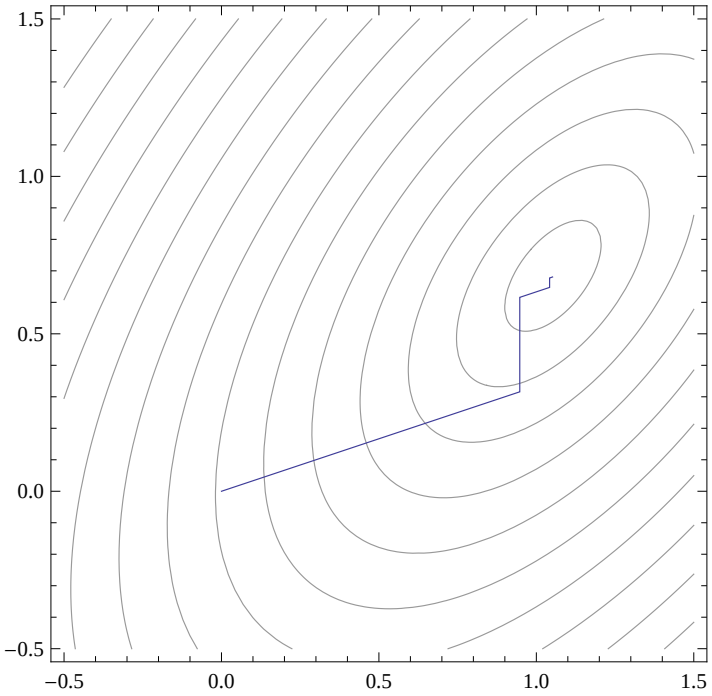


Figure 7.4: MR Iteration algorithm

Residual norm steepest descent method

This is the last introduced one-dimensional projection method, where the vectors are defined as $\boldsymbol{v} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{r}$ and $\boldsymbol{w} = \boldsymbol{A}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{r}$. This method is the same as the MINRES iteration method, using a left preconditioner $\boldsymbol{C}_L = \boldsymbol{A}^{\mathrm{T}}$ which reformulates the equation system to

$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{b} \tag{7.7.45}$$

The matrix product $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ results in a positive definite matrix for each non-singular matrix $\boldsymbol{A}$. So the need that $\boldsymbol{A}$ is positive definite will be relaxed by this approach. The value α yields

$$\alpha_j = \frac{\langle \boldsymbol{r}_j, \boldsymbol{A}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{r}_j \rangle}{\langle \boldsymbol{A}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{r}_j, \boldsymbol{A}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{r}_j \rangle} = \frac{\langle \boldsymbol{A}^{\mathrm{T}} r_j, \boldsymbol{A}^{\mathrm{T}}\boldsymbol{r}_j \rangle}{\langle \boldsymbol{A}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{r}_j, \boldsymbol{A}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{r}_j \rangle} \tag{7.7.46}$$

7.7.3 Conjugate gradient method (CG)

This method is an improvement of the steepest descent algorithm, which develops steps in a more accurate direction towards the final solution. A restriction is that this method works only for symmetric positive definite (spd) or hermitian positive definite (hpd) matrices. Compared to the method of the steepest descent, where an improvement of an actual solution $\boldsymbol{x}_j$ depends on the actual residual (7.7.40), this algorithm introduces direction vectors $\boldsymbol{p}_j$ instead to improve the solution. The reoccurrence of the solution is defined as

$$\boldsymbol{x}_{j+1} = \boldsymbol{x}_j - \alpha_j \boldsymbol{p}_j \tag{7.7.47}$$

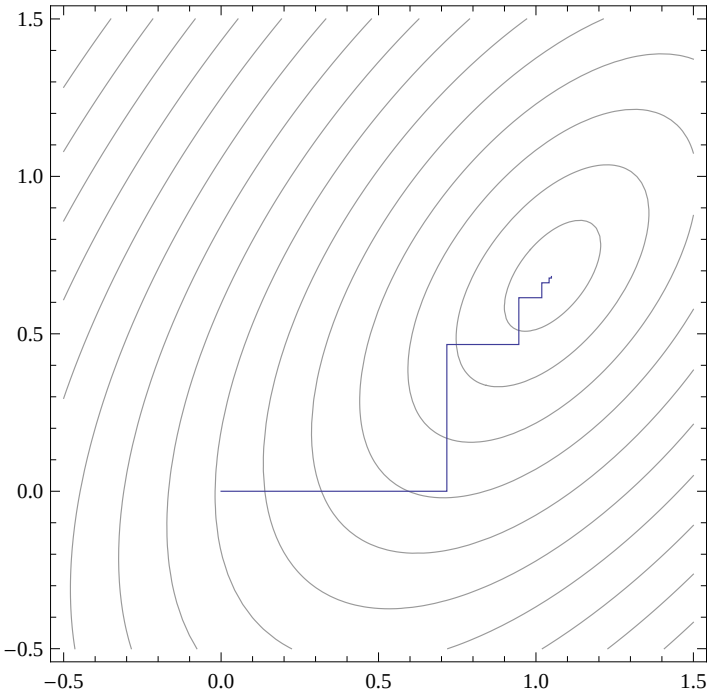


Figure 7.5: Residual norm steepest descent algorithm

which would be the same as the steepest descent if $\boldsymbol{p} = \boldsymbol{r}$. The actual residual can then be expressed as

$$\boldsymbol{r}_{j+1} = \boldsymbol{r}_j - \alpha_j \boldsymbol{A}\boldsymbol{p}_j \tag{7.7.48}$$

For a 2×2 matrix, compared to the algorithm of the steepest descent, it is easy to see that a faster and stabler solution algorithm can be obtained by adding a second direction vector conjugate or $\boldsymbol{A}$-orthogonal to the first. So only two steps are needed to calculate the final result as shown in figure 7.6.

Expanding this situation back to matrices of arbitrary size N, direction vectors $\boldsymbol{p}_j$ are chosen which are conjugate to each other. The conjugate direction vectors follow

$$\langle \boldsymbol{p}_j, \boldsymbol{A}\boldsymbol{p}_i \rangle = 0 \mid i \neq j \tag{7.7.49}$$

α_j defines the size of the optimal step length into the direction of $\boldsymbol{p}_j$ and it minimizes the $\boldsymbol{A}$-norm of the error vector over a two-dimensional affine space when

$$\alpha_j = \frac{\langle \boldsymbol{r}_j, \boldsymbol{r}_j \rangle}{\langle \boldsymbol{p}_j, \boldsymbol{A}\boldsymbol{p}_j \rangle} \tag{7.7.50}$$

For the conjugate gradient method the direction vectors $\boldsymbol{p}_j$ are then determined by the Krylov space $\mathcal{K}_{j+1}(\boldsymbol{A}, \boldsymbol{r}_0)$.

$$\operatorname{span}\{\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_n\} = \operatorname{span}\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \boldsymbol{A}^2\boldsymbol{r}_0, \ldots, \boldsymbol{A}^n\boldsymbol{r}_0\} = \mathcal{K}_{n+1}(\boldsymbol{A}, \boldsymbol{r}_0) \tag{7.7.51}$$

Similar to the method of the steepest descent, an update of the direction vector $\boldsymbol{p}_{j+1}$ can be achieved by subtracting the $\boldsymbol{A}$-projection of the error onto the affine subspace of $\boldsymbol{r}_{j+1}$ in the direction of $\boldsymbol{p}_j$

$$\boldsymbol{p}_{j+1} = \boldsymbol{r}_{j+1} - \frac{\langle \boldsymbol{r}_{j+1}, \boldsymbol{A}\boldsymbol{p}_k \rangle}{\langle \boldsymbol{p}_j, \boldsymbol{A}\boldsymbol{p}_j \rangle} \tag{7.7.52}$$

This ensures, that the direction vectors are conjugate to each other in the sense of equation (7.7.49).

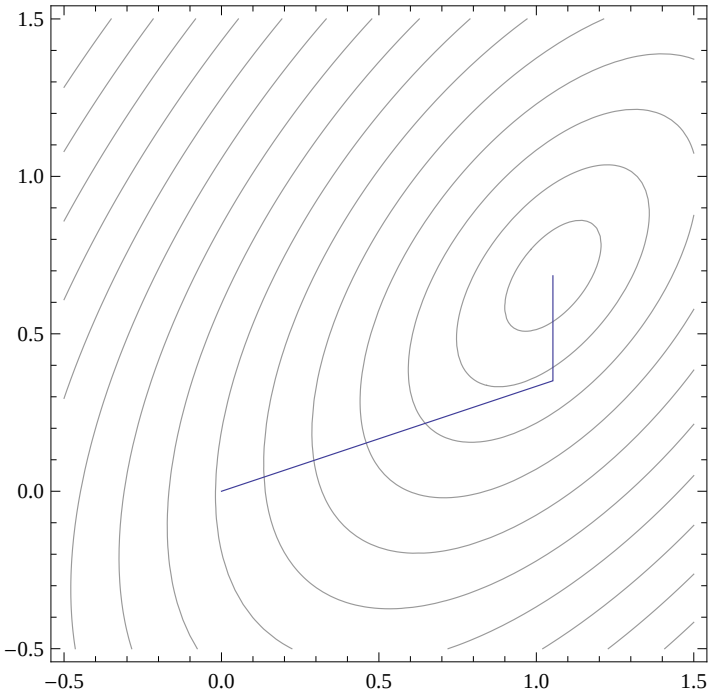


Figure 7.6: Conjugate gradient algorithm

7.7.4 Biconjugate gradient method (BiCG)

The bi-conjugate gradient method uses, in contrast to the conjugate gradient method, a second set of residuals, the so-called *shadow residuals* $\tilde{\boldsymbol{r}}_n$ to approach the final solution from two sides. This method is also usable for non-symmetric matrices, but needs, compared to the CG algorithm, two matrix-vector multiplications to extend the Krylov spaces $\mathcal{K}_n$ and $\tilde{\mathcal{K}}_n$.

$$\boldsymbol{r}_j = \boldsymbol{A}^j \boldsymbol{r}_0 \tag{7.7.53}$$
$$\tilde{\boldsymbol{r}}_j = (\boldsymbol{A}^T)^j \tilde{\boldsymbol{r}}_0 \tag{7.7.54}$$

For the Krylov spaces and the shadowed Krylov spaces, one gives

$$\mathrm{span}\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \boldsymbol{A}^2\boldsymbol{r}_0, \ldots, \boldsymbol{A}^n\boldsymbol{r}_0\} = \mathcal{K}_{n+1}(\boldsymbol{A}, \boldsymbol{r}_0) \tag{7.7.55}$$
$$\mathrm{span}\{\tilde{\boldsymbol{r}}_0, \boldsymbol{A}^T\tilde{\boldsymbol{r}}_0, (\boldsymbol{A}^T)^2\tilde{\boldsymbol{r}}_0, \ldots, (\boldsymbol{A}^T)^n\tilde{\boldsymbol{r}}_0\} = \mathcal{K}_{n+1}(\boldsymbol{A}^T, \tilde{\boldsymbol{r}}_0) = \tilde{\mathcal{K}}_{n+1}(\boldsymbol{A}, \tilde{\boldsymbol{r}}_0) \tag{7.7.56}$$

where $\tilde{\boldsymbol{r}}_0$ can be chosen freely providing $\langle \boldsymbol{r}_0, \tilde{\boldsymbol{r}}_0 \rangle \neq 0$. The residuals $\boldsymbol{r}$ and $\tilde{\boldsymbol{r}}$ and the search directions $\boldsymbol{p}$ and $\tilde{\boldsymbol{p}}$ form bi-orthogonal bases of the Krylov spaces $\mathcal{K}$ and $\tilde{\mathcal{K}}$.

$$\langle \tilde{\boldsymbol{r}}_i, \boldsymbol{r}_j \rangle = 0 \mid i \neq j \tag{7.7.57}$$
$$\langle \tilde{\boldsymbol{p}}_i, \boldsymbol{A}\boldsymbol{p}_j \rangle = 0 \mid i \neq j \tag{7.7.58}$$

Again, the actual residual $\boldsymbol{r}_j$ represents the accuracy of the actual solution $\boldsymbol{x}_j$. Similar to the conjugate gradient method the conjugate direction vectors follow

$$\langle \boldsymbol{p}_i \boldsymbol{A} \boldsymbol{p}_j \rangle = 0 \mid i \neq j \tag{7.7.59}$$
$$\langle \tilde{\boldsymbol{p}}_i \boldsymbol{A}^{\mathrm{T}} \tilde{\boldsymbol{p}}_j \rangle = 0 \mid i \neq j \tag{7.7.60}$$
$$\tag{7.7.61}$$

and the improvement of an actual solution $\boldsymbol{x}_j$ can be calculated by

$$\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \boldsymbol{p}_j \alpha_j \tag{7.7.62}$$

α_j defines the size of the optimal step length into the direction of $\boldsymbol{p}_j$ and is computable by

$$\alpha_j = \frac{\langle \boldsymbol{r}_j, \tilde{\boldsymbol{r}}_j \rangle}{\langle \boldsymbol{A}\boldsymbol{p}_j, \tilde{\boldsymbol{p}}_j \rangle} \tag{7.7.63}$$

7.8 Transpose-free variants

7.8.1 Conjugate gradient squared method (CGS)

A transpose-free variant of the biconjugate gradient method is the conjugate gradient squared method once developed by Sonneveld in 1984 [Sonneveld, 1989]. The main goal was to avoid the multiplication by the transpose of $\boldsymbol{A}$ which occurs in the BiCG algorithm, without incurring additional cost. From BiCG it is known that the residuals and the direction vectors can be expressed by two certain polynomials ϕ_j and π_j of degree j

$$\boldsymbol{r}_j = \phi_j(\boldsymbol{A})\boldsymbol{r}_0 \tag{7.8.64}$$
$$\boldsymbol{p}_j = \pi_j(\boldsymbol{A})\boldsymbol{r}_0 \tag{7.8.65}$$

as well as for the shadowed residuals and the direction vectors

$$\tilde{\boldsymbol{r}}_j = \phi_j(\boldsymbol{A}^{\mathrm{T}})\tilde{\boldsymbol{r}}_0 \tag{7.8.66}$$
$$\tilde{\boldsymbol{p}}_j = \pi_j(\boldsymbol{A}^{\mathrm{T}})\tilde{\boldsymbol{r}}_0 \tag{7.8.67}$$

The step size, introduced in the BiCG method in equation (7.7.63), can now be rewritten as

$$\alpha_j = \frac{\langle \boldsymbol{r}_j, \tilde{\boldsymbol{r}}_j \rangle}{\langle \boldsymbol{A}\boldsymbol{p}_j, \tilde{\boldsymbol{p}}_j \rangle} \tag{7.8.68}$$
$$= \frac{\langle \phi_j(\boldsymbol{A})\boldsymbol{r}_0, \phi_j(\boldsymbol{A}^{\mathrm{T}})\tilde{\boldsymbol{r}}_0 \rangle}{\langle \boldsymbol{A}\pi_j(\boldsymbol{A})\boldsymbol{r}_0, \pi_j(\boldsymbol{A}^{\mathrm{T}})\tilde{\boldsymbol{r}}_0 \rangle} \tag{7.8.69}$$
$$= \frac{\langle \phi_j^2(\boldsymbol{A})\boldsymbol{r}_0, \tilde{\boldsymbol{r}}_0 \rangle}{\langle \boldsymbol{A}\pi_j^2(\boldsymbol{A})\boldsymbol{r}_0, \tilde{\boldsymbol{r}}_0 \rangle} \tag{7.8.70}$$

For a better reading one defines

$$\boldsymbol{r}_{j+1}^{\mathrm{S}} = \phi_{j+1}^2(\boldsymbol{A})\boldsymbol{r}_0 \tag{7.8.71}$$
$$\boldsymbol{p}_{j+1}^{\mathrm{S}} = \pi_{j+1}^2(\boldsymbol{A})\boldsymbol{r}_0 \tag{7.8.72}$$
$$\boldsymbol{q}_{j+1}^{\mathrm{S}} = \phi_{j+1}(\boldsymbol{A})\pi_j(\boldsymbol{A})\boldsymbol{r}_0 \tag{7.8.73}$$

The CGS method finds an iteration sequence for the residual and the direction vectors which satisfies

$$\boldsymbol{r}_{j+1}^{\mathrm{S}} = \boldsymbol{r}_j^{\mathrm{S}} - 2\alpha_j \boldsymbol{A}(\boldsymbol{r}_j^{\mathrm{S}} + \beta_j \boldsymbol{q}_j^{\mathrm{S}}) + \alpha_j^2 \boldsymbol{A}^2 \boldsymbol{p}_j^{\mathrm{S}} \tag{7.8.74}$$
$$\boldsymbol{p}_{j+1}^{\mathrm{S}} = \boldsymbol{r}_{j+1}^{\mathrm{S}} + 2\beta_{j+1}\boldsymbol{q}_{j+1}^{\mathrm{S}} + \beta_{j+1}^2 \boldsymbol{p}_j^{\mathrm{S}} \tag{7.8.75}$$
$$\boldsymbol{q}_{j+1}^{\mathrm{S}} = \boldsymbol{r}_j^{\mathrm{S}} + \beta_j \boldsymbol{q}_j^{\mathrm{S}} - \alpha_j \boldsymbol{A}\boldsymbol{p}_j^{\mathrm{S}} \tag{7.8.76}$$

For many cases this method works quite well as long as rounding errors do not too much damage the algorithm. More details about irregular convergence of the residual can be found in the book of der Vorst [2003] and Saad [2003].

7.8.2 Biconjugate gradient stabilized method (BiCGStab)

A stabilized variant of the CGS method is the biconjugate gradient stabilized method. The residuals and direction vectors produced in this method are using the approach

$$\begin{aligned} \boldsymbol{r}_j &= \psi_j(\boldsymbol{A})\phi_j(\boldsymbol{A})\boldsymbol{r}_0 && (7.8.77)\\ \boldsymbol{p}_j &= \psi_j(\boldsymbol{A})\pi_j(\boldsymbol{A})\boldsymbol{r}_0 && (7.8.78) \end{aligned}$$

$\psi_j(\boldsymbol{A})$ is a new recursively defined polynomial which is used to smooth and stabilize the convergence behavior of the BiCG algorithm and is defined by the recurrence rule

$$\psi_{j+1}(\boldsymbol{A}) = (\boldsymbol{I} - \omega_j \boldsymbol{A})\psi_j(\boldsymbol{A}) \tag{7.8.79}$$

From the BiCG method the update rules for the residual $\boldsymbol{r}_j$ and the direction vector $\boldsymbol{p}_j$ are known as

$$\begin{aligned} \boldsymbol{r}_{j+1} &= \boldsymbol{r}_j - \alpha_j \boldsymbol{A}\boldsymbol{p}_j && (7.8.80)\\ \boldsymbol{p}_{j+1} &= \boldsymbol{r}_j + \beta_j \boldsymbol{p}_j && (7.8.81) \end{aligned}$$

Including the new polynomial $\psi_j(\boldsymbol{A})$ for this stabilized approach, the residual $\boldsymbol{r}_{j+1}$ can then be written as

$$\begin{aligned} \boldsymbol{r}_{j+1} &= \psi_{j+1}(\boldsymbol{A})\phi_{j+1}(\boldsymbol{A})\boldsymbol{r}_0 \\ &= (\boldsymbol{I} - \omega_j \boldsymbol{A})\psi_j(\boldsymbol{A})\phi_{j+1}(\boldsymbol{A})\boldsymbol{r}_0 && (7.8.82)\\ &= (\boldsymbol{I} - \omega_j \boldsymbol{A})(\psi_j(\boldsymbol{A})\phi_j(\boldsymbol{A}) - \alpha_j \boldsymbol{A}\psi_j(\boldsymbol{A})\pi_j(\boldsymbol{A}))\boldsymbol{r}_0 \\ &= (\boldsymbol{I} - \omega_j \boldsymbol{A})(\boldsymbol{r}_j - \alpha_j \boldsymbol{A}\boldsymbol{p}_j) && (7.8.83)\\ &= (\boldsymbol{I} - \omega_j \boldsymbol{A})\boldsymbol{s}_j \\ &= \boldsymbol{s}_j - \omega_j \boldsymbol{A}\boldsymbol{s}_j && (7.8.84) \end{aligned}$$

where $\boldsymbol{s}_j$ is used as a temporary vector for later usage and is defined as

$$\boldsymbol{s}_j \equiv \boldsymbol{r}_j - \alpha_j \boldsymbol{A}\boldsymbol{p}_j \tag{7.8.85}$$

Adapting the equation (7.8.81) for the new approach yields, for the direction vectors

$$\begin{aligned} \boldsymbol{p}_{j+1} &= \psi_{j+1}(\boldsymbol{A})\pi_{j+1}(\boldsymbol{A})\boldsymbol{r}_0 \\ &= \psi_{j+1}(\boldsymbol{A})(\phi_{j+1} + \beta_j \pi_j(\boldsymbol{A}))\boldsymbol{r}_0 \\ &= (\psi_{j+1}(\boldsymbol{A})\phi_{j+1}(\boldsymbol{A}) + \beta_j \psi_{j+1}(\boldsymbol{A})\pi_j(\boldsymbol{A}))\boldsymbol{r}_0 \\ &= (\psi_{j+1}(\boldsymbol{A})\phi_{j+1}(\boldsymbol{A}) + \beta_j (\boldsymbol{I} - \omega_j \boldsymbol{A})\psi_j(\boldsymbol{A})\pi_j(\boldsymbol{A}))\boldsymbol{r}_0 \\ &= \boldsymbol{r}_{j+1} + \beta_j (\boldsymbol{I} - \omega_j \boldsymbol{A})\boldsymbol{p}_j && (7.8.86) \end{aligned}$$

The scalar α_j representing the step size into the direction $\boldsymbol{p}_j$ can be expressed as [Saad, 2003]

$$\alpha_j = \frac{\langle \boldsymbol{r}_j, \tilde{\boldsymbol{r}}_0 \rangle}{\langle \boldsymbol{A}\boldsymbol{p}_j, \tilde{\boldsymbol{r}}_0 \rangle} \tag{7.8.87}$$

β_j can be computed by

$$\beta_j = \frac{\langle \boldsymbol{r}_{j+1}, \tilde{\boldsymbol{r}}_0 \rangle}{\langle \boldsymbol{r}_j, \tilde{\boldsymbol{r}}_0 \rangle} \times \frac{\alpha_j}{\omega_j} \tag{7.8.88}$$

The last scalar value ω_j can be viewed as a new additional parameter being also responsible to achieve a steepest descent in the direction of the residual. In the book of Saad [2003] the parameter is chosen

to minimize the norm of the vector $\boldsymbol{r}_{j+1}$ defined by equation (7.8.82). The value for the optimal step ω_j is given by

$$\omega_j = \frac{\langle \boldsymbol{A}\boldsymbol{s}_j, \boldsymbol{s}_j \rangle}{\langle \boldsymbol{A}\boldsymbol{s}_j, \boldsymbol{A}\boldsymbol{s}_j \rangle} \tag{7.8.89}$$

The update rule for the residual $\boldsymbol{r}_{j+1}$ from equation (7.8.84) to the next iteration step can also be written as

$$\boldsymbol{r}_{j+1} = \boldsymbol{s}_j - \omega_j \boldsymbol{A}\boldsymbol{s}_j = \boldsymbol{r}_j - \alpha_j \boldsymbol{A}\boldsymbol{p}_j - \omega_j \boldsymbol{A}\boldsymbol{s}_j \tag{7.8.90}$$

which results finally for the improved solution $\boldsymbol{x}_{j+1}$ in

$$\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \alpha_j \boldsymbol{p}_j + \omega_j \boldsymbol{s}_j \tag{7.8.91}$$

Chapter 8

Haptic device

8.1 Introduction

The word *haptic*, which might come from the Greek word *haptesthai*, means to grasp or to touch or the other Greek word *haptikos*, which can be translated as being able to come into contact with [Dontschewa et al., 2004]. The touch sense of a human being is often understood as a haptic sense. The touch sense is the first sense developed in the fetus and is nearly as important as the visual sense for a human being, and has been therefore the target of much research in this topic. Physiologically, the haptic sense is an interpretation of a perceptual channel based input from the skin by which an individual gets information about its environment and its body. Generally, a haptic system can be defined as a combination of tactile and kinesthetic senses used in a mechanical interaction with the environment. The haptic device is an apparatus to produce such tactile and kinesthetic senses for an user by rendering forces for a target to give an intuitive and better feeling while working in virtual or augmented reality. The device sends its position and orientation to the augmented reality application which reacts with information about forces rendered by the device. However, such haptic devices have limitations in their workspace, caused by necessary hardware links and joints. The costs for those devices are widely ranged from 2.000€ up to 50.000€. However, haptic feedback is gaining widespread acceptance as an important part for virtual and augmented reality systems, by adding the possibility of feeling objects next to the visual aspects of information transport from the system to the user. It opens a new way to gain 3-D impressions of a virtual system, which is partly already done by using the stereo property for the visual feedback by shutter glasses or glasses with polarization filters. The user can feel 3-D virtual objects like a blind man does and his brain automatically generates a 3-D impression of the surrounding environment. In human-computer interaction, haptic feedback means both tactile and force feedback. Tactile, or touch feedback, is the term applied to sensations felt by the skin. Tactile feedback allows users to feel things such as the texture of surfaces, temperatures and vibration. Force feedback reproduces directional forces that can result from solid boundaries, the weight of grasped virtual objects, mechanical compliance of an object and inertia [Berkley, 2003]. The human sense of touch involves a closed loop system of receptors sensing and transmitting messages to and from the brain, thinking and manipulating. Haptic interfaces require a similar system, but one that is electromechanical and computer-based.

8.2 SensAble™

A company which should be acknowledged here is the American company SensAble™. They developed some different types of haptic devices and software drivers. Last but not least, because the

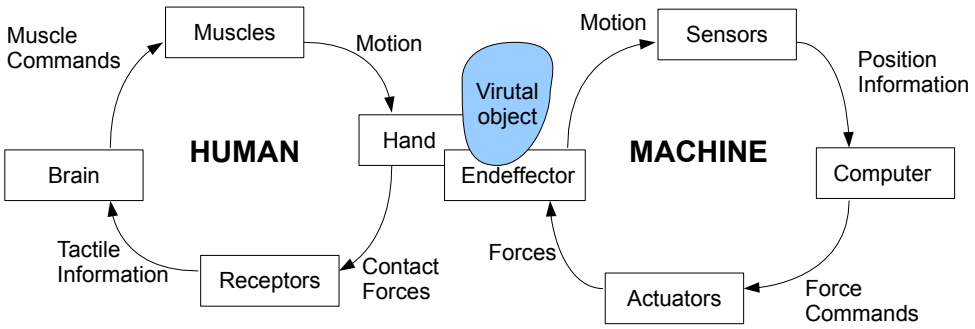


Figure 8.1: Interaction loop between human and machine

device used in this paper was developed by SensAble™. SensAble™ Technologies is a privately-owned company based in Woburn, Massachusetts, USA which provides software and devices that add the sense of touch to the digital world, including 3D touch-enabled modeling systems and the PHANTOM® line of haptic devices and the OpenHaptics™ toolkit.
SensAble™ modeling systems are used for product design, medical and dental modeling, digital content creation, and fine arts. The PHANTOM® force-feedback devices, which enable users to touch and manipulate virtual objects, and the developer toolkit, are used for simulation and training, robotics, and third-party development. The first PHANTOM® haptic device was designed and built in the early 1990s and SensAble™ was formally incorporated in 1993.

8.3 Haptic feedback devices

A lot of force feedback devices are found in the computer gaming industry. A few of them are listed below. Some low-end haptic devices are already commonly available. Some joysticks and game controllers provide haptic feedback, commonly marketed as force feedback. The simplest form is the Rumble Pak, which is simply an attachment which vibrates upon command from the software.
Force feedback wheels attempt to recreate the force felt by drivers in real cars. They are used on computer and console racing simulators. These wheels vary in quality and realism and are manufactured by Logitech and other companies. It allows players to feel the road, all the bumps, car handling and crashes, thus making the game more realistic. The ability to change the temperature of a controlling device could also be used. However, the technology may be cost prohibitive in terms of how much power it would need to operate properly.
When the PlayStation 2 computer entertainment system was introduced, the controller included was manufactured with two additional vibration levels, was considerably lighter and most of the buttons were pressure sensitive.

8.4 PHANTOM® devices

In comparison to the game controllers, the haptic devices like the PHANTOM® Omni™ are used more in research areas, like medicine, construction and planning. The SensAble Technologies PHANTOM® product line of haptic devices makes it possible for users to touch and manipulate virtual objects. The

PHANTOM® Omni™ model is the most cost-effective haptic device available today. Portable design, compact footprint, and IEEE-1394, a FireWire port interface, ensure quick installation and ease-of-use (Figure 8.2). Technical information about the PHANToM® Omni™ can be found in Table 8.1. To compare these property values, a second and better version of a Phantom from SensAble is listed here. The PHANTOM® 3.0/6DOF (Figure 8.3) device allows users to explore application areas that require force feedback in six degrees of freedom (6DOF). Simulating torque force feedback makes it possible to feel the collision and reaction forces and torques. Its technical data are listed in Table 8.2. Other haptic devices are shown in Figure 8.4.

Figure 8.2: Phantom® Omni™

Force feedback workspace	160 W x 120 H x 70 D mm
Weight (device only)	2.8 kg
Range of motion	Hand movement pivoting at wrist
Nominal position resolution	0.055 mm.
Maximum exertable force at nominal position	3.3 N
Continuous exertable force	0.88 N
Stiffness	X axis 1.26 N/mm Y axis 2.31 N/mm Z axis 1.02 N/mm
Inertia (apparent mass at tip)	45 g
Force feedback	x, y, z
Position sensing	x, y, z, yaw, pitch, roll
Interface	IEEE-1394 FireWire port
Supported platforms	Intel-based PCs

Table 8.1: PHANTOM® Omni™ technical specification

Figure 8.3: Force feedback PHANTOM® 3.0/6DOF

Force feedback workspace	Translational [mm] (WxHxD)	160x120x70
	Rotational [°] (Y/P/R)	297/260/335
Weight (device only)	Detachable portion	7.5 kg
	Electronics console	24.0 kg
Range of motion	Full arm movement pivoting at shoulder	
Nominal position resolution	Translational [mm]	0.02
	Rotational [°] (Y/P/R)	0.0023/0.0023/0.0080
Maximum exertable force and	Translational	22N
torque at nominal position	Rotational [mNm] (Y/P/R)	515/515/170
Continuous exertable force and	Translational	3N
torque at nominal position	Rotational [mNm] (Y/P/R)	188/188/48
Stiffness	1 N/mm	
Force feedback	x, y, z, yaw, pitch, roll	
Position sensing	x, y, z, yaw, pitch, roll	
Interface	Parallel port	
Supported platforms	Intel-based PCs	

Table 8.2: PHANTOM® 3.0/6DOF technical specification

(a) Phantom® Desktop

(b) Novint Falcon

Figure 8.4: Other haptic feedback devices

Part III

Contribution

Chapter 9

Extension to the boundary element method

This section gives an 1-D example for the boundary element method and explains the needed reformulation of the boundary integral equation, since this equation is written in a way being not well suited for an implementation of the linear case approach. Further it explains how the collocation method must be adapted for the reformulated integral equation.

9.1 Elastostatic example

For this example it must be said, that the term boundary element method is not really adequate, since the boundary in a 1-D example is represented by only two points, which further also simplifies the integration over the boundary elements, i.e. over these points. In this example a rod will be loaded with a force as shown in Figure 9.1. Furthermore, linear-elastic deformations, which means the material returns to its initial state after deforming, are assumed. The bending line or elastic curve follows the beam-type differential equation

$$\begin{aligned} -EI_z w''''(x) &= 0 \\ -EI_z w'''(x) &= Q(x) \\ -EI_z w''(x) &= M(x) \\ -w'(x) &= \tan\alpha(x) \end{aligned} \tag{9.1.1}$$

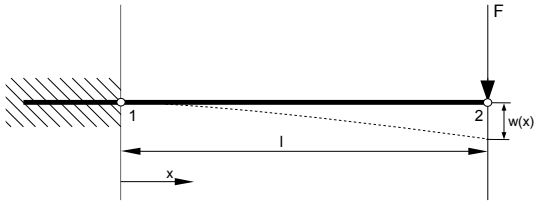


Figure 9.1: Rod loaded with a force

with $w(x)$ as displacement, $M_z(x)$ as momentum function around z and $Q(x)$ as normal load force situated in y-direction. $\alpha(x)$ can be seen as the bending angle at x. E indicates the Young's modulus and is a material constant. I_z is denoted as the geometrical moment of inertia against moments around z and depends on the geometry cross section in x-direction. The boundary conditions for this example are obvious and are set to

$$\begin{aligned} w(0) &= 0 \\ \tan\alpha(0) &= 0 \\ M(l) &= 0 \\ Q(l) &= F \end{aligned}$$

An analytical solution can be found after integrating equation (9.1.1) three times

$$\begin{aligned} -EI_z w(x) &= \iiint Q(x)\mathrm{d}^3x \\ &= \frac{Qx^3}{6} + C_1\frac{x^2}{2} + C_2 x + C_3 \end{aligned}$$

and applying the boundary conditions to determine the integration constants C_1, C_2 and C_3 which finally results in

$$w(x) = \frac{Flx^2}{2EI_z} - \frac{Fx^3}{6EI_z} \tag{9.1.2}$$

Before one can apply the boundary element method it is necessary to calculate the fundamental solution $w^*(\xi, x)$ of the situated problem. Recall section 3.6, this can be done by solving the governing differential equation of the beam-type as follows

$$\begin{aligned} -EI_z w_*''''(x,\xi) &= \delta(x-\xi) \\ -EI_z w_*'''(x,\xi) &= -H(x-\xi) + C_1 \;\; (C_1 = \frac{1}{2}) \\ -EI_z w_*''(x,\xi) &= \frac{x}{2} - (x-\xi)H(x-\xi) + C_2 \;\; (C_2 = -\frac{\xi}{2}) \\ -EI_z w_*'(x,\xi) &= \frac{x^2+2x\xi}{4} - \frac{(x-\xi)^2}{2}H(x-\xi) + C_3 \;\; (C_3 = \frac{\xi^2}{4}) \\ -EI_z w^*(x,\xi) &= \frac{(x-\xi)^3}{12} - \frac{(x-\xi)^3}{6}H(x-\xi) + C_4 \;\; (C_4 = 0) \\ w^*(x,\xi) &= \frac{(x-\xi)^3}{6EI_z}H(x-\xi) - \frac{(x-\xi)^3}{12EI_z} \end{aligned}$$

$H(x)$ denotes the Heaviside function (Appendix B). The fundamental solutions of Q^*, M_x^* and $\tan\alpha^*$ can be calculated out of $w^*(x,\xi)$ following the equations above and results in

$$\begin{aligned} Q^*(x,\xi) &= -EI_z\tfrac{\partial^3}{\partial\xi^3}w^*(x,\xi) &&= H(x-\xi) - \frac{1}{2} \\ M^*(x,\xi) &= -EI_z\tfrac{\partial^2}{\partial\xi^2}w^*(x,\xi) &&= (x-\xi)H(x-\xi) - \frac{x-\xi}{2} \\ \tan\alpha^*(x,\xi) &= -\tfrac{\partial}{\partial\xi}w^*(x,\xi) &&= \frac{(x-\xi)^2}{2}H(x-\xi) - \frac{(x-\xi)^2}{4} \end{aligned}$$

Evaluation of the beam-type operator $\frac{\partial^4()}{\partial x^4}$ following Section 3.5 will lead to a representation formula of the type

$$w(x) = [w^*(x,\xi)Q(\xi) - \tan\alpha^*(x,\xi)M(\xi) + M^*(x,\xi)\tan\alpha(\xi) - Q^*(x,\xi)w(\xi)]_{\xi=0}^{l} \tag{9.1.3}$$

After inserting the defined boundary values this leads to two equations for the boundary, $x = 0$ and $x = l$, with four unknown variables, namely $Q(0)$, $M(0)$, $\tan\alpha(l)$ and $w(l)$. To solve it two additional equations are needed. They can be found by solving

$$\begin{aligned} M''(x) &= 0 \\ M'(x) &= Q(x) \end{aligned}$$

as a part of the beam-type and leads to its representation formula

$$M(x) = [M^*(x,\xi)Q(\xi) - Q^*(x,\xi)M(\xi)]_{\xi=0}^{l} \tag{9.1.4}$$

Now four equations with four unknown variables are given for the boundary $x = 0$ and $x = l$

$$\begin{pmatrix} w(0) \\ w(l) \end{pmatrix} = \begin{pmatrix} 0 & \frac{l^3}{12EI_z} \\ -\frac{l^3}{12EI_z} & 0 \end{pmatrix} \begin{pmatrix} Q(0) \\ Q(l) \end{pmatrix} - \begin{pmatrix} 0 & \frac{l^2}{4EI_z} \\ \frac{l^2}{4EI_z} & 0 \end{pmatrix} \begin{pmatrix} M(0) \\ M(l) \end{pmatrix} + \begin{pmatrix} 0 & \frac{l}{2} \\ -\frac{l}{2} & 0 \end{pmatrix} \begin{pmatrix} \tan\alpha(0) \\ \tan\alpha(l) \end{pmatrix} + \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} w(0) \\ w(l) \end{pmatrix}$$

from equation (9.1.3) and

$$\begin{pmatrix} M(0) \\ M(l) \end{pmatrix} = \begin{pmatrix} 0 & \frac{l}{2} \\ -\frac{l}{2} & 0 \end{pmatrix} \begin{pmatrix} Q(0) \\ Q(l) \end{pmatrix} + \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} M(0) \\ M(l) \end{pmatrix}$$

from equation (9.1.4). Here it needs to be mentioned, that the Heaviside function in $Q^*(x,\xi)$ has to be taken from the correct side, the side from where x reaches ξ, of the discontinuity (Appendix B). This equations can be rearranged and combined to

$$\frac{1}{12} \begin{pmatrix} 0 & -\frac{l^3}{EI_z} & 0 & \frac{3l^2}{EI_z} \\ \frac{l^3}{EI_z} & 0 & \frac{3l^2}{EI_z} & 0 \\ 0 & 6l & -6 & 6 \\ -6l & 0 & 6 & -6 \end{pmatrix} \begin{pmatrix} Q(0) \\ Q(l) \\ M(0) \\ M(l) \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 0 & l & -1 & 1 \\ -l & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \tan\alpha(0) \\ \tan\alpha(l) \\ w(0) \\ w(l) \end{pmatrix} \tag{9.1.5}$$

This equation represents now the interaction of the nodal boundary values, normal load force, momentum, bending angle and bending. For each node, 1 and 2, the boundary condition can be defined and the values can be set. The known boundary values for this example are defined above. After application the system can be rearranged again, putting known values on the right hand side and unknown values on the left hand side and can be solved in a straightforward manner for the unknown values.

$$\begin{aligned} \frac{1}{12} \begin{pmatrix} 0 & -6l & 0 & -6 \\ \frac{l^3}{EI_z} & 0 & \frac{3l^2}{EI_z} & 6 \\ 0 & 0 & -6 & 0 \\ -6l & 0 & 6 & 0 \end{pmatrix} \begin{pmatrix} Q(0) \\ \tan\alpha(l) \\ M(0) \\ w(l) \end{pmatrix} &= \frac{1}{12} \begin{pmatrix} 0 & \frac{l^3}{EI_z} & -6 & -\frac{3l^2}{EI_z} \\ 6l & 0 & 6 & 0 \\ 0 & -6l & 0 & -6 \\ 0 & 0 & 0 & 6 \end{pmatrix} \begin{pmatrix} \tan\alpha(0) \\ Q(l) \\ w(0) \\ M(l) \end{pmatrix} \\ &= \frac{1}{12} \begin{pmatrix} 0 & \frac{l^3}{EI_z} & -6 & -\frac{3l^2}{EI_z} \\ 6l & 0 & 6 & 0 \\ 0 & -6l & 0 & -6 \\ 0 & 0 & 0 & 6 \end{pmatrix} \begin{pmatrix} 0 \\ F \\ 0 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} \frac{Fl^3}{12EI_z} \\ 0 \\ -\frac{Fl}{2} \\ 0 \end{pmatrix} \end{aligned}$$

The solution found for this linear equation system is given by

$$\begin{pmatrix} Q(0) \\ \tan\alpha(l) \\ M(0) \\ w(l) \end{pmatrix} = \begin{pmatrix} F \\ -\frac{Fl^2}{2EI_z} \\ -Fl \\ \frac{2l^3F}{6EI_z} \end{pmatrix} \tag{9.1.6}$$

This example has shown in an easy way how the boundary element method works. For higher dimensional purposes the principle works similarly with a higher number of nodes and the fundamental solutions are much more complex. Consider for instance Lord Kelvin's fundamental solution of 2-D or 3-D cases, which are shown in section 4.9 in the next chapter.

9.2 Reformulation of the boundary integral equation

Recall the discretized boundary integral equation formulated in Section 3.8.

$$\sum_j \int_{\Gamma_j} u^*(x,\xi)\Phi_j^T \mathrm{d}\Gamma_j p^n = c(\xi)u(\xi) + \sum_j \int_{\Gamma_j} p^*(x,\xi)\Phi_j^T \mathrm{d}\Gamma_j u^n \tag{9.2.7}$$

The construction of the nodal vectors u^n and u^n as well as the construction of the interpolation matrix Φ_j will be explained here. First, a nodal dependent value vector for a boundary Γ_j is introduced, which is defined as

$$\breve{\boldsymbol{V}}_j = \left(v^{(1)}, v^{(2)}, ..., v^{(\kappa)}\right)^T \tag{9.2.8}$$

where $v^{(i)}$ denotes the nodal value or function v for local node i. However, each local node has a unique location on the global boundary where it is referenced as $n^{[k]}$. The boundary values denoted now as functions of η can be written as

$$u(\eta) = \varphi^{(1)}u^{(1)} + \varphi^{(2)}u^{(2)} + ... + \varphi^{(\kappa)}u^{(\kappa)} = \breve{\boldsymbol{\Phi}}_j^T \breve{\boldsymbol{U}}_j \tag{9.2.9}$$

$$p(\eta) = \varphi^{(1)}p^{(1)} + \varphi^{(2)}p^{(2)} + ... + \varphi^{(\kappa)}p^{(\kappa)} = \breve{\boldsymbol{\Phi}}_j^T \breve{\boldsymbol{P}}_j \tag{9.2.10}$$

$\varphi^{(i)}$ is the interpolation function (Section 3.11) for the local node i with the properties

$$\sum_i^{\kappa} \varphi^{(i)}(\eta) = 1 \;\; |\eta| \leq 1 \tag{9.2.11}$$

$$\varphi^{(i)}(\eta) = \begin{cases} 1 & \eta = \eta^{(i)} \\ 0 & \eta = \eta^{(j)} \wedge i \neq j \end{cases} \tag{9.2.12}$$

$\eta^{(i)}$ reflects the local coordinates η of the local node i corresponding to the boundary element. To apply the interpolation function, the integral equation is extended by the weight $\breve{\boldsymbol{\Phi}}_j$. Since $\breve{\boldsymbol{U}}_j$ is constant compared to the integral over Γ_j, it can be put outside of the integral. Thus, the boundary integral equation reads

$$\sum_j \int_{\Gamma_j} u^*(x,\xi)\breve{\boldsymbol{\Phi}}_j^T \mathrm{d}\Gamma_j \breve{\boldsymbol{P}}_j = c(\xi)u(\xi) + \sum_j \int_{\Gamma_j} p^*(x,\xi)\breve{\boldsymbol{\Phi}}_j^T \mathrm{d}\Gamma_j \breve{\boldsymbol{U}}_j \tag{9.2.13}$$

In case of linearity or higher orders nodes are shared by other boundary elements. Their influence values have to be added. Expanding equation (9.2.13) using equations (9.2.9) and (9.2.10) yields

$$\sum_j^E \sum_i^{\kappa} \int_{\Gamma_j} u^*(x,\xi)\varphi_j^{(i)^T} \mathrm{d}\Gamma_j p_j^{(i)} = c(\xi)u(\xi) + \sum_j^E \sum_i^{\kappa} \int_{\Gamma_j} p^*(x,\xi)\varphi_j^{(i)^T} \mathrm{d}\Gamma_j u_j^{(i)} \tag{9.2.14}$$

This is from the point of view of the elements, since the first sum spans over the entire elements. A rearrangement of the equation, to build the sum over the entire nodes, leads to

$$\sum_{k}^{N}\left(\sum_{j}^{E}\gamma_{jk}\int\limits_{\Gamma_j}u^*(x,\xi)\varphi_j^{(\alpha_{jk})^T}\mathrm{d}\Gamma_j\right)p^{[k]}=c(\xi)u(\xi)+\sum_{k}^{N}\left(\sum_{j}^{E}\gamma_{jk}\int\limits_{\Gamma_j}p^*(x,\xi)\varphi_j^{(\alpha_{jk})^T}\mathrm{d}\Gamma_j\right)u^{[k]} \tag{9.2.15}$$

γ_{jk} denotes the dependency of Node k and Element j with the following property

$$\gamma_{jk}=\begin{cases}1 & x^{[k]}\in \breve{\boldsymbol{X}}_j\\ 0 & x^{[k]}\notin \breve{\boldsymbol{X}}_j\end{cases} \tag{9.2.16}$$

α_{jk} can be seen as a function returning for a global node index k its local index corresponding to the boundary element j. From equation (9.2.15) it can be seen, that α_{jk} is only needed when $\gamma_{jk}=1$. Otherwise α_{jk} is not defined at all.

9.3 Collocation method

After reformulating the discretized boundary integral equation, the collocation method needs to be adapted. Remember equation (9.2.15), the discretized form of the boundary integral equation. The inner integrals relate the node k to the element j. Node l is the load point and its position, $x^{[l]}$, was previously denoted as ξ. These integrals will now be denoted as

$$\sum_{j}^{E}\gamma_{jk}\int\limits_{\Gamma_j}u^*(x^{[k]},x^{[l]})\varphi_j^{(\alpha_{jk})^T}\mathrm{d}\Gamma_j = G_{kl} \tag{9.3.17}$$

$$\sum_{j}^{E}\gamma_{jk}\int\limits_{\Gamma_j}p^*(x^{[k]},x^{[l]})\varphi_j^{(\alpha_{jk})^T}\mathrm{d}\Gamma_j = \hat{H}_{kl} \tag{9.3.18}$$

and the discretized boundary integral equation can be rewritten as

$$\sum_{k}^{N}G_{kl}p^{[k]}=c^{[l]}u^{[l]}+\sum_{k}^{N}\hat{H}_{kl}u^{[k]} \tag{9.3.19}$$

The rest of the collocation method follows the same steps as already explaind in Section 3.13.

Chapter 10

Wavelets

10.1 Introduction

Section 5.4.2 shows the usage of wavelets for a hierarchical multiresolution surface representation. However, wavelets can be applied for the boundary element values too which will be shown in this section. Since the boundary element values are represented on the discretized surface through nodes, a wavelet hierarchy for these nodes can be defined. Sub-connectivity of the surface is needed to create this wavelet hierarchy over the boundary element values. This subdivision connectivity can be achieved by surface subdivision methods explained in Chapter 5 or through a resampling of the mesh as presented in the paper of Eck et al. [1995]. The system of equations, introduced in Section 3.13, can be transformed without committing an error by

$$\boldsymbol{G}\boldsymbol{u} = \boldsymbol{H}\boldsymbol{p} \tag{10.1.1}$$

$$\underbrace{\boldsymbol{\Psi}\boldsymbol{G}\boldsymbol{\Psi}^{-1}}_{\boldsymbol{G}'}\underbrace{\boldsymbol{\Psi}\boldsymbol{u}}_{\boldsymbol{u}'} = \underbrace{\boldsymbol{\Psi}\boldsymbol{H}\boldsymbol{\Psi}^{-1}}_{\boldsymbol{H}'}\underbrace{\boldsymbol{\Psi}\boldsymbol{p}}_{\boldsymbol{p}'} \tag{10.1.2}$$

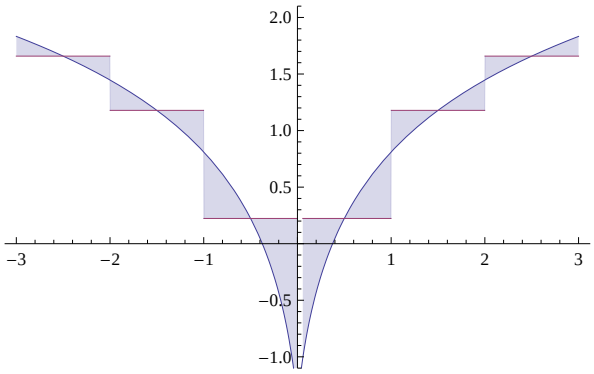
The mother wavelet function is similar to the interpolation function used for the boundary element method introduced in Section 3.11. After this transformation, low valued coefficient entries (representing wavelets) of the interaction matrices $\boldsymbol{G}'$ and $\boldsymbol{H}'$ between two nodes may be tilted by accepting the resulting error, which is expected to be very small. The different types of interpolation functions for the boundary element values on the surface should come along with the wavelet transformation. Figures 10.1 shows a logarithmic function represented by using a constant, linear and quadratic mother wavelet function. All those wavelets are using the same number of nodes. The nodes are located at the integers on the x-axis. It can be observed that the error becomes smaller if a higher degree of interpolation is chosen. For evaluation, an error

$$e = \int_{-3}^{3} (f(x) - w(x))^2 \mathrm{d}x \tag{10.1.3}$$

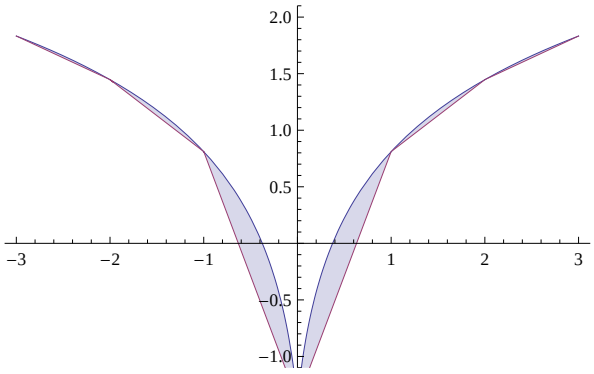
has been calculated. The subdivided surface meshes are defined as M^i where i denotes the level of subdivision depth. Wavelet representation requires that nodes included in mesh M^i are also included in M^{i+1} as shown in Figure 10.2. One can say

$$M^0 \subset M^1 \subset M^2 \subset \ldots$$

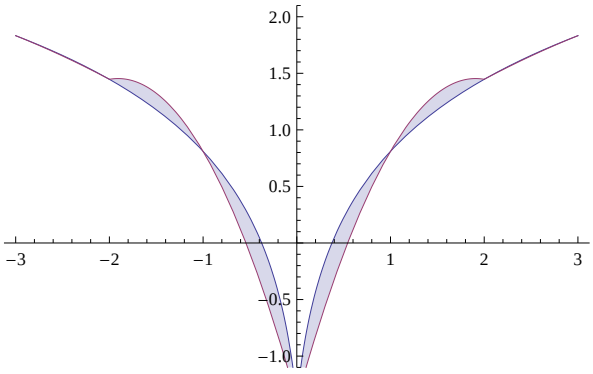
This is obviously valid for the Loop, Catmull-Clark, 1-to-4 and Butterfly subdivision algorithms. Doo-Sabin and $\sqrt{3}$-subdivision create faces which partly overlap a face in the parent level. Therefore, these methods are not well suited for a wavelet representation.



(a) Constant wavelets interpolation ($e = 0.771309$)



(b) Linear wavelets interpolation ($e = 0.339815$)



(c) Quadratic wavelets interpolation ($e = 0.0226218$)

Figure 10.1: Wavelets interpolations on a logarithmic function

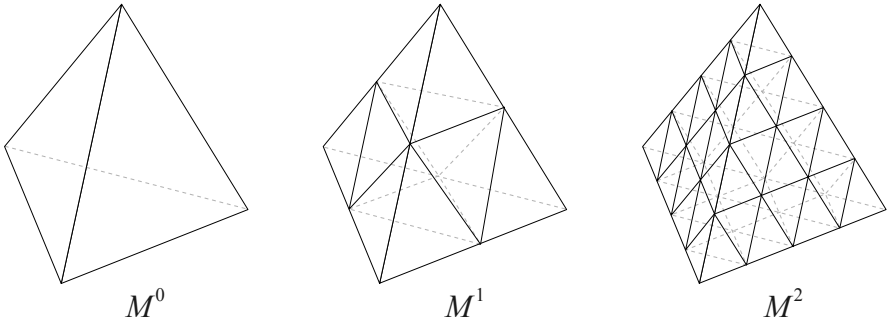


Figure 10.2: Subdivided meshes

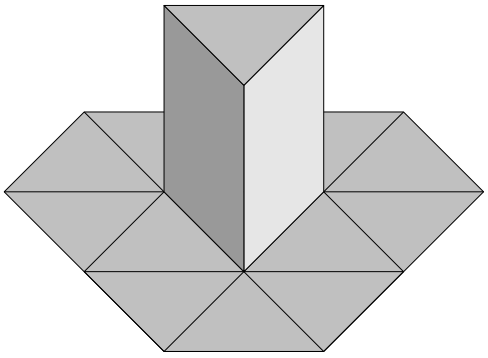


Figure 10.3: Constant 3-D wavelet

10.2 Wavelets for constant boundary elements

For constant boundary elements the boundary values are represented by nodes located in the center of each surface element (Figure 10.4). A constant wavelet in 3-D is shown in Figure 10.3. Constant interpolation defines a wavelet for a node as the difference of itself to another node. This is similar to the Haar wavelet. Figure 10.4 shows a triangle subdivided into four parts, which on their part are subdivided again and finally consists of 16 triangle patches. A constant interpolation without averaging is defined on M^1 as

$$v^{(1)} = x^{(1)} \tag{10.2.4}$$
$$w^{(2)} = x^{(2)} - x^{(1)} \tag{10.2.5}$$
$$w^{(3)} = x^{(3)} - x^{(1)} \tag{10.2.6}$$
$$w^{(4)} = x^{(4)} - x^{(1)} \tag{10.2.7}$$

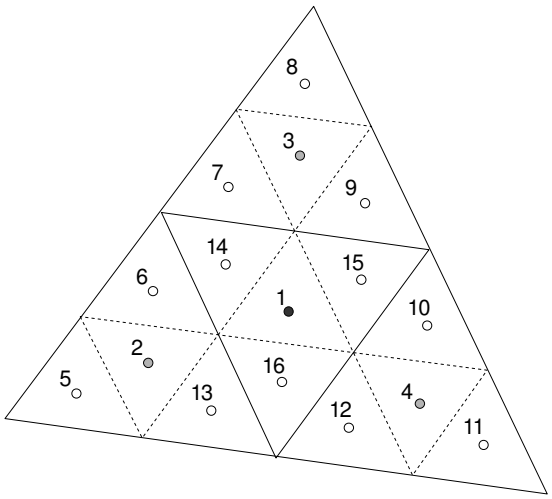


Figure 10.4: Wavelet nodes for a constant boundary element

It is to be noted, that $x^{(1)}$ already exists on M^0 and the wavelets for the child nodes 2-4 on M^1 are calculated. In a general matrix form $\boldsymbol{A}$ can be written as

$$\begin{pmatrix} v^{(a)} \\ w^{(b)} \\ w^{(c)} \\ w^{(d)} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x^{(a)} \\ x^{(b)} \\ x^{(c)} \\ x^{(d)} \end{pmatrix} \tag{10.2.8}$$

$$\begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{pmatrix} = \boldsymbol{A}\boldsymbol{x} \tag{10.2.9}$$

where $\boldsymbol{A}$ defines the wavelet transformation matrix for one surface patch. $\boldsymbol{A}$ can also be written as a block matrix as follows

$$\boldsymbol{A} = \left(\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{array}\right) \tag{10.2.10}$$

$$= \left(\begin{array}{c|c} \boldsymbol{A}_v & \boldsymbol{B}_v \\ \hline \boldsymbol{A}_w & \boldsymbol{B}_w \end{array}\right) \tag{10.2.11}$$

$$= \left(\begin{array}{c|c} \boldsymbol{I} & \boldsymbol{\emptyset} \\ \hline \boldsymbol{A}_w & \boldsymbol{I} \end{array}\right) \tag{10.2.12}$$

Furthermore, an inverse wavelet transformation would redo the transformation and recalculates the original node values. This can be achieved by the inverse of A

$$\boldsymbol{x} = \boldsymbol{A}^{-1} \begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{pmatrix} \tag{10.2.13}$$

$$= \left(\begin{array}{c|c} \boldsymbol{I} & \boldsymbol{\emptyset} \\ \hline -\boldsymbol{A}_w & \boldsymbol{I} \end{array}\right) \begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{pmatrix} \tag{10.2.14}$$

10.3 Improved wavelets for constant elements

The improvement consists in defining the wavelets of the interaction matrices of the nodes as the difference of the average of the child nodes resulting from the subdivision method, where the first child node in the children array stores this average interaction matrix. In Figure 10.4, node $n^{(1)}$ is designed to be the first child in this array. It stores the average value which can be obtained by

$$v^{(1)} = \frac{\sum_{i=1}^{4} x^{(i)}}{4} \tag{10.3.15}$$

The wavelets for the remaining nodes, $n^{(2)}$ until $n^{(4)}$, can be calculated by

$$w^{(i)} = x^{(i)} - v^{(1)} = x^{(i)} - \frac{\sum_{j=1}^{4} x^{(j)}}{4} \quad (i > 1) \tag{10.3.16}$$

This can also be written in matrix notation

$$\boldsymbol{w} = \boldsymbol{A}\boldsymbol{x} \tag{10.3.17}$$

$$\begin{pmatrix} v^{(1)} \\ w^{(2)} \\ w^{(3)} \\ w^{(4)} \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{pmatrix} \begin{pmatrix} x^{(1)} \\ x^{(2)} \\ x^{(3)} \\ x^{(4)} \end{pmatrix} \tag{10.3.18}$$

This follows exactly the Haar wavelet function. The reverse transformation, used to gain the original data again, is given by

$$x^{(1)} = v^{(1)} - \sum_{i=2}^{4} w^{(i)} \tag{10.3.19}$$

$$x^{(i)} = v^{(1)} + w^{(i)} \quad (i > 1) \tag{10.3.20}$$

in matrix notation

$$\boldsymbol{x} = \boldsymbol{A}^{-1} \begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{pmatrix} \tag{10.3.21}$$

$$\begin{pmatrix} x^{(1)} \\ x^{(2)} \\ x^{(3)} \\ x^{(4)} \end{pmatrix} = \begin{pmatrix} 1 & -1 & -1 & -1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v^{(1)} \\ w^{(2)} \\ w^{(3)} \\ w^{(4)} \end{pmatrix} \tag{10.3.22}$$

10.4 Wavelets for linear boundary elements

For linear boundary elements, the nodes are located at the corner of each surface element (Figure 10.6). A linear wavelet in 3-D is shown in Figure 10.5. In this case a wavelet representation for the interaction matrices of the nodes located on the edges of the parent surface element is defined as the difference to the mean value of the interaction matrices of those nodes forming the edge. In Figure 10.6, node $n^{(1)}$ and $n^{(2)}$ form an edge, and the difference of node $n^{(4)}$ lying on this edge, to the average value $(n^{(1)} + n^{(2)})/2$ defines the wavelet value for node $n^{(4)}$. Furthermore, as introduced in Section 3.11.2, a linear interpolation for a triangle is given by its barycentric coordinates as

$$\varphi(\lambda_1, \lambda_2, \lambda_3) = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} \left(x^{(1)}, x^{(2)}, x^{(3)} \right) \tag{10.4.23}$$

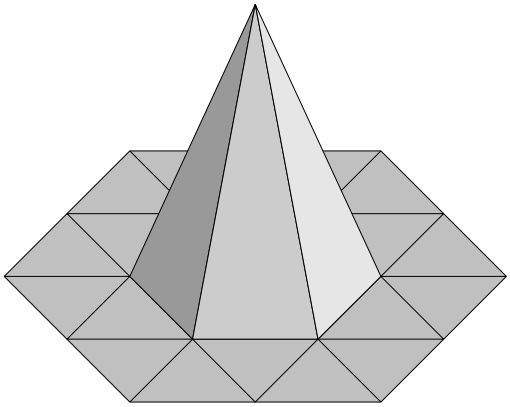


Figure 10.5: Linear 3-D wavelet

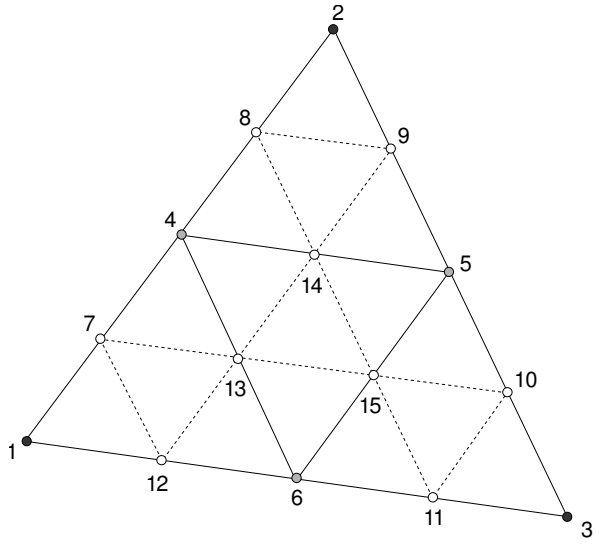


Figure 10.6: Wavelet nodes for a linear boundary element

In general the wavelets for a linear interpolation are defined as

$$v^{(1)} = x^{(1)} \tag{10.4.24}$$
$$v^{(2)} = x^{(2)} \tag{10.4.25}$$
$$v^{(3)} = x^{(3)} \tag{10.4.26}$$
$$w^{(4)} = x^{(5)} - \varphi(\lambda^{(4)}) = x^{(4)} - \frac{x^{(1)} + x^{(2)}}{2} \tag{10.4.27}$$
$$w^{(5)} = x^{(5)} - \varphi(\lambda^{(5)}) = x^{(4)} - \frac{x^{(2)} + x^{(3)}}{2} \tag{10.4.28}$$
$$w^{(6)} = x^{(6)} - \varphi(\lambda^{(6)}) = x^{(4)} - \frac{x^{(3)} + x^{(1)}}{2} \tag{10.4.29}$$

or in matrix notation

$$\begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{pmatrix} = \left(\begin{array}{c|c} \boldsymbol{I} & \boldsymbol{\emptyset} \\ \hline \boldsymbol{A}_w & \boldsymbol{I} \end{array}\right) \boldsymbol{x} \tag{10.4.30}$$
$$= \boldsymbol{A}\boldsymbol{x} \tag{10.4.31}$$

with

$$\boldsymbol{A}_w = \frac{1}{2}\begin{pmatrix} -1 & -1 & . \\ . & -1 & -1 \\ -1 & . & -1 \end{pmatrix} \tag{10.4.32}$$

The reverse transformation of the linear wavelets is defined as

$$x^{(1)} = v^{(1)} \tag{10.4.33}$$
$$x^{(2)} = v^{(2)} \tag{10.4.34}$$
$$x^{(3)} = v^{(3)} \tag{10.4.35}$$
$$x^{(4)} = w^{(4)} + \varphi(\lambda^{(4)}) = w^{(4)} + \frac{v^{(1)} + v^{(2)}}{2} \tag{10.4.36}$$
$$x^{(5)} = w^{(5)} + \varphi(\lambda^{(5)}) = w^{(5)} + \frac{v^{(2)} + v^{(2)}}{2} \tag{10.4.37}$$
$$x^{(6)} = w^{(6)} + \varphi(\lambda^{(6)}) = w^{(6)} + \frac{v^{(3)} + v^{(2)}}{2} \tag{10.4.38}$$

which can be expressed also in matrix form as

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{pmatrix} \tag{10.4.39}$$
$$= \left(\begin{array}{c|c} \boldsymbol{I} & \boldsymbol{\emptyset} \\ \hline -\boldsymbol{A}_w & \boldsymbol{I} \end{array}\right)\begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{pmatrix} \tag{10.4.40}$$

10.5 Wavelets for quadratic boundary elements

In this case the nodes are placed on several points on the surface element as shown in Figure 10.8. A quadratic wavelet in 3-D is shown in Figure 10.7. The wavelets determination works similarly to the linear interpolation, but using a quadratic function instead. Each triangle contains six nodes, 1-6, which stores the boundary values, displacement and traction, needed for the quadratic shape function.

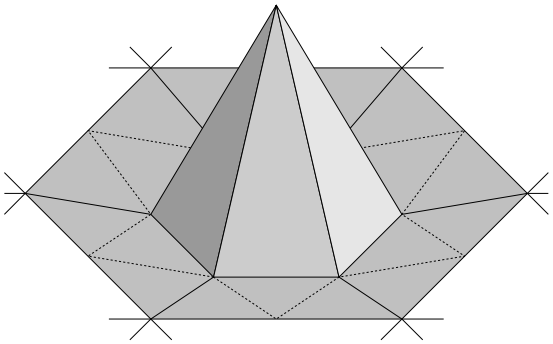


Figure 10.7: Quadratic 3-D wavelet

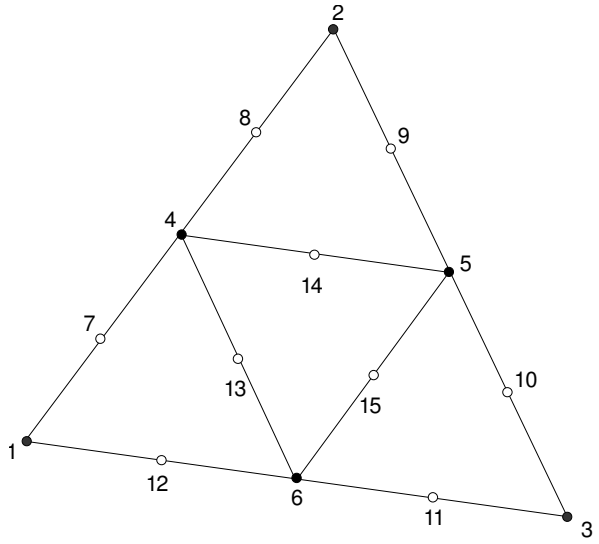


Figure 10.8: Wavelet nodes for a quadratic boundary element

Next a quadratic interpolation of the interaction matrices over a 2-D triangle boundary element is applied. A quadratic interpolation on a triangle is defined in its barycentric coordinates as

$$\varphi(\lambda_1,\lambda_2,\lambda_3) = \begin{pmatrix} (2\lambda_1-1)\lambda_1 \\ (2\lambda_2-1)\lambda_2 \\ (2\lambda_3-1)\lambda_3 \\ 4\lambda_1\lambda_2 \\ 4\lambda_2\lambda_3 \\ 4\lambda_3\lambda_1 \end{pmatrix} \left(x^{(1)},x^{(2)},x^{(3)},x^{(4)},x^{(5)},x^{(6)}\right) \tag{10.5.41}$$

An example for some arbitrary node values $x^{(i)}|1 \le i \le 6 = (0.2, 0.2, 1.1, 0.3, 0.5, 0.5)^{\mathrm{T}}$ is shown in Figure 10.9 Analogous to the linear wavelets it can be written

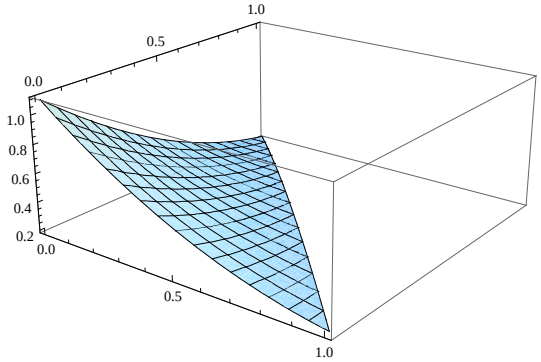


Figure 10.9: Quadratic interpolation of six values over a triangular surface

$$v^{(i)} = x^{(i)} \quad 1 \le i \le 6 \tag{10.5.42}$$
$$w^{(j)} = x^{(j)} - \varphi(\boldsymbol{\lambda}^{(j)}) \quad 7 \le j \le 15 \tag{10.5.43}$$

For completeness this is also shown in matrix notation

$$\begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{pmatrix} = \left(\begin{array}{c|c} \boldsymbol{I} & \boldsymbol{\emptyset} \\ \hline \boldsymbol{A}_w & \boldsymbol{I} \end{array}\right) \boldsymbol{x} \tag{10.5.44}$$
$$= \boldsymbol{A}\boldsymbol{x} \tag{10.5.45}$$

$$\boldsymbol{A}_w = \frac{1}{8}\begin{pmatrix} -1 & . & 3 & . & . & 7 \\ . & -1 & 3 & . & 7 & . \\ . & 3 & -1 & . & 7 & . \\ -1 & 3 & & 7 & . & . \\ 3 & -1 & . & 7 & . & . \\ 3 & . & -1 & . & . & 7 \\ -1 & -1 & . & 2 & 4 & 4 \\ -1 & . & -1 & 4 & 4 & 2 \\ . & -1 & -1 & 4 & 2 & 4 \end{pmatrix} \tag{10.5.46}$$

$\boldsymbol{A}_v$ is once again an identity matrix of dimension 6 and $\boldsymbol{B}_w$ an identity matrix of size 9.

10.6 Higher order boundary elements

For higher ordered boundary elements a higher ordered wavelet interpolation can be used. The algorithm works the same way as shown above. Interpolation functions for cubic interpolation are shown in Section 3.11.

10.7 Assembly of wavelet matrices

As already explained, there are wavelets of different types (constant, linear, etc.) to transform patches locally. This section will guide through a technique how those wavelets can be assembled into one large matrix for a wavelet transformation of the whole mesh. The wavelet matrices $(\boldsymbol{A}_w)^i_j$, introduced in the sections above, can be assembled at the i^{th} level of depth following

$$(\boldsymbol{A}_w)^i = \boldsymbol{D}(\boldsymbol{\alpha}) \sum_{j=1}^{s(M^i)} (\boldsymbol{A}_w)^i_j \tag{10.7.47}$$

where j denotes the j^{th} surface patch on mesh M^i and $s(M^i)$ the number of surface patches on Mesh M^i. The involved nodes and their indices must be taken into account. $\boldsymbol{D}(\boldsymbol{\alpha})$ is a diagonal matrix of size $n(M^i) - n(M^{i-1})$ with the scaling vector $\boldsymbol{\alpha}$ as the main diagonal, which depends on the type of interpolation used, since a node might be shared with other surface patches. α_k assigns $\frac{1}{2}$ if $n^{(k)}$ lies on an edge and 1 if $n^{(k)}$ lies inside the surface patch. $n(M^i)$ reflects the number of nodes counted in Mesh M^i. As an example, the linear wavelet transformation whose matrix $(A_w)^1_1$ on level 1, can be computed out of nodes 1, 4, 6, 7, 12 and 13 (Figure 10.10), is given by

$$\begin{pmatrix} w^{(7)} \\ \vdots \\ w^{(12)} \\ w^{(13)} \\ \vdots \\ w^{(16)} \end{pmatrix} = \begin{pmatrix} x^{(7)} \\ \vdots \\ x^{(12)} \\ x^{(13)} \\ \vdots \\ x^{(16)} \end{pmatrix} - \underbrace{\frac{1}{2} \begin{pmatrix} 1 & \cdots & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & \cdots & 1 \\ 1 & \cdots & 0 & \cdots & 1 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 \end{pmatrix}}_{(\boldsymbol{A}_w)^1_1} \begin{pmatrix} x^{(1)} \\ \vdots \\ x^{(4)} \\ \vdots \\ x^{(6)} \end{pmatrix} \tag{10.7.48}$$

Each other value will be zero. This results in a block matrix of type

$$\boldsymbol{A}^i = \left(\begin{array}{c|c} \boldsymbol{I} & \boldsymbol{\emptyset} \\ \hline (\boldsymbol{A}_w)^i & \boldsymbol{I} \end{array} \right) \tag{10.7.49}$$

The same holds for the inverse transformation matrix where

$$(\boldsymbol{P}_w)^i = -\sum_{j=0}^{s(M^i)} (\boldsymbol{A}_w)^i_j = -(\boldsymbol{A}_w)^i \tag{10.7.50}$$

$$\boldsymbol{P}^i = \left(\begin{array}{c|c} \boldsymbol{I} & \boldsymbol{\emptyset} \\ \hline (\boldsymbol{P}_w)^i & \boldsymbol{I} \end{array} \right) = \left(\begin{array}{c|c} \boldsymbol{I} & \boldsymbol{\emptyset} \\ \hline -(\boldsymbol{A}_w)^i & \boldsymbol{I} \end{array} \right) \tag{10.7.51}$$

In order to gain a total transformation over all level of depth, the matrices $\boldsymbol{A}^i$ and $\boldsymbol{P}^i$ can be combined to the newly introduced matrices $\boldsymbol{\Psi}$ and $\boldsymbol{\Psi}^{-1}$ where

$$\boldsymbol{\Psi} = \hat{\boldsymbol{A}}^0 \hat{\boldsymbol{A}}^1 \dots \hat{\boldsymbol{A}}^n \tag{10.7.52}$$

$$\boldsymbol{\Psi}^{-1} = \hat{\boldsymbol{P}}^n \hat{\boldsymbol{P}}^{n-1} \dots \hat{\boldsymbol{P}}^0 \tag{10.7.53}$$

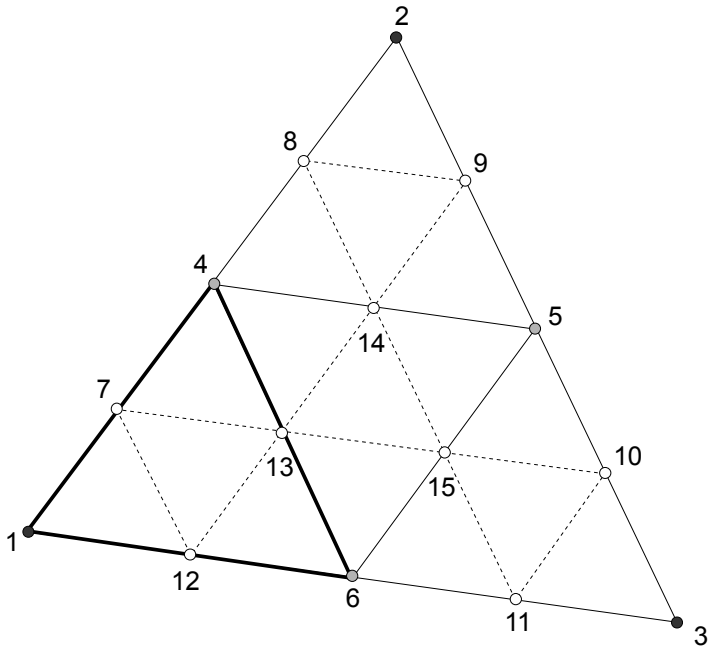


Figure 10.10: Matrix assembly example

Since matrices at different levels of depth have different sizes, the matrices at higher levels ($i \rightarrow 0$) must be stretched before they can be multiplied. This can be done by filling the missing parts as follows

$$\hat{\boldsymbol{A}}^i = \left(\begin{array}{c|c} \boldsymbol{A}^i & \boldsymbol{\emptyset} \\ \hline \boldsymbol{\emptyset} & \boldsymbol{I} \end{array}\right) \tag{10.7.54}$$

$$\hat{\boldsymbol{P}}^i = \left(\begin{array}{c|c} \boldsymbol{P}^i & \boldsymbol{\emptyset} \\ \hline \boldsymbol{\emptyset} & \boldsymbol{I} \end{array}\right) \tag{10.7.55}$$

Chapter 11

Application

The thesis' implementation was developed under Linux Debian, Suse and Ubuntu. For the preparation, a standard installation of Linux was set up with a Firewire interface assumed. This interface is required to act with the Phantom Omni device from SensAble. An installation of the application can be found in the Appendix D. This chapter will show some implementation details of the software and how different problems have been solved.

11.1 CPU specific implementations

11.1.1 Template based boundary element classes

The boundary elements differ on the one hand in their types of surfaces, triangular, quadrilateral and so on, and on the other hand in their types of interpolation of the fundamental solution for the integration. So it is a good attempt to implement the boundary element class as a template class having surface and node classes as template parameters. These template parameter classes have defined interface functions returning necessary data about them. For instance, the template parameter class may have a function called *area* returning the size of the surface element. The implementation of the boundary element template class is done as follows:

```
1 template <class Surface, class Node>
2 class BoundaryElement : public Surface {}
```

Here the class *Surface* contains information about the surface and the class *Node* contains data about the nodes differing at the interpolation type. This can be easily seen when comparing nodes for the constant element case in contrast to the linear element case. In the constant interpolation case each node has one predecessor and four successors while in the linear interpolation case each node has two predecessors and n, the valence of the vertex, successors. More details about the number of nodes per element and their relation to the elements is explained in-depth in Chapter 10. The great benefit of templates is that needed dependencies and relations are checked at compile-time and no further if-conditions are needed resulting in a more efficient machine code and in a better readable source code. Concrete implementations for different surfaces and interpolation types are done in the derived classes.

```
1 template <class Surface>
2 class ConstantBoundaryElement : public BoundaryElement<Triangle,
      TriangleNode>
```

and

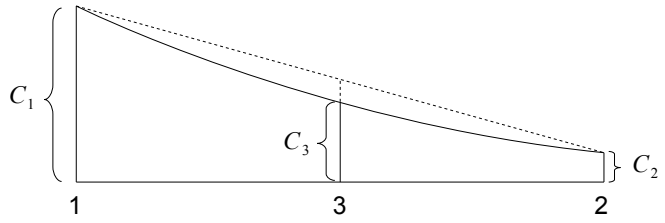


Figure 11.1: Arising error due to linear interpolation

```
1 template <class Surface>
2 class LinearBoundaryElement : public BoundaryElement<Triangle,
    VertexNode>
```

11.1.2 Accuracy dependent integration over boundary elements

First, it is recommended to read Chapter 3 explaining the boundary element method for a better understanding. Calculations of matrix coefficients are computationally expensive, since an integration of a complex function over each triangle is required. The function is strictly increasing from the source point towards the load point. In other words, the farther the source point is away from the load point the more constant becomes the function to integrate. This can be used to save computation time since the integration of a nearly constant function over an area can be approximated very well by the size of the area times an averaged function value for this area.

$$\int_A f(x)\mathrm{d}x \approx \mathrm{avg}(f(x))A \tag{11.1.1}$$

Including the idea of wavelets to represent the boundary element values will not affect the final result. As already mentioned in section 10.1 the equation system

$$\boldsymbol{H}p = \boldsymbol{G}u \tag{11.1.2}$$

will be transformed using the wavelets transformation matrix $\boldsymbol{\Psi}$ to

$$\boldsymbol{\Psi}\boldsymbol{H}\boldsymbol{\Psi}^{-1}\boldsymbol{\Psi}p = \boldsymbol{\Psi}\boldsymbol{G}\boldsymbol{\Psi}^{-1}\boldsymbol{\Psi}u \tag{11.1.3}$$

However, since a swapping of the matrix columns of $\boldsymbol{H}$ and $\boldsymbol{G}$ is needed, depending on the boundary conditions, the reverse transformation (multiplication with $\boldsymbol{\Psi}^{-1}$), will be ignored. The finally resulting equation system is of the type

$$\boldsymbol{\Psi}\boldsymbol{H}p = \boldsymbol{\Psi}\boldsymbol{G}u \tag{11.1.4}$$

The arising error using lazy wavelets (Section 10.4) for the linear case is computed as

$$error = \frac{1}{2}(C_1 + C_2) - C_3 \tag{11.1.5}$$

This is shown in Figure 11.1. If this error is small enough, C_3 will be set to the average value of C_1 and C_2 which is exactly the same like setting the wavelet value for C_3 to zero. The application marks this node to be computed if the error is not small enough.

11.1.3 Subdivided mesh storage in a hierarchical representation

First of all, the mesh of the deformable object consists of triangles only. These triangles are being subdivided to generate a denser mesh having more boundary nodes. In the case of a linear interpolation (Section 3.11), these nodes are the vertices of the triangles. The triangles of the original mesh are the roots of a quad-tree structure where each triangle knows its parent, children and neighbors. Additionally, the original mesh is collected in an n-tree structure with a pseudo root entry. Figure 11.2 shows the structure for a twice subdivided cube consisting of twelve triangular surface elements in its original mesh. The elements inside the tree are the boundary elements storing more than only the

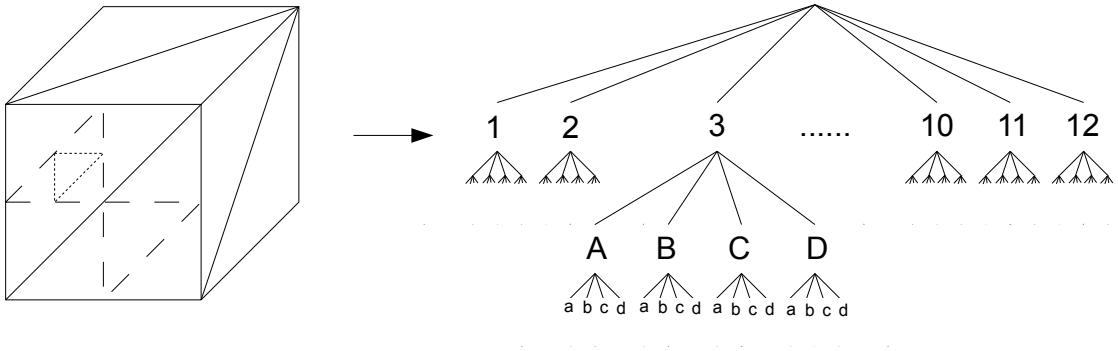


Figure 11.2: Subdivision hierarchy of a cube

geometric definition of the triangle. They also store the local positions of the nodes inside the surface patch, which are needed for the boundary integration. Functions called on the root of this tree will be applied on each element and might change attributes of the elements depending on the function called.

11.1.4 Wavelet transformation matrix construction

The wavelet transformation matrices are constructed in a recursive way, bottom-up for the forward and reverse transformation matrix. The following pseudo code explains the algorithm to construct linear wavelets.

```
1  function constructWavelets(node) {
2    if node is not part of the base mesh {
3      p1 = left parent of node
4      p2 = right parent of node
5      if p1 is not constructed
6        constructWavelets(p1)
7      if p2 is not constructed
8        constructWavelets(p2)
9      for all nodes {
10       matrix_reverse[node, nodes] += matrix_reverse[p1, nodes]/2
11       matrix_reverse[node, nodes] += matrix_reverse[p2, nodes]/2
12     }
```

```
13        matrix_forward[node, p1] = -0.5;
14        matrix_forward[node, p2] = -0.5;
15      }
16      matrix_forward[node, node] = 1.0;
17      matrix_reverse[node, node] = 1.0;
18  }
```

This algorithm computes the reverse transformation entries at the leafs nodes and propagates the wavelet values up to the root nodes.

11.2 GPU specific implementations (CUDA)

First, a short overview of CUDA will be given. CUDA is a general-purpose parallel programming API and, as already mentioned in the Chapter 2, nowadays GPUs contain up to 512 multi-cores, consisting of CUDA cores or threads collected in several blocks. Each core has access to several memories (Figure 11.3),

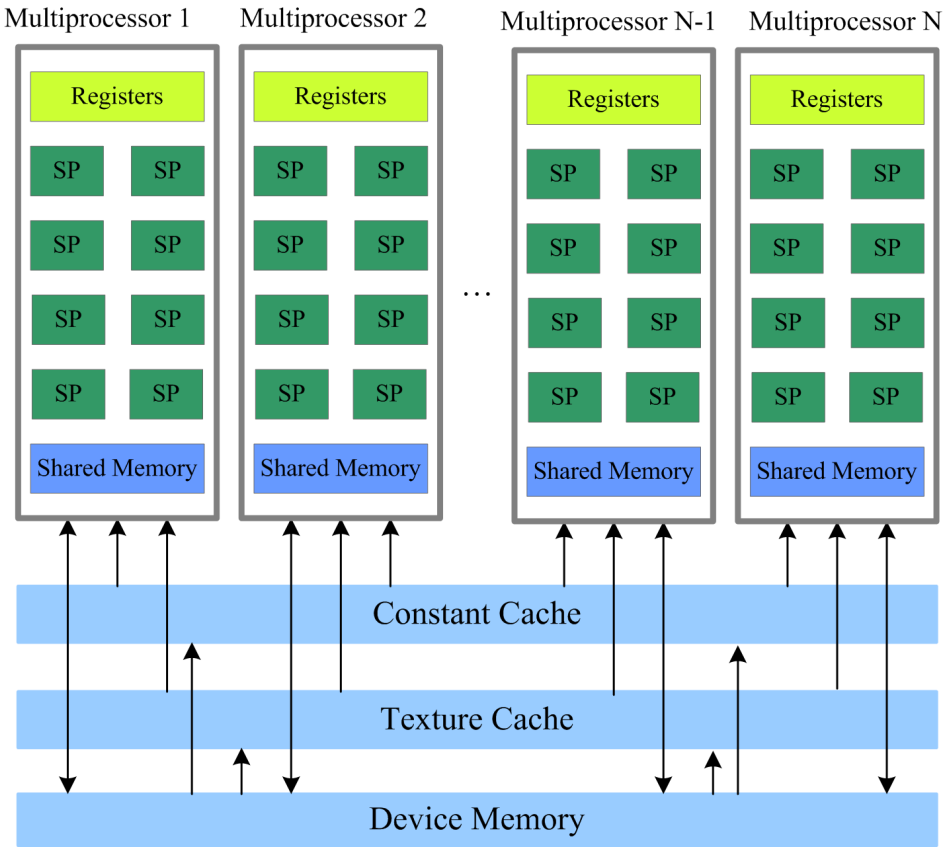


Figure 11.3: CUDA memory accesses [Liu et al., 2009]

- a local memory (only accessible for itself)
- a shared memory (accessible for all cores collected in the same block)

- a global memory (accessible for all cores)
- a constant memory (only for reading, filled by the CPU)
- a texture memory

Each CUDA code is executed on each processor and invoked by a kernel, as shown in Figure 11.4,

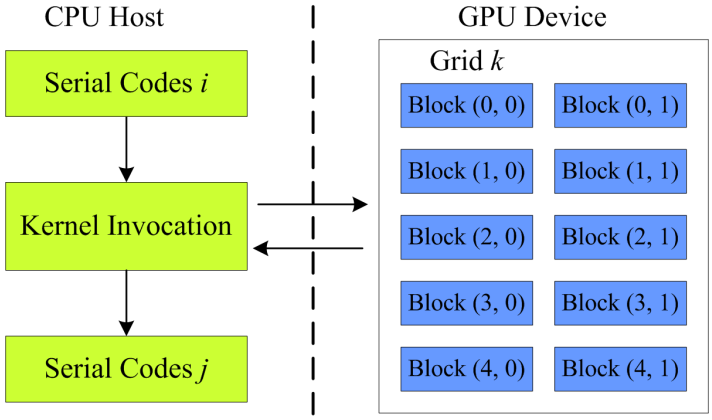


Figure 11.4: CUDA code invokation by a kernel [Liu et al., 2009]

```
1 kernel<<<grid dimension, block dimension>>>(parameter list);
```

A kernel is a function being called from the CPU and executed on the GPU. More details about kernel, host and device functions are explained the CUDA programming guide [nVidia, 2008]. Developing programs in CUDA has its benefits and drawbacks:

+ *C++*-like programmable parallel processors
+ Synchronized within a thread block (single instruction multiple data)
+ Fast access to shared memory for each thread inside its block
+ High portability, runs on different operation systems

– Code optimizations are more difficult
– runs only with nVidia hardware

To ensure parallel global memory accesses, it is inalienable to have neighboring threads accessing neighboring memory spaces. This is called coalescing, and is an important term in context of CUDA programming. Often it is necessary to rearrange the memory data to be processed to fulfill this condition. The CUDA Profiler [nVidia, 2008] is an useful tool to check if coalescing has been achieved or not. For debugging purposes the application can also be run in an CPU emulation mode, which allows value dumps, but is a multiple slower than executed on the device.

11.2.1 Sparse matrix-vector multiplication

The implementation of a multiplication of a sparse matrix and a vector (SpMV) is explained here. The sparse matrix is stored in CRS format, being explained in Section 6.3.3. A multiplication with a vector, $y = \boldsymbol{A}x$, will be calculated by the following source code

```
1  int row = blockIdx.x * blockDim.x + threadIdx.x;
2  if (row<N) {
3    int row_start_index = row_start_indexes[row];
4    int row_end_index = row_start_indexes[row+1];
5    for(int idx_ptr=row_start_index; idx_ptr<row_end_index; idx_ptr
       ++) {
6      int column = column_indexes[idx_ptr];
7      y[row] += A[idx_ptr]*x[column];
8  }
9  }
```

Each thread of the GPU, indexed by the thread index row, multiplies one row of the matrix by the vector x and stores the result in $y[row]$. The if condition ensures, that no invalid row indices are accessed. However, the listing above does not reflect coalesced memory access, since the data of the matrix array A are accessed by the counter index idx_ptr which differs from the thread index. The listing below is improved and uses coalescing.

```
1  int row = blockIdx.x * blockDim.x + threadIdx.x;
2  if (row < N) {
3    int row_start_index = row_start_indexes[row];
4    int row_end_index = row_start_indexes[row+1];
5    int offset = 0;
6    for(int idx_ptr=row_start_index; idx_ptr<row_end_index; idx_ptr
       ++) {
7      int column = column_indexes[idx_ptr];
8      int rows = row_counter[idx_ptr-row_start_index];
9      float x_col = tex1Dfetch(x_as_tex, column);
10     y[row] += A[row+offset]*x_col;
11     offset+=rows;
12   }
```

For coalescing, it can be seen now, the matrix is accessed by the row index pointer, which is the same as the thread index pointer and the matrix has been rearranged. Further, since the entire values of x are read in an arbitrary order it is useful to bind the vector x to a texture for a faster read-only access which has been implemented as well. $x_a s_t ex$ represents the texture where the vector x is bound to. For the boundary element method in 3-D the values of the vectors x and y are 3-D vectors and the coefficient entries $\boldsymbol{A}[row + offset]$ are 3×3 matrices.

11.2.2 BiCGStab solver

Most of the computing time lies in solving the equation system, which is implemented in CUDA to use the parallel program capability of the GPU. In context of real-time, the duration of the iterative solver is the bottleneck. The application computes by turns 20 iterative steps of the solver and visualizes the solution. The iterative BiCGStab solver implementation in CUDA itself is a little bit tricky since synchronizations over all blocks and threads are needed. A SpMV completely must be computed

first before continuing, for instance, building an inner product of the result. CUDA 2.x supports threads synchronizations but does not support blocks synchronizations at all. This might be a feature of CUDA 3.0. To force synchronization, the solver, controlled by the CPU, calls stepwise CUDA kernels invoking a computation on the GPU. When a kernel terminates, it is ensured, that all threads in all blocks are finished. A succeeding implementation follows

```
1   kernel::bicgstab_step_1<<<GRID_SIZE, BLOCK_SIZE>>>(m_nodelist,
        m_size, m_values);
2   for (int i=0; i<20; i++) {
3     if (i==0)
4       kernel::bicgstab_step_2a<<<GRID_SIZE, BLOCK_SIZE>>>(
            m_nodelist, m_size, m_values);
5     else
6       kernel::bicgstab_step_2b<<<GRID_SIZE, BLOCK_SIZE>>>(
            m_nodelist, m_size, m_values);
7     kernel::bicgstab_step_3<<<GRID_SIZE, BLOCK_SIZE>>>(m_nodelist,
          m_size, m_values);
8     kernel::bicgstab_step_4<<<GRID_SIZE, BLOCK_SIZE>>>(m_nodelist,
          m_size, m_values, i);
9   }
10  kernel::bicgstab_step_5<<<GRID_SIZE, BLOCK_SIZE>>>(m_nodelist,
        m_size, m_values);
```

m_nodelist contains the matrix and vectors needed by this solver. Each part needing a synchronization afterwards is collected in an own kernel. A terminating kernel waits until all threads of all blocks are terminated and so a synchronization over the blocks is achieved.

11.2.3 Inner vector products

Parallelization of the inner product can be done by striding the vector and calculating the inner products for each part. Afterwards, these partial results are summed up using a parallel reduction as shown in Figure 11.5. This is repeated until only two partial results remain and finally summed up by one processor, while the other ones must wait. However, this is the most efficient solution for computing inner products on a parallel processor machine. The listing below shows how this can be implemented in CUDA.

```
1  __device__ float sumInBlock[GRID_SIZE];
2  __shared__ float sumOverBlock[BLOCK_SIZE];
3  __device__ unsigned int counter=0;
4  __shared__ bool isLastBlockDone;
5
6  __device__ void dot(float* x, float* y, float* result, int N) {
7    int idx = blockIdx.x * blockDim.x + threadIdx.x;
8    if (idx<N)
9      sumOverBlock[threadIdx.x] = x[idx]*y[idx];
10   else
11     sumOverBlock[threadIdx.x] = 0;
12   for(int i = BLOCK_SIZE; i /= 2;) {
13     __syncthreads();
14     if(threadIdx.x < i)
```

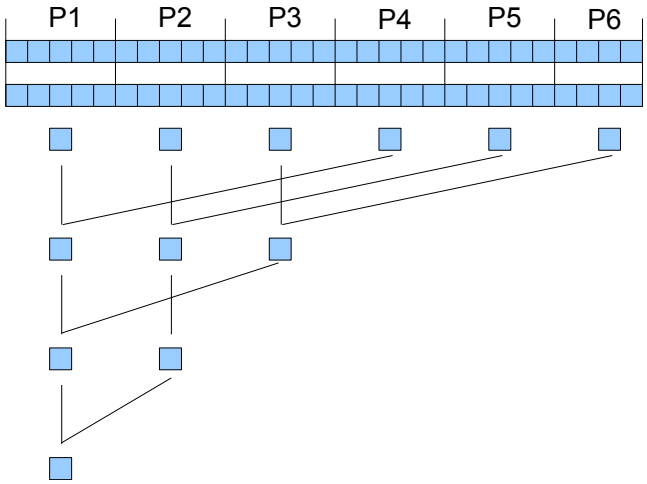


Figure 11.5: Parallelized inner product computation for one block

```
15          sumOverBlock[threadIdx.x] += sumOverBlock[i + threadIdx.x];
16      }
17      __syncthreads();
18      if (threadIdx.x==0) {
19        sumInBlock[blockIdx.x] = sumOverBlock[0];
20        __threadfence();
21          unsigned int value = atomicInc(counter, gridDim.x);
22        isLastBlockDone = (value == (gridDim.x - 1));
23      }
24      __syncthreads();
25      if (isLastBlockDone) {
26        float totalSum;
27        if (threadIdx.x == 0) {
28          totalSum = 0;
29          for(int i=0; i<gridDim.x; i++)
30            totalSum += sumInBlock[i];
31        }
32        if (threadIdx.x == 0) {
33          result[0] = totalSum;
34          counters = 0;
35        }
36      }
37    }
```

Since the processors on the graphics card are collected in blocks, the sums of each block must be added to compute the final inner vector product. In line 9 each thread multiplies two corresponding vector entries, which are summed up in the current block to a partial result using parallel reduction (line 12–16). The following nested if-block (line 18–23) checks which block finishes last and let it sum up the partial results in the last nested if-block (line 25–36). The parallel reduction is done in shared memory. Accesses to the shared memory are a multiple faster than to the global memory. However, for a valid result the number of threads must be at least equal to the length of the vectors

for the inner product.

11.2.4 OpenGL buffer registration for CUDA

As the computation of the deformation already happens on the GPU, it suggests itself to copy the data directly from the CUDA global memory to the OpenGL render memory instead of copying the results back to the CPU memory and then via OpenGL commands back to the graphics card. The data transfer rate from CUDA memory to OpenGL memory via the GPU bus is a multiple faster than from CUDA to the CPU memory. To handle this, a OpenGL vertex buffer object must be created and registered from CUDA. Therefore, it is necessary to know how much memory must be allocated to store the vertex data and their surface normals. However, once this buffer is registered, the OpenGL buffer data can be accessed like normal CUDA memory and the results can be copied. Before and after changing the OpenGL data by CUDA, commands are needed to synchronize read and write command with the rendering pipeline. More details about registration of buffer objects by CUDA can be found in the CUDA programming guide [nVidia, 2008].

Chapter 12

Results

12.1 Wavelet approach

In this section the efficiency of the wavelet approach for the boundary element method will be evaluated. In Figure 12.1 a normal computed dense coefficient matrix is shown as it results form the boundary element method. Black colored content stands for non-zero matrix entries while white content stands for close-to-zero entries. The wavelet approach transforms these matrix data entries into wavelet coefficients whose numerical values are less then the original matrix data entries. The original matrix is multiplied with the wavelet transformation matrix which is depicted for linear wavelets in Figure 12.2. The resulting wavelet transformed matrix is shown in Figure 12.3.

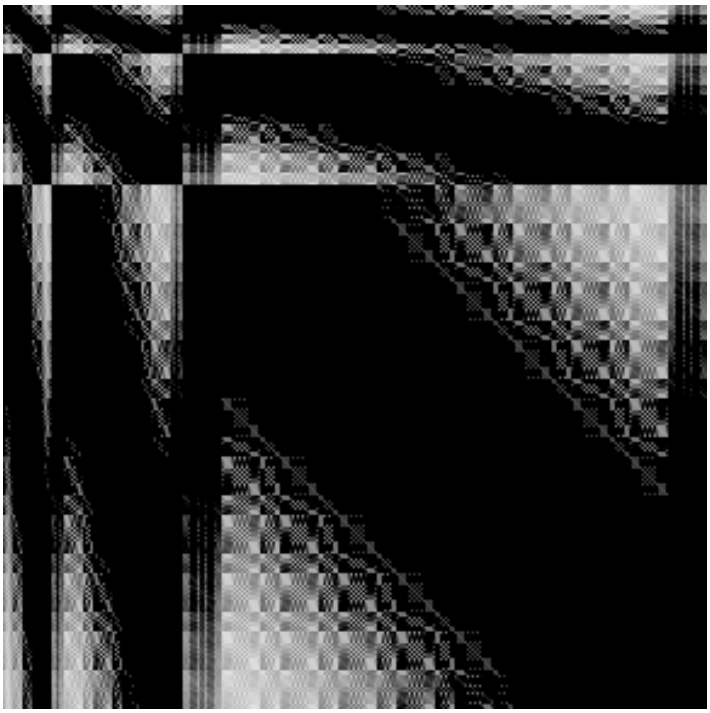


Figure 12.1: Original dense coefficient matrix

Figure 12.2: Wavelet transformation matrix using linear wavelets

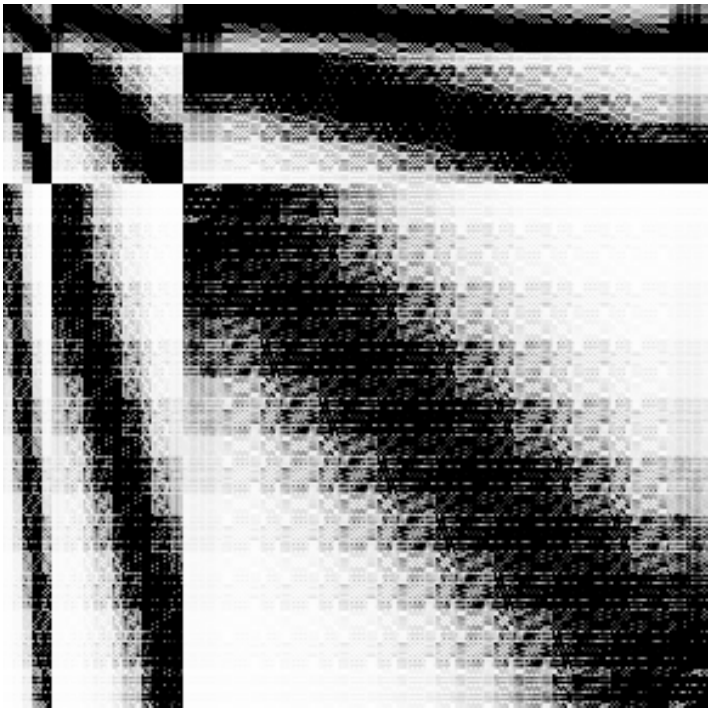


Figure 12.3: Wavelet transformed coefficient matrix

12.2 Test machines

The application has been tested on three different test platforms whose technical specifications are listed in Table 12.1.

9800 GT	GPU	NVIDIA GeForce 9800 GT OC+
	GPU Memory	512Mb GDDR3 @ 900MHz (256-bit)
	CUDA cores	112 @ 1500MHz
	CPU	Intel(R) Core(TM)2 Duo E7300 @ 2.66GHz
	CPU Memory	2x2GB DDR2-800
	OS	Ubuntu 9.04 (Linux 2.6.28-11 x86 64)
	CUDA	CUDA 2.2
	Host Compiler	GCC 4.3.3
280 GTX	GPU	NVIDIA GeForce 280 GTX
	GPU Memory	1GB GDDR3 @ 1107MHz (512-bit)
	CUDA cores	240 @ 1296MHz
	CPU	Intel(R) Core(TM)2 Duo E6400 @ 2.13GHz
	CPU Memory	2x2GB DDR2-800
	OS	Suse 11.1 (Linux 2.6.27-37 x86 64)
	CUDA	CUDA 2.3
	Host Compiler	GCC 4.3.4
130M GT	GPU	NVIDIA GeForce 130M GT
	GPU Memory	1Gb VDDR2 @ 500MHz (128-bit)
	CUDA cores	32 @ 1500MHz
	CPU	Intel(R) Core(TM)2 Quad Q9000 @ 2.00GHz
	CPU Memory	xx
	OS	Ubuntu 8.10 (Linux 2.6.26-19 x86 64)
	CUDA	CUDA 2.3
	Host Compiler	GCC 4.3.3

Table 12.1: Test platform specifications

12.3 Duration for the iterative solver

An important measure in context of real-time is the duration for the calculation of the next frame. Most of the time is needed to solve the equation system by the iterative BiCGStab solver. A great attention was payed to the parallel computation capabilities of the GPU. In Table 12.2 durations for computing one iteration step of the equation system solver by the CPU and GPU are compared using the fully populated matrix resulting from the boundary element method. It can be observed, that the GPU implementation of the iterative solver is up to 380 times faster than the CPU solution. The matrix size is three times higher than the number of nodes, since the application computes deformations of 3-D models using three coordinates (x, y, z) for each node.

depth	#nodes	matrix size	CPU	GPU	Speedup
3	66	198	4.8 ms	0.18 ms	∼26
4	258	774	62.5 ms	0.3 ms	∼208
5	1026	3078	947 ms	2.5 ms	∼378

Table 12.2: CPU versus GPU duration (9800 GT)

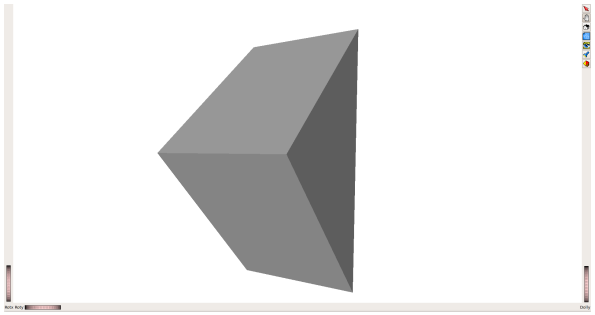


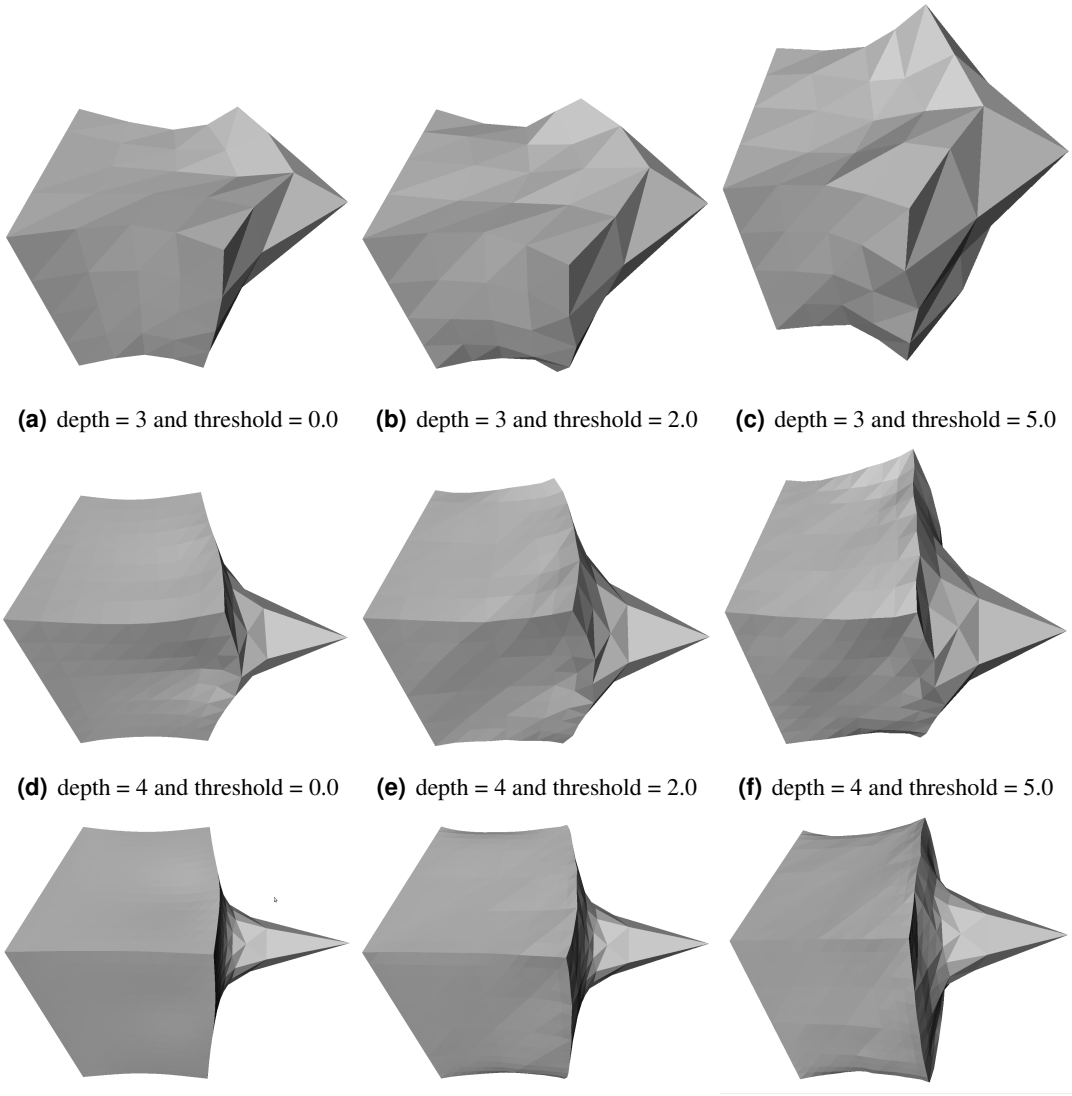
Figure 12.4: Undeformed prism

12.4 Testmodels

12.4.1 Prism

As a test object a prism (Figure 12.4) was deformed using the error thresholds 0.0, 2.0 and 5.0. The results are compared with respect to the population density of the matrix using different error thresholds. Figure 12.5 shows a prism deformed at several level of depth and error thresholds. The boundary condition of the left face is set to a displacement of zero, while one node on the opposite face is moved outwards. The remaining nodes are set to zero traction and their displacement will be computed. The solver aborts when the residual vector reaches a length of 10^{-6}. It can be seen that the deformation slightly varies between the different error thresholds. These errors are taken into account to receive more sparse matrices resulting in a faster computation. The exact deformation, where the error threshold is zero, and the approximated deformation at an error threshold of 2.0 are nearly the same. An error threshold of 5.0 creates a more unnatural deformation since it seems that the prism's volume increases. Figure 12.6 shows color-coded images of the deformed prism at a subdivision level of 5. The color represents the Hausdorff distance between two correlating vertices of the accurate and approximated solutions, while red stands for the largest distance and blue for no distance [Aspert et al., 2002]. The prism has an initial width of 1. How sparse the matrices become is shown in Figure 12.7. Black colored content represents the non-zero entries. The matrix is fully populated if the error threshold is zero. Tables 12.3 and 12.4 shows some interesting results for the deformed prism. These results are:

- the time to determine which nodes need be taken into account,
- the time needed for the integration of the boundary,
- the duration to compute one step of the BiCG iterative solver on different test platforms,
- the average number of iterations needed,

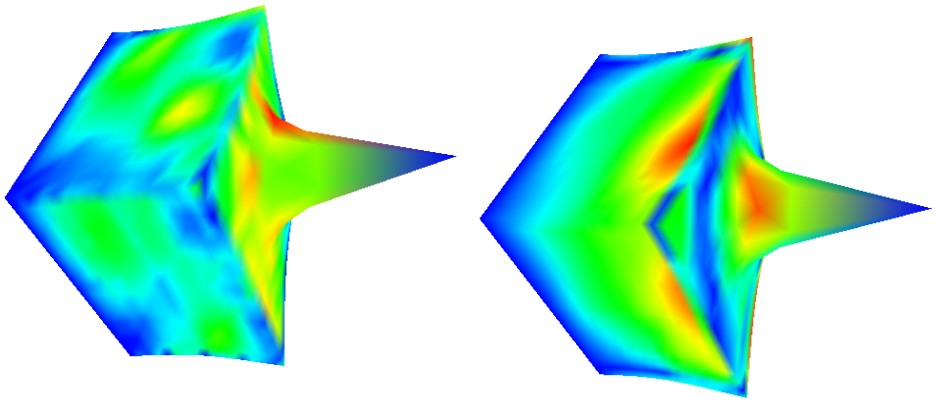


(a) depth = 3 and threshold = 0.0 **(b)** depth = 3 and threshold = 2.0 **(c)** depth = 3 and threshold = 5.0

(d) depth = 4 and threshold = 0.0 **(e)** depth = 4 and threshold = 2.0 **(f)** depth = 4 and threshold = 5.0

(g) depth = 5 and threshold = 0.0 **(h)** depth = 5 and threshold = 2.0 **(i)** depth = 5 and threshold = 5.0

Figure 12.5: Deformed prism



(a) threshold = 2.0, max. distance = 0.210 **(b)** threshold = 5.0, max. distance = 0.111

Figure 12.6: Color coded Hausdorff distances

- the number of non-zero entries as well as
- the amount of memory allocated on the GPU.

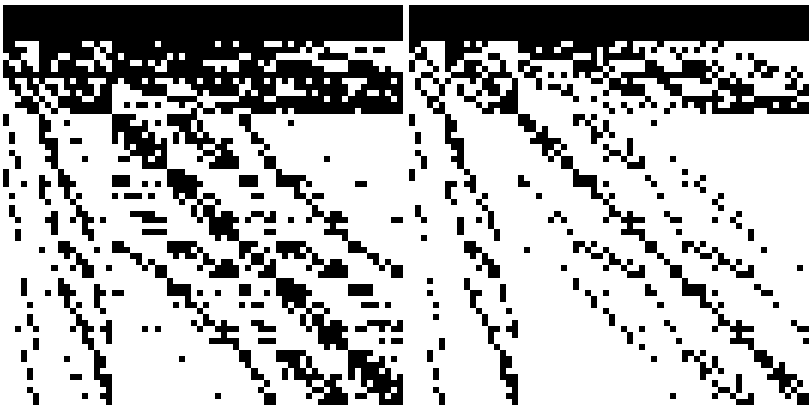
The number of available threads depends on the block and grid size of the CUDA implementation, which can be computed by

$$available\ threads = block\ size \cdot grid\ size$$

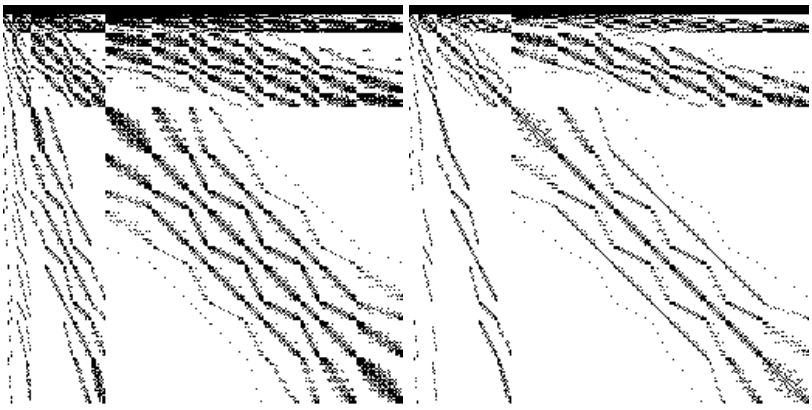
The table of results shows also the number of threads required for a successful sparse matrix vector multiplication (SpMV). For a successful vector copy operation or inner vector product the required number of available threads depends on the size of the matrix, since each thread handles one vector entry. All test cases use the same block size containing 192 threads. The used grid sizes for copy operation (copy) and for SpMV (mult) are listed in the tables as well. It can be seen for the prism at a subdivision depth of 5, having 1026 nodes, that over 90% of the memory can be saved and the duration of the solver decreases also at higher error thresholds due to the more sparse matrix. Table 12.4 shows the results of the deformation of the prism, where the error thresholds are set to 0.5, 1.0 and 2.0 since the computing time will be too high for an error threshold of 0.0. For a subdivision level of 6, the amount of data allocated when accepting no error will be about 1.2Gb, lying outside of the computation capabilities of the test platforms. Further, the time consumed to integrate over the boundary will be also a multiple higher and will take about 1.5 hours. The GeForce 280 GTX takes for one iteration step about 6ms to handle a data amount of 237Mb, resulting still in a real-time computation. At a subdivision level of 7, the algorithm contains a huge coefficient matrix (about 50000×50000) and results will become incomputable since the memory needed to be allocated exceeds the memory capabilities of the test platforms. Further, about 97% of the data are ignored running into a strong approximation.

12.4.2 Tetrahedron

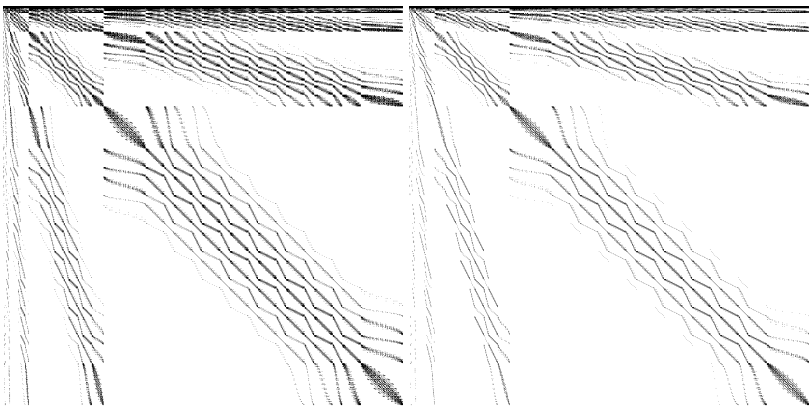
As a second test object a tetrahedron (Figure 12.8) was chosen, since a tetrahedron is the simplest three dimensional object consisting of four faces only and contains an elementary base mesh for sub-connectivity. Figure 12.9 depicts a deformation of a tetrahedron at several subdivision levels and error



(a) depth = 3 and threshold = 2.0 **(b)** depth = 3 and threshold = 5.0



(c) depth = 4 and threshold = 2.0 **(d)** depth = 4 and threshold = 5.0



(e) depth = 5 and threshold = 2.0 **(f)** depth = 5 and threshold = 5.0

Figure 12.7: Coefficient matrices of the prism

			error threshold		
depth	#nodes	result	0.0	2.0	5.0
3	66 (198)	Grid size (copy/mult)	2/6	2/4	2/2
		Mark nodes	27ms	76ms	86ms
		Boundary integration	1100ms	450ms	320ms
		Solver duration (9800 GT)	0.3ms	0.2ms	0.2ms
		Solver duration (280 GTX)	0.2ms	0.2ms	0.2ms
		Solver duration (130M GT)	0.4ms	0.3ms	0.3ms
		avg. #iterations	8	8	10
		#threads for SpMV	1056	648	376
		% of non-zeros	100%	37.5%	25.3%
		Allocated GPU memory	410kb	260kb	160kb
4	258 (774)	Grid size (copy/mult)	5/22	5/8	5/5
		Mark nodes	550ms	1400ms	1500ms
		Boundary integration	18s	3.5s	2.2s
		Solver duration (9800 GT)	1.4ms	0.5ms	0.3ms
		Solver duration (280 GTX)	0.3ms	0.25ms	0.2ms
		Solver duration (130M GT)	3.2ms	0.8ms	0.7ms
		avg. #iterations	13	22	17
		#threads for SpMV	4128	1428	812
		% of non-zeros	100%	18.2%	10.9%
		Allocated GPU memory	5.2Mb	1.8Mb	1.0Mb
5	1026 (3078)	Grid size (copy/mult)	17/86	17/15	17/9
		Mark nodes	12s	25s	26s
		Boundary integration	5m16s	26s	16s
		Solver duration (9800 GT)	17ms	2.2ms	1.2ms
		Solver duration (280 GTX)	2.5ms	0.7ms	0.6ms
		Solver duration (130M GT)	51ms	4.6ms	3.1ms
		avg. #iterations	55	84	81
		#threads for SpMV	16416	2847	1580
		% of non-zeros	100%	7.88%	4.53%
		Allocated GPU memory	76Mb	13.6Mb	7.7Mb

Table 12.3: Numerical results of a prism deformation (Part I)

			error threshold		
depth	#nodes	result	0.5	1.0	2.0
6	4098 (12294)	Grid size (copy/mult)	65/67	65/44	65/31
		Mark nodes	6m53s	6m54s	6m59s
		Boundary integration	7m25s	4m52s	3m22s
		Solver duration (9800 GT)	23ms	20ms	15ms
		Solver duration (280 GTX)	6.3ms	4.7ms	4.4ms
		Solver duration (130M GT)	74ms	50ms	32ms
		#threads for SpMV	12710	8265	5919
		% of non-zeros	7.7%	5.0%	3.2%
		Allocated GPU memory	237Mb	154Mb	111Mb
7	16386 (49159)	Mark nodes	1h52m	1h52m	
		Boundary integration	51min	38min	
		Solver duration (9800 GT)	failed	failed	
		Solver duration (280 GTX)	failed	failed	
		Solver duration (130M GT)	failed	failed	
		% of non-zeros	3.1%	2.0%	
		Allocated GPU memory	1760Mb	1450Mb	

Table 12.4: Numerical results of a prism deformation (Part II)

thresholds. The boundary displacements are set to zero for one face, while on an opposite face one node is displaced outwards. For the remaining nodes the displacement is computed having a traction of zero. Numerical results of the deformation can be found in Table 12.5.

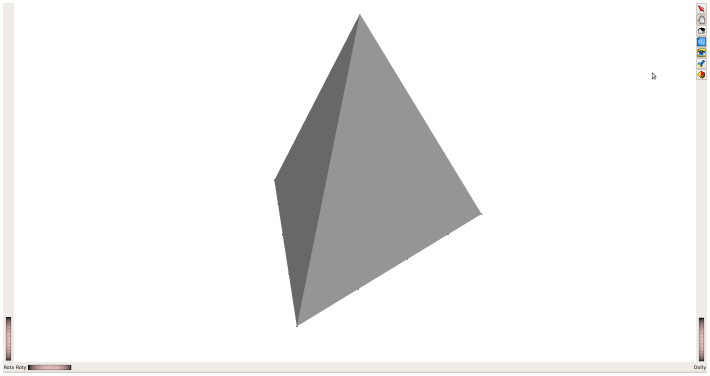
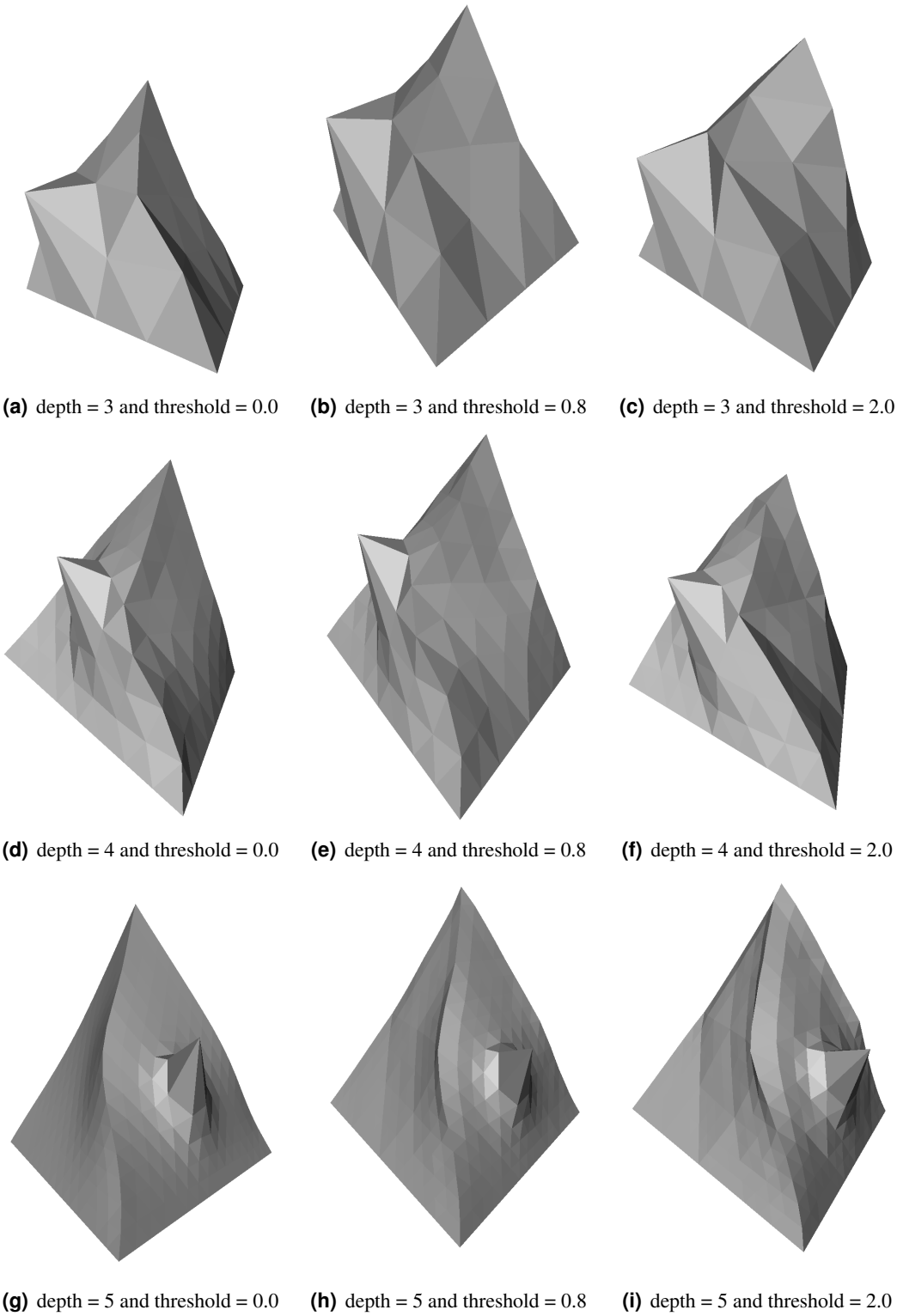


Figure 12.8: Undeformed tetrahedron

12.4.3 Rod

As a more illustrative object for a deformation, a rod will be deformed. The original mesh of the rod consists of 20 vertices forming 36 triangles and will be subdivided several times to gain a denser mesh for a more accurate solution of the deformation. In Figure 12.10 a deformation of the rod at several error thresholds and subdivision levels is shown. The solver aborts when a residual vector length of



(a) depth = 3 and threshold = 0.0 **(b)** depth = 3 and threshold = 0.8 **(c)** depth = 3 and threshold = 2.0

(d) depth = 4 and threshold = 0.0 **(e)** depth = 4 and threshold = 0.8 **(f)** depth = 4 and threshold = 2.0

(g) depth = 5 and threshold = 0.0 **(h)** depth = 5 and threshold = 0.8 **(i)** depth = 5 and threshold = 2.0

Figure 12.9: Deformed tetrahedron

			error threshold			
depth	#nodes	result	0.0	0.8	2.0	5.0
3	34 (102)	Grid size (copy/mult)	1/3	1/2	1/2	1/1
		Mark nodes	6.8ms	15ms	19.2ms	21.8ms
		Boundary integration	277ms	195ms	128ms	80ms
		Solver duration (9800 GT)	0.2ms	0.2ms	0.2ms	0.2ms
		Solver duration (280 GTX)	0.2ms	0.2ms	0.2ms	0.2ms
		Solver duration (130M GT)	0.2ms	0.2ms	0.2ms	0.2ms
		avg. #iterations	7	7	8	14
		#threads for SpMV	544	352	208	136
		% of non-zeros	100%	63%	41.3%	24.7%
		Allocated GPU memory	140kb	90kb	55kb	40kb
4	130 (390)	Grid size (copy/mult)	3/11	3/5	3/4	3/3
		Mark nodes	136ms	322	364ms	383ms
		Boundary integration	4560ms	1570ms	1060ms	680ms
		Solver duration (9800 GT)	0.4ms	0.3ms	0.3ms	0.3ms
		Solver duration (280 GTX)	0.25ms	0.2ms	0.2ms	
		Solver duration (130M GT)	1.3ms	0.5ms	0.4ms	0.3ms
		avg. #iterations	18	14	20	38
		#threads for SpMV	2080	896	708	352
		% of non-zeros	100%	32.4%	21.2%	13.3%
		Allocated GPU memory	1.4Mb	630kb	500kb	270kb
5	514 (1542)	Grid size (copy/mult)	9/43	9/11	9/5	9/5
		Mark nodes	2.8s	6.1s	6.2s	6.3s
		Boundary integration	1m16s	12s	6.4s	4.7ms
		Solver duration (9800 GT)	4.5ms	0.9ms	0.6ms	0.6ms
		Solver duration (280 GTX)	1ms	0.4ms	0.4ms	0.4ms
		Solver duration (130M GT)	13ms	2.1ms	1.3ms	1.1ms
		avg. #iterations	19	33	34	46
		#threads for SpMV	8224	1955	928	831
		% of non-zeros	100%	14%	7.9%	5.5%
		Allocated GPU memory	19.9Mb	4.8Mb	2.3Mb	2.1Mb
6	2050 (6150)	Grid size (copy/mult)	33/171	33/19	33/14	33/13
		Mark nodes	1m10s	1m41s	1m41s	1m44s
		Boundary integration	21m47s	1m24s	51s	33s
		Solver duration (9800 GT)	failed	5.1ms	2.9ms	2.3ms
		Solver duration (280 GTX)	11ms	1.6ms	1.3ms	1.1ms
		Solver duration (130M GT)	216ms	15ms	7.7ms	5.5ms
		avg. #iterations	93	88	58	113
		#threads for SpMV	32800	3491	2640	2367
		% of non-zeros	100%	6.0%	3.4%	1.8%
		Allocated GPU memory	307Mb	33Mb	25Mb	22Mb

Table 12.5: Numerical results of a tetrahedron deformation

			error threshold				
depth	#nodes	result	0.0	0.4	1.0	2.0	5.0
3	290 (870)	Grid size (copy/mult)	5/25	5/14	5/7	5/6	5/4
		Mark nodes	590ms	1.7s	1.6s	1.8s	1.9s
		Boundary integration	24s	7.4s	4.7s	3.6s	2.8s
		Solver duration (9800 GT)	1.5ms	0.8ms	0.5ms	0.5ms	0.5ms
		avg. #iterations	29	29	30	28	27
		#threads for SpMV	4640	2604	1176	965	644
		% of non-zeros	100%	30%	19%	14.9%	11.2%
		Allocated GPU memory	6.5Mb	3.6Mb	1.7Mb	1.4Mb	950kb
4	1154 (3462)	Grid size (copy/mult)	19/97	19/23	19/13	19/10	19/8
		Mark nodes	12s	30s	31s	32s	32s
		Boundary integration	6m36s	55s	35s	24s	18s
		Solver duration (9800 GT)	17ms	3.5ms	2.2ms	1.9ms	1.7ms
		avg. #iterations	82	98	82	82	82
		#threads for SpMV	18464	4327	2466	1829	1508
		% of non-zeros	100%	13%	8.0%	5.5%	3.8%
		Allocated GPU memory	98Mb	23Mb	13Mb	9.9Mb	8.2Mb

Table 12.6: Numerical results of a rod deformation

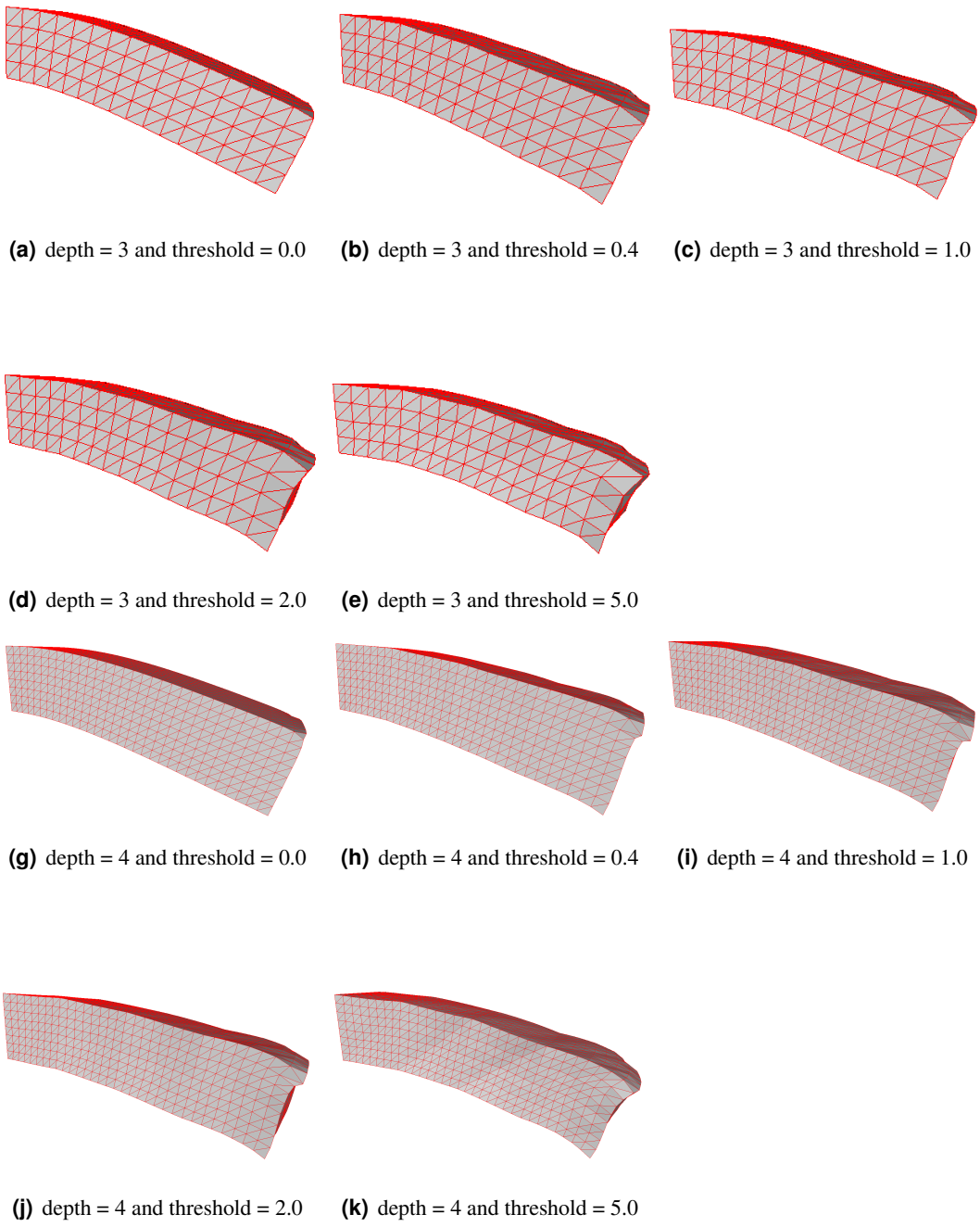
10^{-6} is reached. The rod is fixed in its left ending and is pulled down at the right ending at the upper two vertices. If the error threshold is too high, the iterative solver may not converge anymore or leads to unnatural results as shown in Figure 12.11. Figure 12.12 depicts again the Hausdorff distance of two corresponding vertices of the accurate and approximated solutions. Maximum distance means the maximum vertex displacement from the accurate solution. The rod has an initial length of 4. Table 12.6 shows the corresponding numerical results.

12.5 Memory allocations

Since the data is stored in a sparse matrix format, it is complicated to calculate the amount of memory needed to be allocated by the GPU. However, as an upper boundary, it is possible to calculate the size of the coefficient matrices whose entries are computed by the CPU in an offline prestep. The size of the matrices depends on the number of nodes used for the boundary element method. If a triangulated 3-D model consisting of f_0 triangular faces and v_0 vertices is assumed, the number of faces and vertices after j subdivision steps can be computed by

$$\begin{aligned} f_j &= f_0 4^j \\ v_j &= v_0 + \frac{3}{2} f_0 \sum_{i=1}^{j} 4^i \\ &= v_0 + 2f_0(4^j - 1) \end{aligned}$$

These equations assume, that the number of faces increases by a factor of 4 per step. For high n the number of vertices increases by a factor of 2 compared to the number of faces. The number of nodes



(a) depth = 3 and threshold = 0.0 **(b)** depth = 3 and threshold = 0.4 **(c)** depth = 3 and threshold = 1.0

(d) depth = 3 and threshold = 2.0 **(e)** depth = 3 and threshold = 5.0

(g) depth = 4 and threshold = 0.0 **(h)** depth = 4 and threshold = 0.4 **(i)** depth = 4 and threshold = 1.0

(j) depth = 4 and threshold = 2.0 **(k)** depth = 4 and threshold = 5.0

Figure 12.10: Deformed rod

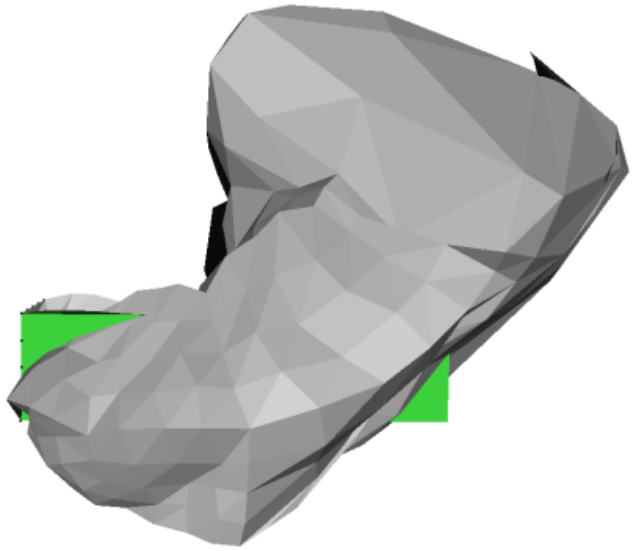
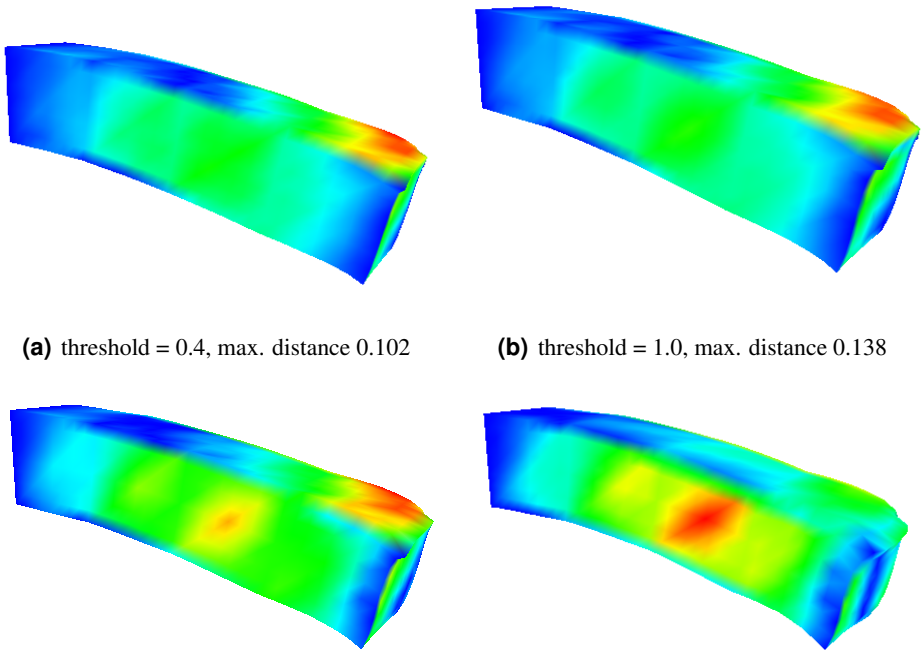


Figure 12.11: Unreal solution due to a too high threshold



(a) threshold = 0.4, max. distance 0.102

(b) threshold = 1.0, max. distance 0.138

(c) threshold = 2.0, max. distance 0.131

(d) threshold = 5.0, max. distance 0.194

Figure 12.12: Color coded Hausdorff distances of a deformed rod

n varies by the used interpolation function and can be calculated by

$$\begin{array}{ll} \text{constant} & n_j = f_j \\ \text{linear} & n_j = v_j \\ \text{quadratic} & n_j = v_j + \frac{3}{2} f_j \\ \text{cubic} & n_j = v_j + 4 f_j \end{array}$$

The dimension of the coefficient matrices H and G is $3n_j \times 3n_j$ after j subdivision steps.

12.5.1 Prism

The number of non-zeros inside the coefficient matrix is the most important measure to describe the efficiency of the achievement using the boundary element method with wavelets. Diagram 12.13 shows the percentage value of non-zeros over the error threshold for the previously deformed prism.

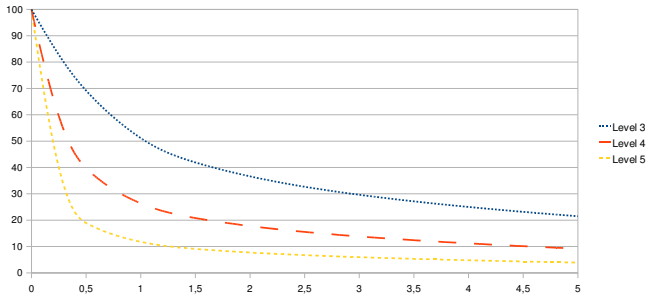


Figure 12.13: Percentage values of non-zeros over the threshold (Prism)

12.5.2 Rod

Diagram 12.14 shows the percentage value of non-zeros inside the coefficient matrices over the threshold of the error being accepted for the previously deformed rod.

12.6 Convergence behavior

Diagram 12.15 shows the convergence behavior of the BiCGStab iterative solver using a diagonal preconditioner for different levels of subdivision and error thresholds. The algorithm terminates if the residual norm is lower than 10^{-6}. It can be observed that the residual norm does not decrease for each iteration, but these peaks are typically in a solving process for non-symmetric matrices. However, it can be said that the solver converges anyway, while the number of iterations increases for larger matrices, resulting from the denser mesh through subdivision.

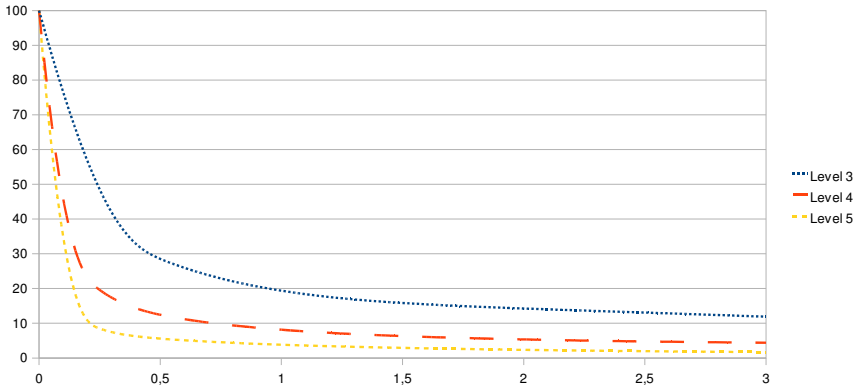
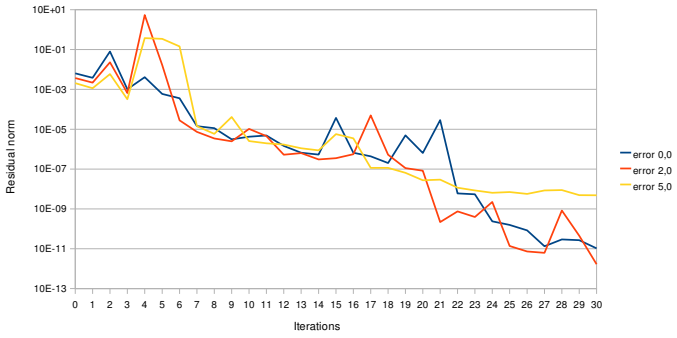
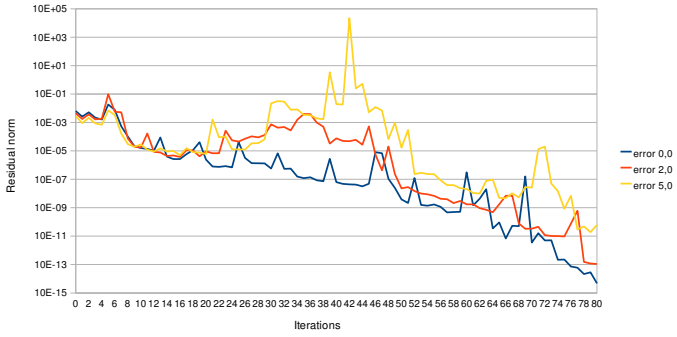


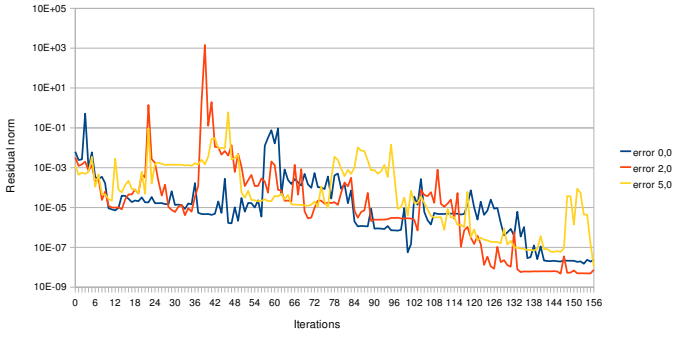
Figure 12.14: Percentage values of non-zeros over the threshold (Rod)



(a) depth=3 with 66 nodes



(b) depth=4 with 258 nodes



(c) depth=5 with 1026 nodes

Figure 12.15: Convergence behavior of the iterative solver (Prism)

Part IV

Closing

Chapter 13

Conclusion and future work

13.1 Conclusion

In this thesis it has been shown how 3-D models can be deformed by the boundary element method and how computation can be saved by linear wavelets. The iterative solver was implemented on the graphics card using CUDA giving a great speedup compared to a CPU solution. The boundary element method computes accurate deformations of 3-D models, while the computing time is quite high. However, the formulation of the fundamental solutions in elastostatics next to its evaluation is very complex in 3-D and only a numerical integration of these fundamental solutions over all surface patches is possible. For integration, the Gauss quadrature rule was extended and adapted for 2-D surface triangles. Nodes defined on the surface are used to set displacements or tractions and boundary conditions. The computed integration data is collected in nodes-to-nodes interaction matrices. Different interpolations of these fundamental solutions take more or less nodes, placed on the boundary element, into account and give more accurate or less accurate results. Higher interpolation types give better results, however, at the expense of the computing time. Inclusion of wavelets to achieve a reduction of this computing time has succeeded as well while a sub-connectivity representation of the model is needed. Models consisting of few patches with subdivision connectivity are more suited for the wavelet based boundary element method, since the simpler the base mesh of the model is all the more computation time can be saved by the wavelets technique. Hair wavelets are required for the constant interpolation type while lazy wavelets must be used for the linear interpolation type. A high percentage of the computing time and of the memory needed to store the nodes-to-nodes interaction values in the matrices can be saved. A good and fast result for the deformation of an arbitrary model can be achieved by a recalculation of the model surface elements to gain a model with high sub-connectivity. Body deformations can be computed in real-time, if the models geometry is not too complex and the number of nodes can be reduced by the wavelets approach. The equation system resulting from the collocation method is solved iteratively by the BiCG solver. The inclusion of a haptic force feedback device gives a better understanding of the model's deformation since the application also generates feedback forces which can be felt by the user and deformation becomes more intuitive. Haptic feedback devices have gained a widespread use in virtual and augmented reality, since they increase the information transport from the system to the user. The boundary element method is only valid for very small deformations and results computed for large deformations may look unreal. Applications using the boundary element method for virtual deformations, or similar neighbor topics – heat transfers and fluid flows –, are widely used and can be found, for instance, in medicine and engineering.

13.2 Future work

This thesis covers a solution for deformable objects using constant and linear wavelets for a transformation into the wavelet domain. One topic in future work may be a quadratic (k-disks) wavelet transformation producing more accurate results with increasing computing time. In this case, the boundary element values of the nodes are needed to be evaluated and calculated by quadratic interpolation functions. This works similar as already shown for the linear case. However, the resulting solution of the nodes needs to be visualized differently. In the quadratic case, a triangle of the boundary element mesh contains six nodes which can be visualized by four triangles. In the cubic case nine triangles can be drawn to visualize the boundary element displacements. Another topic in future work is the integration of other mesh grid types, like quadrilateral and polygonal surface elements. It can also be envisioned that the mesh consists of different and mixed surface element types. For quadrilateral elements, their interpolation functions have been shown in this thesis. The wavelet transformations will have to be adapted also for these newly incorporated surface element types. To ensure high-level sub-connectivity of arbitrary 3-D models, the surface points and structure needs to be recomputed, which was not covered by this thesis and is another topic in future work. Different types of models, for instance, models containing holes, have to be considered. Last but not least, as a proof of concept, an integration into the augmented reality framework Studierstube using OpenTracker can be assumed [Gervautz et al., 1999]. OpenTracker already supports several tracking modules for different input devices, to control the position of the pointer, as for example a tracking module for the Phantom Omni haptic force feedback device from SensAble.

Appendix A

Cartesian tensor notation

The Cartesian tensor notation uses subscripts (1, 2, 3) to represent the coordinates (x, y, z). It also represents summations, if a subscript occurs twice in a term, or derivation, if a subscript is separated by a comma. The following shows these rules by choosing examples from of this thesis.

A.1 Tensor notation rules

Inner vector product

$$a_i b_i = a_1 b_1 + a_2 b_2 + a_3 b_3$$

Trace of a matrix

$$a_{kk} = a_{11} + a_{22} + a_{33}$$

Matrix vector product

$$p_i = \sigma_{ij} n_j = \sigma_{i1} n_1 + \sigma_{i2} n_2 + \sigma_{i3} n_3$$

Fourth ordered tensor second ordered tensor product

$$\sigma_{ij} = C_{ijkl} \varepsilon_{kl} = \sum_{k=1}^{3} \sum_{l=1}^{3} C_{ijkl} \varepsilon_{kl}$$

Partial derivations

$$r_{,i} = \frac{\partial r}{\partial x_i} = \frac{r_i}{||r||}$$

$$\varepsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}) = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$$

$$\sigma_{ij,j} + b_i = 0 = \sum_{j=1}^{3} \frac{\partial \sigma_{ij}}{\partial x_j} + b_i = \begin{pmatrix} \frac{\partial \sigma_{11}}{\partial x_1} + \frac{\partial \sigma_{12}}{\partial x_2} + \frac{\partial \sigma_{13}}{\partial x_3} \\ \frac{\partial \sigma_{21}}{\partial x_1} + \frac{\partial \sigma_{22}}{\partial x_2} + \frac{\partial \sigma_{23}}{\partial x_3} \\ \frac{\partial \sigma_{31}}{\partial x_1} + \frac{\partial \sigma_{32}}{\partial x_2} + \frac{\partial \sigma_{33}}{\partial x_3} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

A.2 Kronecker delta

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

Appendix B

Dirac impulse and Heaviside function

B.1 Dirac impulse

$\delta(x)$ is the so-called *Dirac impulse* and satisfies

$$\int_{-\infty}^{\infty} \delta(x)h(x)\mathrm{d}x = h(0). \tag{2.1.1}$$

The function can be constructed by

$$\delta(x) = \lim_{n\to\infty} w_n(x) \tag{2.1.2}$$

with

$$w_n(x) = \begin{cases} \frac{1}{2n} & |x| < n \\ 0 & |x| > 0 \end{cases} \tag{2.1.3}$$

Function $w_n(x)$ is plotted in Figure B.1 with $n = \{1/4, 1/2, 1, 2\}$. As a result of this $\delta(0)$ results ∞

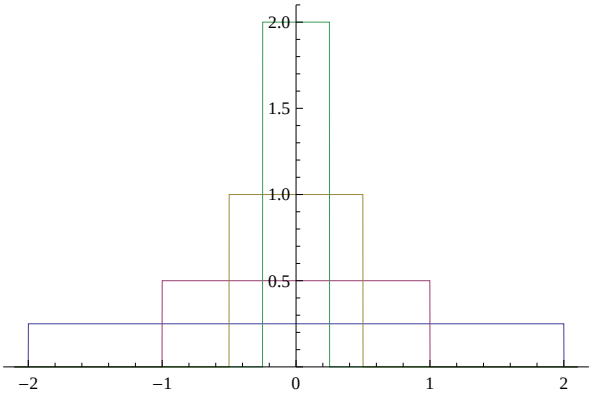


Figure B.1: $\delta(x)$ approximation by $w_n(x)$ with $n \leftarrow \infty$

and

$$\int_{-\infty}^{\infty} \delta(x)\mathrm{d}x = 1 = \int_{-\infty}^{\infty} w_n(x)\mathrm{d}x \tag{2.1.4}$$

This can be written as an indefinite integral

$$\int \delta(x)\mathrm{d}x = H(x) \tag{2.1.5}$$

using the Heaviside function explained in the next section. Furthermore for higher dimensions the Dirac impulse is defined as

$$\delta(x_1, x_2, ..., x_n) = \prod_{i=1}^{n} \delta(x_i) \tag{2.1.6}$$

For a closed domain Ω with a defined function $u(x)$ over Ω equation (2.1.7) is valid if $\xi \in \Omega$.

$$\int_{\Omega} \delta(x - \xi)u(x)\mathrm{d}\Omega = u(\xi) \tag{2.1.7}$$

B.2 Heaviside function

$H(x)$ is commonly known as the *Heaviside function* with the property

$$H(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases} \tag{2.2.1}$$

The value at $x = 0$ depends on the side from which the function is coming closer to $x = 0$ and is defined there as

$$\lim_{x \to 0} H(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases} \tag{2.2.2}$$

A plot of this function is shown in Figure B.2.

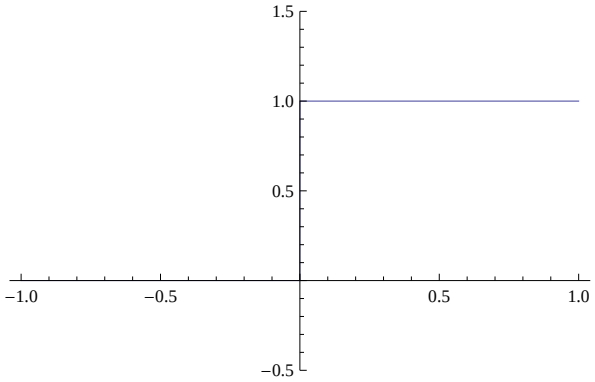


Figure B.2: Heaviside function $H(x)$

Appendix C

Barycentric coordinate system

The homogeneous coordinates for triangles are commonly the so-called *barycentric coordinates* and are denoted as $(\lambda_1, \lambda_2, \lambda_3)$ where one coordinate depends on the other two. The third coordinate is for convenience and the coordinates are defined as

$$\boldsymbol{\lambda} = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ 1 - \lambda_1 - \lambda_2 \end{pmatrix} \tag{3.0.1}$$

On the other hand one can say $\lambda_1 + \lambda_2 \leq 1$ which forces each point to lie inside the convex hull for the area spanned by λ_1 and λ_2. Figure C.1 gives an impression of this coordinate system.

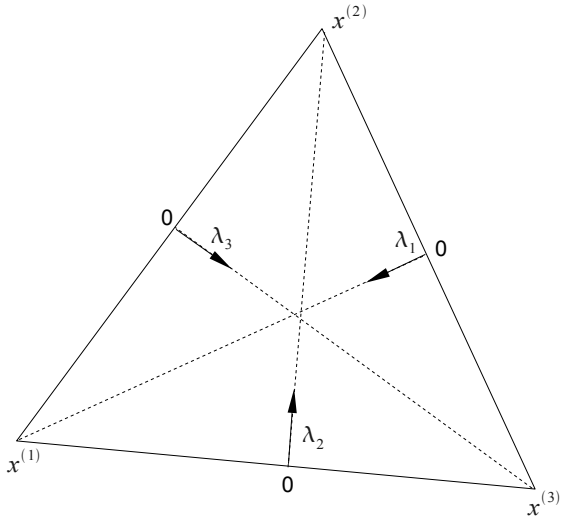


Figure C.1: Barycentric coordinates of a triangle

Appendix D

Installation guides

D.1 Application

D.1.1 Installation

The application depends on several packages which have to be installed first to guarantee a successful compilation. From the standard package mananger, aptitude for Debian (Ubuntu) and Yast for Suse the following packages have to be installed

- C++ Compiler (cobj++)
- Qt4 (qt4-devel)
- Coin3D (coin-3.0.0-devel)
- SoQt (soqt-1.4.2-devel)
- cmake (cmake-2.6)
- CUDA (nVidia's homepage)
- IcgCmakeModules (rpm.icg.tu-graz.ac.at)

Additionally to these packages the latest CUDA drivers™ and CUDA OpenToolkit™ has to be downloaded from the homepage of nVidia™. For this application the version 2.2 of CUDA was used. After installing and/or updating these packages the application can be configured by cmake using the command *./configure* in the root directory of the application. For the listed packages the corresponding cmake file is needed to find the package's libraries and includes. Nevertheless this might fail due to wrong include and library path variables. A check for installation paths and adaption inside the cmake configuration file *CMakeLists.txt* will fix the installation routine. Finally a *make* will compile the program.

D.1.2 Execution parameters

The software can be executed afterwards typing *deformation [file.iv]* where file.iv stands for an OpenInventor file containing the geometric data of the mesh and several deformation parameters. This data are collected in an own Inventor node called *SoDeformation*. As an example for a possible Inventor file consider the following inventor file.

```
1  SoDeformation {
2    object SoSeparator {
3      SoCube {}
4    }
5    depth 3
6    errorThreshold 0.2
7    elasticityConstant 80000
8    poissionRatio 0.3
9    pointerDevice PHANTOM
10   pointerSize 0.2
11 }
```

This example subdivides a cube's triangular mesh three times for the required level of depth and in the case of a pointer-object interaction the cube would be deformed. The parameters for the node are

object:	This field of the type SoNodeField contains the definition of the mesh which should be deformed. The node will be triangulated and stored for the internal mesh representation.
depth:	This defines how often the mesh's triangles should be subdivided to generate coarser or finer meshes for computation. The program subdivides each triangle, depending on this level, into subtriangles using the 1-to-4 subdivision method.
elasticityConstant:	For the calculation of the deformation this value stores the shear elasticity constant and is $80000 N/mm^2$ for steel. More information can be found in chapter 4.
possionRatio:	This is also a material depended constant and is 0.31 (dimensionless) for steel (Chapter 4).
pointerDevice:	The device used for the interaction with the deformable object. It can be set to PHANTOM if the PHANToM Omni force feedback device is connected or MOUSE if the pointer should be controlled by the mouse.
pointerSize:	The pointer is limited to a sphere and its radius can be defined via this parameter.

D.2 PHANToM device driver

D.2.1 Installation

To get a haptic feedback for the user, this application uses the Phantom Omni device from Sensable. The device returns forces in the direction of the three spatial axes, however effects no torsions around the limbs. Depending on the disk operation system, the phantom device driver can be installed using the corresponding software package. The package can be downloaded from Sensable's homepage in the Internet (www.sensable.com), which however is not for free, and can be installed via the console by typing

```
dpkg -i phantomdevicedriver.deb
```

for Debian and Ubuntu and by the command

```
rpm -i phantomdevicedriver.rpm
```

for Suse operation system. It has to be taken care to use the correct software for 32-bit or 64-bit systems. Additionally, at this point one has to mention, that SensAble at the moment only supports drivers for Linux kernels lower than 2.6.26. If this would be a problem, the system needs to be downgraded to such a kernel or lower. To do so an older kernel – the package is called linux-image-*version* – needs to be installed with the package-manager of the used distribution. Test cases with a 2.6.18 kernel have succeeded. In addition to the Phantom device driver, SensAble provides also a so-called OpenHaptics™ software containing libraries for easier access to the device. It includes functionality to configure, initialize and read and write values from and to the device. It can be installed in the same manner as above.

```
dpkg -i openhaptics.deb
```

for debian and Ubuntu and

```
rpm -i openhaptics.rpm
```

for Suse. The PHANToM Omni uses the firewire interface for communication and therefore the module *raw1394* has to be loaded.

```
lsmod | grep 1394
```

will help to figure out whether the module is loaded or not. For the case of a missing installation of the module *raw1394* superusers can install it from the package manager of by

```
apt-get install libraw1394
modprobe raw1394
chmod a+rw /dev/raw1394
```

and loading it afterwards. To gain access to the device for normal user the read and write rights have to be set by the *chmod* command.

D.2.2 Configuation and Testing

After a successful installation, the functionality of the device can be configured with the program *PHANToMConfiguration* (Figure D.1) and finally tested with the program *PHANToMTest* (Figure D.2) both located under */usr/sbin/*.

```
PHANToMConfiguration
PHANToMTest
```

In case that the position data of the limbs in PHANToMTest are distorted or wrong (Figure D.3, the PHANToMConfiguration application might create bad files. These created files are located under */etc/SensAble/PhantomDeviceDriver/* and of the type *Phantomxx.ini*. With a short look inside these files it can be checked whether the floating point for the floats is a dot or a comma. Commas will produce wrong results for the PhantomTest application. These files can be changed manually by replacing those wrong commas by a dot. This replacement has to be done each time PHANToMConfiguration application is executed recreating these files. An automated python script which can be executed afterwards to correct the files will look like this [Forsslund, 2009]

```
1 # -*- coding: utf-8 -*-
2 # Python
3
4 import shutil
5
```

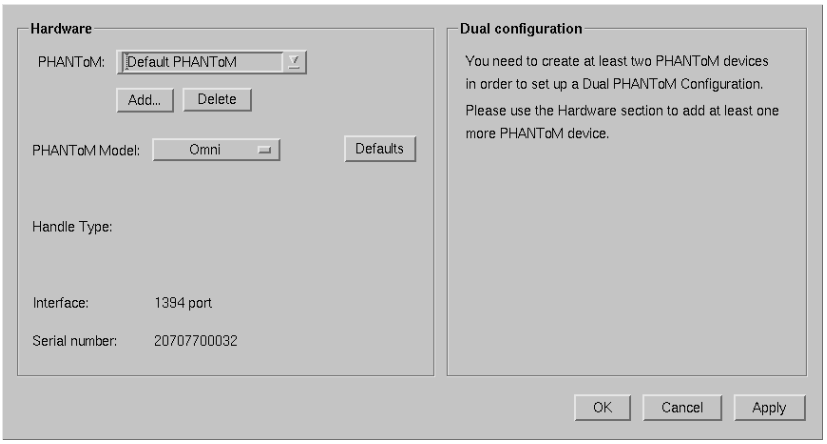


Figure D.1: Application PHANToMConfiguration

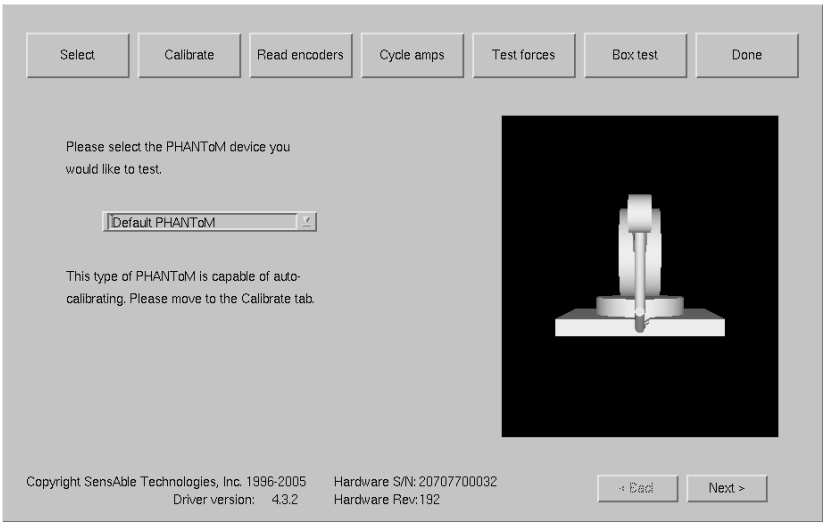


Figure D.2: Application PHANToMTest with correct limb positions

Figure D.3: Application PHANToMTest with wrong limb positions

```
6  print '**************************************************'
7  print '* Fixing ,-> . in floating point numbers         *'
8  print '* in /etc/SensAble/PHANToMDeviceDrivers/PHANToM0.ini  *'
9  print '* (backup file created)                          *'
10 print '* usage: sudo python phantomfix.py               *'
11 print '**************************************************'
12
13 shutil.copyfile('/etc/SensAble/PHANToMDeviceDrivers/PHANToM0.ini',
       '/etc/SensAble/PHANToMDeviceDrivers/PHANToM0.ini.bak')
14 input = open('/etc/SensAble/PHANToMDeviceDrivers/PHANToM0.ini.bak')
15 output = open('/etc/SensAble/PHANToMDeviceDrivers/PHANToM0.ini','w'
       )
16
17 for s in input:
18     s = s.replace(',0','.0');
19     s = s.replace(',1','.1');
20     s = s.replace(',2','.2');
21     s = s.replace(',3','.3');
22     s = s.replace(',4','.4');
23     s = s.replace(',5','.5');
24     s = s.replace(',6','.6');
25     s = s.replace(',7','.7');
26     s = s.replace(',8','.8');
27     s = s.replace(',9','.9');
28     output.write(s)
29 output.close()
30 input.close()
```

This software bug can also be avoided by changing the locale to *en_US*. The actually defined locale can be identified by the *echo* command and set to the American locale with *set* or *export* depending

on the installed Linux distribution.

```
echo \$LANG
set LANG=en_US
export LANG=en_US
```

Afterwards PHANToMConfiguration has to be executed again to generate correct configuration files.

Bibliography

Tomas Akenine-Moller and Eric Haines. *Real-Time Rendering (2nd Edition)*. AK Peters, Ltd., 2 edition, Jul 2002. ISBN 9781568811826. 53

Ferri Aliabadi. *The Boundary Element Method: Applications in Solids and Structures*. Wiley, May 2002. ISBN 9780470842980. 45

Nicolas Aspert, Diego Santa-Cruz, and Touradj Ebrahimi. MESH: Measuring errors between surfaces using the Hausdorff distance. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, volume I, pages 705–708, 2002. 120

Prasanta Kumar Banerjee. *The Boundary Element Methods in Engineering*. Mcgraw-Hill College, rev sub edition, Jan 1994. ISBN 9780077077693. 12

Jeffrey J. Berkley. Haptic devices. Mimic Technologies Inc. 4033 Aurora Ave N. Suite 201 Seattle, WA 98103, Mar 2003. 81

Martin Bertram, Mark A. Duchaineau, Bernd Hamann, and Kenneth I. Joy. Generalized B-spline subdivision-surface wavelets for geometry compression. *IEEE Trans. Vis. Comput. Graph*, 10 (Mar):326–338, 2004. 61

Dietrich Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge University Press, 3 edition, Apr 2007. ISBN 9780521705189. 10, 15

C. A. Brebbia. *Boundary Element Techniques in Engineering*. Butterworth-Heinemann, Jun 1979. ISBN 9780408003407. 15, 20, 45, 51

Edwin Catmull and Jim Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10:350–355, 1978. 12, 57

Charles K. Chui. *An introduction to wavelets*. Academic Press Professional, Inc., San Diego, CA, USA, 1992. ISBN 0-12-174584-8. 60

Ingrid Daubechies. *Ten Lectures on Wavelets (CBMS-NSF Regional Conference Series in Applied Mathematics)*. SIAM: Society for Industrial and Applied Mathematics, 1 edition, Jun 1992. ISBN 9780898712742. 60

Henk A. Van der Vorst. *Iterative Krylov Methods for Large Linear Systems (Cambridge Monographs on Applied and Computational Mathematics)*. Cambridge University Press, Jun 2003. ISBN 9780521818285. 77

M. Dontschewa, G. Kempter, P. Roux, A. Künz, and M. Marinov. Experimental set-up for haptic investigation of real and virtual environments. *E. Paper presentation at 13th International Scientific and Applied Science Conference on Electronics in Sozopol*, Sep 2004. 81

Daniel Doo. A subdivision algorithm for smoothing down irregularly shaped polyhedrons. In *Int'l Conf. Ineractive Techniques in Computer Aided Design*, pages 157–165, Bologna, Italy, 1978. IEEE Computer Soc. 58

Daniel Doo and Malcom Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer Aided Design*, 10(6):356–360, 1978. 58

Nira Dyn, David Levine, and John A. Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Trans. Graph.*, 9(2):160–169, 1990. ISSN 0730-0301. 56

Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, volume 29 of *Annual Conference Series*, pages 173–182. ACM SIGGRAPH, Addison Wesley, August 1995. 12, 61, 95

Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, Mar 2003. ISBN 9780321194961. 8

James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice in C (2nd Edition)*. Addison-Wesley Professional, 2 edition, Aug 1995. ISBN 9780201848403. 8

Jonas Forsslund. Phantom configuration file fixer, Nov 2009. URL `http://www.nada.kth.se/~jofo02/H3DTrunkGNULinux.html`. 147

Ivar Fredholm. Sur une classe d'equations fonctionelles. *Acta Mathematica*, 27:365–390, March 1903. 15

Lothar Gaul. Boundary element method, 2009. URL `http://www.iam.uni-stuttgart.de/bem/index.htm`. 8, 11

Lothar Gaul, Martin Kögl, and Marcus Wagner. *Boundary Element Methods for Engineers and Scientists*. Springer, 1 edition, Apr 2003. ISBN 9783540004639. 12, 17, 22, 23, 26, 27, 39, 40, 41, 45, 49

M. Gervautz, A. Hildebrand, and D. Schmalstieg, editors. *Virtual Environments '99: Proceedings of the Eurographics Workshop in Vienna, Austria*. Springer, illustrated edition edition, 6 1999. ISBN 9783211833476. 138

Martin H. Gutknecht. A brief introduction to Krylov space methods for solving linear systems. In Yukio Kaneda, Hiroshi Kawamura, and Masaki Sasai, editors, *Frontiers of Computational Science: Proceedings of the International Symposium on Frontiers of Computational Science 2005*, pages 53–62. Springer, Apr 2007. ISBN 9783540463733. 71

Ridha Hambli, Abdessalam Chamekh, and Hédi Bel Hadj Salah. Real-time deformation of structure using finite element and neural networks in virtual reality applications. *Finite Elem. Anal. Des.*, 42 (11):985–991, 2006. ISSN 0168-874X. 14

Volker Hähnke. Anatomic Simulation – Das Spring-Mass-Damper-Model. Technical report, Department of Computer Science, Johann Wolfgang Goethe University Frankfurt am Main, 2004. 12

Peter Hunter and Dr. Andrew Pullan. FEM/BEM notes. Technical report, Department of Engineering Science, University of Auckland, New Zealand, 2001. 51

Doug L. James and Dinesh K. Pai. Artdefo – accurate real time deformable objects. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, Annual Conference Series, pages 65–72, Los Angeles, 1999. ACM Siggraph, Addison Wesley Longman. 12, 51

M.A. Jaswon and G.T. Symm. *Integral Equation Methods in Potential Theory and Elastostatics (Computational mathematics and applications)*. Academic Press Inc.,U.S., Dez 1977. ISBN 9780123810502. 15

Leif Kobbelt. $\sqrt{3}$-subdivision. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 103–112, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN 1-58113-208-5. 55

Erwin Kreyszig. *Advanced Engineering Mathematics*. Wiley, 9 edition, 11 2005. ISBN 9780471488859. 39

A.M. Linkov. *Boundary Integral Equations in Elasticity Theory (Solid Mechanics and Its Applications)*. Springer, 1 edition, Apr 2002. ISBN 9781402005749.

Yongchao Liu, Douglas L. Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-waterman sequence database searches for CUDA-enabled graphics processing units, 2009. URL `http://www.biomedcentral.com/1756-0500/2/73`. 110, 111

Charles Loop. Smooth subdivision surfaces based on triangles. Master's thesis, Utah University, USA, 1987. 12, 54

Michael Lounsbery, Tony D. DeRose, and Joe Warren. Multiresolution analysis for surfaces of arbitrary topological type. *ACM Trans. Graph.*, 16(1):34–73, 1997. ISSN 0730-0301. 61

Frank Luna. *Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach (Wordware Game and Graphics Library)*. Jones and Bartlett Publishers, 1 edition, Jun 2006. ISBN 9781598220162. 8

Frank D. Luna. *Introduction to 3D Game Programming with DirectX 10*. Jones and Bartlett Publishers, 1 edition, Okt 2008. ISBN 9781598220537.

St´ephane G. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(7):674–693, 1989. ISSN 0162-8828. 60

Andreas Meister. *Numerik linearer Gleichungssysteme. Einführung in moderne Verfahren.* Vieweg Verlagsgesellschaft, Nov 1999. ISBN 9783528031350.

Ken'Ichi Morooka, Xian Chen, Ryo Kurazume, Seiichi Uchida, Kenji Hara, Yumi Iwashita, and Makoto Hashizume. Real-time nonlinear FEM with neural network for simulating soft organ model deformation. In *MICCAI '08: Proceedings of the 11th International Conference on Medical Image Computing and Computer-Assisted Intervention, Part II*, pages 742–749, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85989-5. 14

nVidia. *NVIDIA CUDA Programming Guide 2.0.* nVidia, 2008. 8, 111, 115

Jörg Peters and Ulrich Reif. *Subdivision Surfaces (Geometry and Computing)*. Springer, 1 edition, Jun 2008. ISBN 9783540764052. 53

Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, Mar 2005. ISBN 9780321335593. 53, 58, 59

H.T. Rathod, K.V. Nagaraja, B. Venkatesudu, and N.L. Ramesh. Gauss legendre quadrature over a triangle. *Journal Indian Institute of Science*, 84:183–188, 2004. doi: http://journal.library.iisc.ernet.in/vol200405/paper6/rathod.pdf. 41

Thomas Rüberg. *Non-conforming FEM/BEM Coupling in Time Domain*. Technical University of Graz, Mar 2008. ISBN 3902465980. 10, 17, 20, 23

Yousef Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, 2 edition, Apr 2003. ISBN 9780898715347. 63, 64, 68, 70, 71, 77, 78

Christoph Schwab. *Randelementmethoden*. Teubner B.G. GmbH, Jun 2004. ISBN 9783519003687. 17

Peter Sonneveld. Cgs, a fast lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10(1):36–52, 1989. ISSN 0196-5204. 77

Jos Stam. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. *Computer Graphics (SIGGRAPH '98) Proceedings*, pages 395–404, Jul 1998.

Eric J. Stollnitz, Anthony D. DeRose, and David H. Salesin. *Wavelets for Computer Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, 1st edition, Jan 1996. ISBN 9781558603752. 61

L. C. Wrobel. *The Boundary Element Method*. Wiley, 1 edition, Mar 2002. ISBN 9780471720393. 11, 12

Seung-Hyun Yoon. *Sweep-based Approach to Three-Dimensional Shape Deformations: Sweep-based Shape Deformations*. VDM Verlag, May 2008. ISBN 9783639030716. 12

Yongmin Zhong, Bijan Shirinzadeh, Gursel Alici, and Julian Smith. Haptic deformation modelling through cellular neural network. 2008. 14