Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 39

Development of a FEM code for fluid-structure coupling

Stephan Herb

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Miriam Mehl
Betreuer/in:	Dipl.-Ing. Florian Lindner
Beginn am:	08. Juni 2015
Beendet am:	08. Dezember 2015
CR-Nummer:	G.1.8, G.4, I.6, J.2

Abstract

The development of a FEM structure solver for a coupled fluid-structure interaction simulation is presented in this thesis. Different aspects were created to evaluate existing FEM libraries. The libMesh framework was considered to be best suitable and was used in the development of the program. Two different discretizations of flat shell elements were implemented, a triangular and a quadrilateral element. The shell elements are constructed by the superposition of plane and plate elements. For both, the plane and plate elements, different existing models were considered focusing on thin-plate theory and membranes. The developed program offers two versions, one coupled version to be used in a multi-physics simulation and a stand-alone version whose intention was to better validate the implemented finite elements. Both versions are parallelized with MPI. The coupling environment preCICE was used in the development of the coupled version. The validation of the elements showed good accuracy for the plane element components compared to analytical solutions as well as commercially available software. The plate element's accuracy is lower compared to the plane elements due to the chosen models that have a simpler approximation of the physical circumstances. The superimposed shell element's accuracy is well suited to be used in the structure solver. For both finite elements the accuracy can be increased arbitrarily by subdividing the mesh further. The parallelization test showed a good scaling with the number of processes for the assembly of the system's matrix and right-hand side as well as for the solving step. As an coupling example, a fluid-structure coupling between the developed program and a fluid solver developed with openFOAM was tested. The developed structure solver showed good performance in the simulation and is qualified for further multi-physics simulations connected via preCICE.

Contents

1 Introduction

The finite element method (FEM) is a method for finding numerical approximate solutions to field problems for partial differential equations [CMPW02]. It subdivides the whole problem domain into small parts, called finite elements, distinguishing them from infinitesimal element used in calculus. Since the spatial variation of each element is described in simpler form, for example by polynomial terms, FEM only provides an approximate solution. Despite the fact of inexact solutions by FEM, it is applicable to many fields in practice: Heat transfer, stress analysis or magnetic fields, for instance. The analyzed structure is not limited to any shape and by refining the mesh the approximation can be improved easily. Therefore, FEM is widely used in fluid and structure simulations [CMPW02].

While simulations of fluids and structure for their own are present for many decades, fluid-structure interactions (FSI) are much younger due to the computational complexity when combining both physical fields in one simulation [Gat15]. Nowadays FSI simulations are not the only examples of multi-physics simulations, as fluid-structure-acoustics ([LKBD09], [SMU+10]) even combine three physical fields in one simulation. When dealing with multi-physics problems one has two options: Try to hold all parts of the problem together in one application code independent from other computing applications. This is also called the monolithic approach [Gat15]. The other option is to break the problem apart into smaller pieces that can be better managed in development and recombine the parts to a coupled solution. This is denoted as loose coupling or partitioned approach [LCY+06]. The challenge of the second option is to preserve stability of the simulation. The benefits on the other hand are the possibilities to reuse existing code and to encapsulate the solver codes from each other and the coupling functionality. With preCICE a coupling tool was introduced that eases the combination of multiple different solver codes into one simulation, following the idea of the partitioned approach.

The goal of this work is to develop a structure solver with FEM that is applicable to a multi-physics simulations like a fluid-structure interaction coupling. The program uses a FEM library which provides data structures and functionalities to ease programming parts like the initialization, setup and solving of the system. In order to find a suitable framework, an evaluation based on several aspects is performed on different FEM frameworks in the thesis.

The solver works on two-dimensional meshes consisting of triangular and quadrilateral elements. In the scope of this thesis flat shell elements are implemented. A flat shell element can be constructed by superimposing a plane and a plate element. For this reason six finite element models are implemented: One plane, plate and shell element sharing a three-node triangular finite element approach, the others sharing a four-node quadrilateral finite element approach. The triangular and quadrilateral plane element is implemented based on the model description from [Ste15]. The triangular plate element is based on a model from [Spe88], while the quadrilateral plate elements implements the Discrete Kirchhoff Quadrilateral (DKQ) element introduced by [ZT00]. Both belongs to the group of models sharing the thin-plate theory of Kirchhoff [Ste15].

The implementation of the FEM-code gets illustrated in detail, including a practical

examples of features of the FEM framework that was chosen in the evaluation. The program is parallelized with the help of MPI and the preCICE API is integrated to make it applicable for multi-physics coupling simulations. A validation of the implemented models as well as the parallelization and the coupling is performed to demonstrate the correctness and accuracy of the implementation and benefits from parallel execution.

1.1 Organization

The thesis is made up of seven chapters including the introduction. A detailed list of aspects used to evaluate several FEM frameworks is presented in chapter two, along with short outlines of the surveyed libraries and a motivation of the chosen library. Chapter three contains the mathematical derivations of the shell elements. After a motivation of linear elastic problems, first the plane element in its triangular and quadrilateral form is described, following by the plate elements. Before the construction of the shell elements, a section about coordinate transformation is presented. Chapter four describes details on the implementation process of the FEM code. First, an introduction to libMesh is shown. The remainder of this chapter describes the single parts of the code, like the mesh file import or the system matrix assembly, as well as the parallelization of the program with MPI. In chapter five the coupling via preCICE is presented. An overview of preCICE is given and details on the coupling with it is described, including the different coupling methods, data mappings and communication methods. The integration of preCICE into the existing structure solver code closes chater five. Chapter six contains many example problems with the help of which the shell elements and its components were validated. Also, the parallelization and the coupling were tested in this chapter. The seventh and last chapter presents a summary and discussion of the results and gives suggestions for future development.

2 Framework Evaluation

Part of the thesis was to analyze several frameworks which ease the work with the finite element method. An evaluation of frameworks was done to select a suitable one for the given task. The evaluation's criteria are presented in this chapter together with a short description of the studied frameworks and a motivation for the library that were finally chosen.

2.1 General Aspects

In preparation of evaluating the frameworks many criteria were created in order to find the most suitable library. The individual aspects are as follows:

- Open Source: All frameworks under consideration need to be published under a free license that allows modification and redistribution. The implemented program should be used by anyone without requiring to purchase additional commercial software.

- Parallelization: Due to the need of accelerating calculations, the framework has to support an implementation of the widely used Message Passing Interface (MPI). The library should be able to use MPI internally for its own functions and procedures, but also support the developer with auxiliary function for communicating framework-related data.

- The programming language was chosen to be C++. This aspect is subjective and due to the individual experience of the author that is larger than, for example, with Python. It is not required that the framework is internally written in C++, but if not, it must provide an interface to C++ to work with.

- Mesh file import: Since the program should be able to work with complex geometries, the definition of such a mesh cannot be done in source code. Therefore, the library should provide a possibility to import meshes from file. The file must contain definitions of the node positions as well as the topology of the elements. Additionally, boundary conditions must be definable through identifiers at nodes or edges. The actual formats of the mesh files supported is not important, as long it supports the addressed requirements. A framework proprietary format would also be possible if it is simply reproducible.

- Linked to the previous aspect is the variety of different finite element types the library has to provide. The current program version uses triangles and quadrilaterals with three and four nodes, respectively. To be able to expand the program's functionality in the future, the library should support elements like six node triangles, nine node quadrilaterals or three dimensional elements like tetrahedra and hexahedra.

- Built-in solvers: The framework must provide a variety of different iterative solvers that can be interchanged at runtime by the user, or at least easily in the source

code. Thus, the user has a higher flexibility when optimizing a problem or comparing different solvers in order to find the best for his or her problem. In the context of a structure solver for linear elastic problems it is focused especially on the group of linear iterative solvers.

- Accessible and detailed documentation: In order to guarantee maintainability and expandability the framework has to have a good documentation. This includes a complete documentation of every class and function, that is actively growing with the library itself. Additional documented example codes help the developer learning how to use the library correctly and efficiently. If problems with the framework occur the developer should be able to get in contact with the framework developers via mailing-lists or forums.

- Up-to-date: The framework should be well maintained and actively supported by its developers to ensure a long term compatibility with new features of the program code.

- The framework should be used by at least a few projects or has been part of publications. This shows the framework's importance and usability.

- Easy to learn syntax and structure: A rather subjective aspect, not less important. The developer should be able to concentrate more on the mathematical/physical problems and less on programming details. This accompanies the choice of the programming language as well as the documentation aspect.

2.2 Frameworks Overview

The following list contains the FEM libraries which were evaluated in detail.

2.2.1 Feel++

Feel++ stands for "'Finite Element Embedded Language in C++"' and is a unified framework for finite and spectral Galerkin methods in 1D, 2D and 3D to solve partial differential equations [PCD$^+$12]. It was created in 2005 and is still actively maintained with daily commits and the last release version from February, 2015. One main aspect of Feel++ is to have syntax, semantics and pragmatics close to the mathematics. It allows creation of versatile mathematical kernels for testing and comparing different techniques and methods in solving problems. A second aim is to have a small library that is well manageable but makes use wherever possible of established libraries, for example to solve linear systems. It interfaces seamlessly with MPI simplifying the work's program parallelization. Different mesh file formats are available for import, like the GMSH format and a collection of different finite element types are supported. It is currently used in projects at Cemosis (Center for Modeling and Simulation in Strasbourg, France) including fluid structure interactions, high field magnets simulation and optical tomography [Con].

2.2.2 OOFEM

[Pat09] The "'Object Oriented Finite Element Method"' framework is an multi-physics finite element code with object oriented architecture [PB01]. The aim of is to provide an efficient and robust tool for FEM computations as well as to offer highly modular and extensible environment for development [Pat09]; extensible in terms of new element types, boundary conditions or numerical algorithms that can be created by the user. OOFEM focuses on efficient and robust solution to mechanical, transport and fluid problems providing corresponding modules for it. It is written in C++ with focus on high portability between platforms and interfaces various external software libraries like PETSc, ParMETIS, and ParaView. The last stable release were published on February, 2014 but it is still actively developed. The framework is used in several publications [Pat].

2.2.3 GetFEM++

GetFEM++ is a generic finite element C++ library. It aims at providing finite element methods and elementary matrix computations for solving (non-)linear problems numerically. The user describes a model by gathering the variables, data and terms of a problem and some predefined bricks representing classical models. It allows easy switching from one method to another due to separation of geometric transformation or integration methods, for instance. It uses MPI for parallelization, though it is stated that "'a certain number of procedures are still remaining sequential"' [SPR] at the time this work were written. While the framework is implemented in C++, it provides interfaces to languages like Python and Matlab. Thus, the framework can be handled by scripts written in non-C++ languages. The latest release is dated from July, 2015 with daily commits from the developers. GetFEM++ is used in project like IceTools [Jar] (open source model for glaciers), EChem++ [BLSS] (Problem Solving Environment for Electochemistry) and SimNIBS [Thi] (software for Simulation of Non-invasive Brain Stimulation) and is part of some publications [PR].

2.2.4 MFEM

The "Modular Finite Element Method" library acts as a toolbox that provides the building blocks for developing finite element algorithms. MFEM aims to enable research and development of scalable finite element discretization and solver algorithms through general finite element abstractions. It has a wide variety of 2D and 3D finite element types, for example triangular and quadrilateral 2D elements, curved boundary elements or topologically periodic meshes. In addition to Galerkin methods, MFEM supports mixed finite elements, discontinuous Galerkin (DG) methods, or isogeometric analysis methods, for instance. The framework supports MPI-based parallelism throughout the library with minimal changes to the code to make the serial code parallel. A variety of solvers are available for the resulting linear algebra systems, including serial and parallel smoothers, solvers such as PCG, MINRES and GMRES, high-performance preconditioners from the hypre library, to name a few [mfeb]. The latest release of MFEM is dated

January, 2015 but it is still under active development. The library is used in several publications [mfea].

2.2.5 libMesh

[KPSC06] The libMesh library provides a framework for the numerical simulation of partial differential equations using arbitrary unstructured discretizations on serial and parallel platforms. A major goal of the library is to provide support for adaptive mesh refinement (AMR) computations in parallel while allowing a research scientist to focus on the physics they are modeling. It makes use of existing software whenever possible. PETSc can be used for the solution of linear systems on both serial and parallel platforms, and LASPack is included with the library to provide linear solver support on serial machines, for instance. LibMesh supports a variety of 1D, 2D, and 3D geometric and finite element types and seamlessly integrates parallel functionality with MPI throughout the whole library [KPSC06]. The framework is actively developed with daily commits and the latest release is dated from February, 2015. It is part of many publications [lib].

LibMesh was chosen to be the most suitable framework for this thesis. Its seamlessly integration of MPI reduces the effort of parallelization for the user, the required finite element types are supported and a range of many more are available for future expansions of the program's code. Complex geometries can be defined in several different mesh format as well as a simple libMesh particular format. The integration of external libraries like PETSc gives a high flexibility for users to change solvers and/or preconditioners. The library's classes and functions are well documented and a large collection of example codes are available. It is actively maintained by fixing bugs and adding more features. Although the structure and components of the framework are intuitive to understand, guided by the documentation, support by the developers are provided through mailing-lists.

3 Shell Elements

Shell elements are used when structures need to be modeled where plane stresses as well as bending stresses are present. Different types of shell elements are available, including curved shell elements, shells of revolution, generalized shells or flat shell elements [CMPW02]. The flat shell elements are subject of this work. Such an element can be constructed by the superposition of a plane and a plate element. This section contains the fundamentals of linear elastic problems - the field of application for the shell element - as well as the mathematical derivation of the plane and plate element, their combination resulting in the shell element and the necessary coordinate transformation that is needed for the implementation.

3.1 Introduction to Linear Elastic Problems

In the following the fundamental equations of linear elasticity will be considered. Here, the spatial case is used for demonstration, but every lower dimensional problem can easily be derived from it. The following definitions will be used in this thesis:

$$\vec{u} = \begin{pmatrix} u & v & w \end{pmatrix}^T \text{ displacement vector} \quad \vec{f} = \begin{pmatrix} f_x & f_y & f_z \end{pmatrix}^T \text{ external force vector} \tag{1}$$

The strains and stresses can either be described in form of tensors $\underline{\varepsilon}$ and $\underline{\sigma}$, or as vectors $\vec{\varepsilon}$ and $\vec{\sigma}$:

$$\underline{\varepsilon} = \begin{pmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{yx} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{zx} & \varepsilon_{zy} & \varepsilon_{zz} \end{pmatrix} \tag{2}$$

$$\underline{\sigma} = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{pmatrix} \tag{3}$$

$$\vec{\varepsilon} = \begin{pmatrix} \varepsilon_{xx} & \varepsilon_{yy} & \varepsilon_{zz} & 2\varepsilon_{xy} & 2\varepsilon_{yz} & 2\varepsilon_{zx} \end{pmatrix}^T \tag{4}$$

$$\vec{\sigma} = \begin{pmatrix} \sigma_{xx} & \sigma_{yy} & \sigma_{zz} & \sigma_{xy} & \sigma_{yz} & \sigma_{zx} \end{pmatrix}^T \tag{5}$$

In the case of isotropic materials the strains and stresses are symmetrical, i.e. $\varepsilon_{xy} = \varepsilon_{yx}, \varepsilon_{xz} = \varepsilon_{zx}, \varepsilon_{yz} = \varepsilon_{zy}$ and $\sigma_{xy} = \sigma_{yx}, \sigma_{xz} = \sigma_{zx}, \sigma_{yz} = \sigma_{zy}$. As stated in [Ste15] the relation between displacements and strains is as follows:

$$\underline{\varepsilon} = \frac{1}{2} \left(\nabla \vec{u} + \vec{u} \nabla \right) \tag{6}$$
$$\vec{\varepsilon} = \underline{L}\vec{u}$$

Equation (6) relates the displacement vector field $\vec{u}$ with the strain field $\underline{\varepsilon}$, or $\vec{\varepsilon}$ respectively. Here, $\underline{L}$ is a differential operator. This strain-displacement relation is also called *kinematic relationship* [Ste15].

In general initial strains can exist inside the material for example due to temperature changes or shrinkage. Such initial strains are denoted $\vec{\varepsilon_0}$ and the stresses will be influenced by the difference between the actual and initial strains. Additionally one could imagine initial residual stresses $\vec{\sigma_0}$ that can be added to the general equation:

$$\vec{\sigma} = \underline{D}\left(\vec{\varepsilon} - \vec{\varepsilon_0}\right) + \vec{\sigma_0} \tag{7}$$

where $\underline{D}$ is the material matrix. In the simplest case of linear elasticity with isotropy, $\underline{D}$ only contains two parameters, namely the elastic modulus E (also known as the Young's modulus) and the Poisson's ratio ν. The former one defines the relationship between the stress and strain in a material, the latter one results as the quotient of the fraction of expansion and the fraction of compression for small changes.

In the following the initial conditions are ignored, resulting in the a simpler form of equation (7):

$$\vec{\sigma} = \underline{D}\,\vec{\varepsilon} \tag{8}$$

For the said isotropic case $\underline{D}$ results in [ZT00]:

$$\underline{D} = \frac{E}{1-\nu^2}\begin{pmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{pmatrix} \tag{9}$$

The equilibrium conditions as described in [Ste15], are as follows:

$$\underline{L}\vec{\sigma} + \vec{b} = \vec{0} \tag{10}$$

where the vector $\vec{b}$ describes internal volume forces and $\vec{0}$ is the zero vector.

Two different boundary conditions are distinguished: Essential and natural boundary conditions. The first one is geometrically described. An initial displacement $\vec{u^0}$ is impressed on a surface part Ω_u of the object:

$$\vec{u} = \vec{u^0} \quad \text{on} \quad \Omega_u \tag{11}$$

The natural boundary conditions are represented by force conditions. They can be described as follows:

$$\underline{n}\vec{\sigma} = \vec{p^0} \quad \text{on} \quad \Omega_p \tag{12}$$

where the matrix $\underline{n}$ contains entries of the object boundary's normal vector, $\vec{p^0}$ described the edge stress and $\vec{\sigma}$ is the stress vector:

$$\vec{n} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \tag{13}$$

$$\underline{n} = \begin{pmatrix} n_x & 0 & 0 & n_y & 0 & n_z \\ 0 & n_y & 0 & n_x & n_z & 0 \\ 0 & 0 & n_z & 0 & n_y & n_x \end{pmatrix} \tag{14}$$

In [Ste15] the principle of the total potential for the fundamental equations of linear elasticity is illustrated. Initial point is the weak formulation of the equilibrium conditions (10) for a body V:

$$\int_V \left(\underline{L}\vec{\sigma} + \vec{b} \right) \vec{\lambda} \; dV = 0 \tag{15}$$

The introduction of Lagrange multipliers $\vec{\lambda}$ leads to a weighting and can be expressed as variation of $\delta\vec{u} = \vec{\lambda}$. The result of the derivation of the above equation is the expression of the total potential Π:

$$\Pi = \frac{1}{2} \int_V \vec{\sigma}^T \vec{\varepsilon} \; dV - \left(\int_V \vec{b}^T \vec{u} \; dV - \int_{\Omega_p} \vec{p^0}^T \vec{u} \; d\Omega_p \right) = \Pi_D - \Pi_e \tag{16}$$

The first term describes the elastic deformation energy Π_D, the two terms in brackets combine the potential of the external forces Π_e

3.2 Plane Element

Plane elements are characterized by that loads are only apply in the mid-surface directions of an element and all displacements, strains and stresses happen in the mid-surface, too. In this section plane elements are discussed in more detail and two different discretizations, a three-node triangular element and a four-node quadrilateral element, are described.

3.2.1 Problem Definition

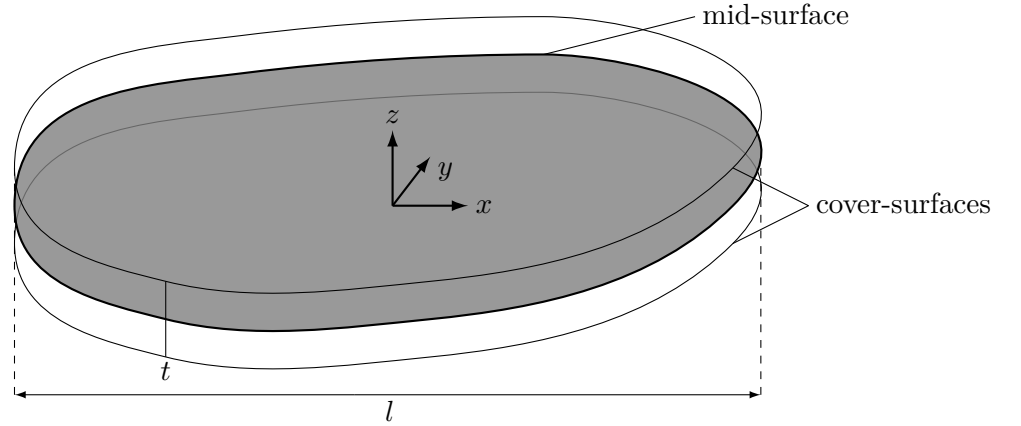Figure 1: Schematic view of a plane object with main dimension l and thickness t. The two cover-surfaces are located at $z = \pm\frac{t}{2}$, the mid-surface at $z = 0$.

In Figure 1 an object is shown which extends to the x and y axis as its primary direction. The extend in z-direction is smaller and denoted by thickness t. The mid-surface located in between the top and bottom surface areas has the coordinate $z = 0$.

14

Its local z-axis equals the normal vector of the mid place. Such an object is called *plane* in the following.

There are two different formulations regarding plane elements: Plane stress and plane strain. The directions of displacements u and v along the orthogonal local x and y axis defining its displacement field is a common feature of both problems. Also, both have in common, that only strains and stresses in the xy plane have to be considered: Instead of nine, only three components remain. While in the case of plane stress all other stress components are zero, in plane strain the stress in direction perpendicular to the xy plane is non-zero. In this thesis only plane stress will be discussed in further detail. More information about plane strain is given in [ZT00] and [Bra07], for instance. The following conditions must be satisfied such that a plane can be in *plane stress* [Ste15]:

- The thickness t varies only slightly and it must hold: $t/l \ll 1$, with l the extent of the larger side of the plane element.

- The load is applied to the mid-surface.

- Displacements, strains and stresses are constant across the thickness.

The stress components $\sigma_{xz}, \sigma_{yz}, \sigma_{zz}$ normal to the surface areas with $z \pm t/2$ vanish (equals zero). Therefore only the two normal stress components σ_{xx} and σ_{yy} and the transverse stress component σ_{xy} are left non-zero.

Displacements can only occur in x and y direction. u will be the displacement along x and v along y. The displacement field $\vec{u}$ is as follows:

$$\vec{u} = \begin{pmatrix} u(x,y) & v(x,y) \end{pmatrix}^T \tag{17}$$

The vector for the strain components:

$$\vec{\varepsilon} = \begin{pmatrix} \varepsilon_{xx} & \varepsilon_{yy} & 2\varepsilon_{xy} \end{pmatrix}^T \tag{18}$$

Sometimes $2\varepsilon_{xy}$ is shortened to γ_{xy} [Ste15]. The vector holding the stress components is similar to that of the strain's vector:

$$\vec{\sigma} = \begin{pmatrix} \sigma_{xx} & \sigma_{yy} & \sigma_{xy} \end{pmatrix}^T \tag{19}$$

The kinematic relationship $\vec{\varepsilon} = \underline{L}\vec{u}$ (eq. (6)) linking the displacements $\vec{u}$ with the strains $\vec{\varepsilon}$ can be written at full length:

$$\vec{\varepsilon} = \begin{pmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\varepsilon_{xy} \end{pmatrix} = \begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \underline{L}\vec{u} \tag{20}$$

With the strains known and considering equation (6) one can calculate the stresses $\vec{\sigma}$:

$$\vec{\sigma} = \begin{pmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{pmatrix} = \frac{E}{1-\nu^2} \begin{pmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{pmatrix} \begin{pmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\varepsilon_{xy} \end{pmatrix} = \underline{D}\vec{\varepsilon} = \underline{D}\ \underline{L}\vec{u} \tag{21}$$

After Steinke [Ste15], the essential boundary conditions $\vec{u} = \vec{u^0}$ and the natural boundary conditions $t\,\underline{n}\,\vec{\sigma} = \vec{q^0}$ will be applied onto the objects boundary Γ, with t is the object's thickness and $\vec{q^0}$ represents a line load. The matrix $\underline{n}$ looks as follows [Ste15]:

$$\underline{n} = \begin{pmatrix} n_x & 0 & n_y \\ 0 & n_y & n_x \end{pmatrix} \tag{22}$$

Beside line loads, nodal forces $\vec{F} = \begin{pmatrix} F_x & F_y \end{pmatrix}^T$ are also possible.

The total potential of the plane element problem looks as follows [Ste15]:

$$\Pi = \frac{1}{2} \int_V \vec{\varepsilon}^T \vec{\sigma} dV - \int_V \vec{u}^T \vec{b}\, dV - \int_{\Gamma_q} \vec{u}^T \vec{q}\, d\gamma - \vec{u}^T \vec{F} \tag{23}$$

The first term describes the elastic strain energy, the last three terms describe the different external forces. The first of the three describes the potential of the volume forces $\vec{b}^T = \rho \begin{pmatrix} g_x & g_y \end{pmatrix}$, where ρ is the object's density and g_x, g_y represents accelerations in the two directions. The second term stands for the line load along an edge Γ_q and the last term contains the single nodal forces.

The external forces are summarized in this work to the last term only. Line loads and volume forces can be converted to nodal load, assuming a linear distribution over the edges and the interior of the element, respectively [Ste15]. Therefore, the total potential of the plane element shrinks to the following:

$$\Pi = \frac{1}{2} \int_V \vec{\varepsilon}^T \vec{\sigma} dV - \vec{u}^T \vec{F} \tag{24}$$

3.2.2 Tri-3 Plane Element

In section 3.2.1 the plane element's functional was derived (eq. (24)). Now the focus is on the functional's discretization. The constructed finite element will be a three-node triangular element in the following also denoted as **Tri-3**. Figure 2 shows a general, planar object defined to be placed in the xy-plane. The first discretization step is to divide the object into single triangles approximating the shape of it. This process is called triangulation. Every one of these triangles then represents a finite element with one node at every corner. The finer the triangulation is done the better the object and its boundary are matched by its discrete complement, but also the more finite elements have to be considered in later calculations.

One triangular finite element is shown in Figure 3. It is defined by the coordinates (x_i, y_i) of its three nodes. Since the element is located in the xy-plane, the z-coordinate is of no interest and will be ignored. At every node, forces can be applied denoted with F_{x_i} and F_{y_i}. Accordingly, every node can be displaced. The movement along the x-axis is denoted with u_i and with v_i along the y-axis, respectively. Note, that the node numbering is in anti-clockwise direction. This convention will be kept throughout the thesis, and is important when implementing the FEM-code. In this thesis only triangles defined by three nodes are discussed. There are many more finite elements forming triangles, such
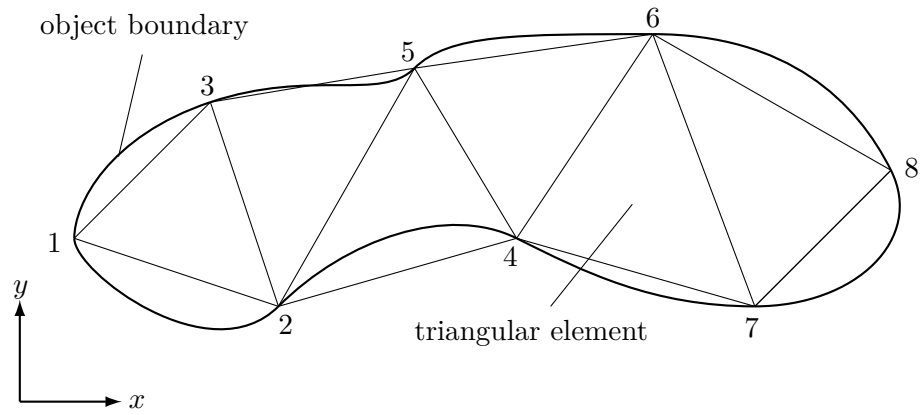
Figure 2: A plane object is partitioned into several triangular elements. This process is called triangulation.
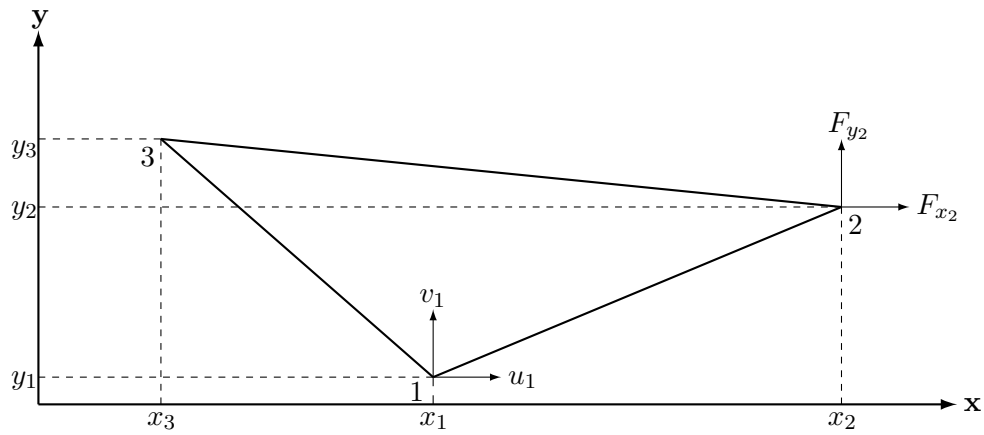

Figure 3: A three-node triangular element is shown with its coordinates projected onto the axes. For the first node its nodal displacements and for the second node its nodal forces are exemplarily illustrated.

as six node triangles or even seven node triangles. The main difference between these types of elements are the order of shape functions. More details about higher order triangular finite elements can be found in [ZT00], [BF85], [CMPW02], [Bra07].

In the case of a three node triangle the basis functions for the two displacements u and v are the same and thus u and v can be replaced by an arbitrary function ψ[Ste15]:

$$\psi(x,y) = a_0 + a_1 L_1 + a_2 L_2 = \begin{pmatrix} 1 & L_1 & L_2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \vec{x}^T \vec{a} \tag{25}$$

defined in triangular coordinates (see Figure 4).
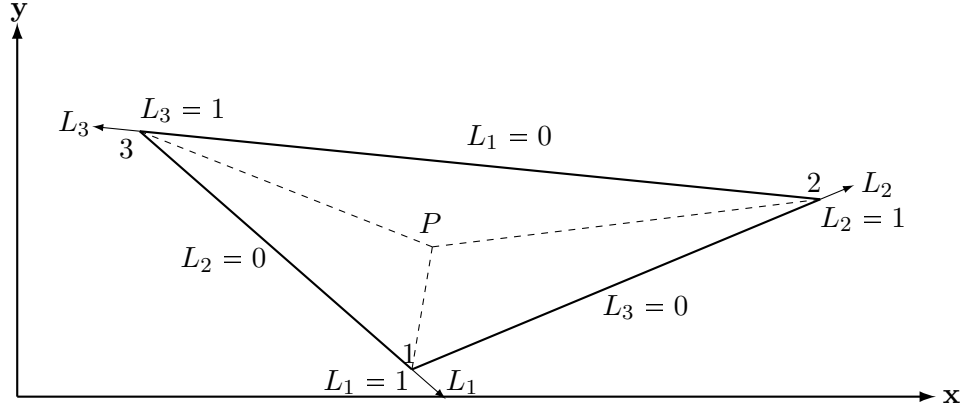
Figure 4: Triangular coordinates L_1, L_2, L_3 and sampling point P within a triangular element. The nodes and edges represent special cases of triangular coordinates.

To get the unknown coefficients a_i, values for the triangular coordinates are set. This creates a system of linear equations:

$$\psi(L_1 = 1, L_2 = 0) = \psi_1 \rightarrow \psi_1 = a_0 + a_1$$
$$\psi(L_1 = 0, L_2 = 1) = \psi_2 \rightarrow \psi_2 = a_0 + a_2$$
$$\psi(L_1 = 0, L_2 = 0) = \psi_3 \rightarrow \psi_3 = a_0 \tag{26}$$

Written as matrix and vector:

$$\underline{A}\vec{a} = \vec{\psi}$$
$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \end{pmatrix} \tag{27}$$

Now, inverting matrix A the coefficients can be found:

$$\vec{a} = \underline{A}^{-1}\vec{\psi} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \end{pmatrix} \tag{28}$$

18

If one put equation (28) into (25), the shape functions for the three node triangular finite element will be derived, as described in [Ste15]:

$$u = \vec{x}^T \vec{a} = \vec{x}^T \underline{A}^{-1} \vec{u} = \vec{N}^T \vec{u}$$

$$\vec{N}^T = \vec{x}^T \underline{A}^{-1} = \begin{pmatrix} 1 & L_1 & L_2 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix}$$

$$= \begin{pmatrix} L_1 & L_2 & 1 - L_1 - L_2 \end{pmatrix} = \begin{pmatrix} N_1 & N_2 & N_3 \end{pmatrix} \tag{29}$$

Characteristically for a shape functions N_i is, as stated in [Ste15], that it evaluates to 1 at node i and to 0 at the two other nodes. The functions are linear with respect to L_1 and L_2 which can be noticed in equation (29). As stated before, these shape functions are the same for displacement u and v. With the knowledge of the displacement values of the element's nodes one can formulate the displacement functions in triangular coordinate notation as follows:

$$u = N_1 u_1 + N_2 u_2 + N_3 u_3$$
$$v = N_1 v_1 + N_2 v_2 + N_3 v_3 \tag{30}$$

Or in matrix form:

$$\vec{\tilde{u}} = \underline{N} \vec{u}$$

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} N_1 & 0 & N_2 & 0 & N_3 & 0 \\ 0 & N_1 & 0 & N_2 & 0 & N_3 \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \end{pmatrix} \tag{31}$$

The vector $\vec{\tilde{u}}$ describes the element's displacements as product of matrix $\underline{N}$ containing the shape functions and vector $\vec{u}$ containing the displacements of the single triangle's nodes. Now, one can put equation (31) into (20):

$$\vec{\varepsilon} = \underline{L} \vec{\tilde{u}} = \underline{L} \, \underline{N} \vec{u} = \underline{B} \vec{u} \tag{32}$$

The product of $\underline{L}$ and $\underline{N}$ is called *strain-displacement matrix* $\underline{B}$. In order to calculate the strain-displacement matrix, one has to assemble the $\underline{L}$ matrix containing the first partial derivatives of the triangular element. With the chain rule applied, the partial derivatives look as follows:

$$\frac{\partial}{\partial L_1} = \frac{\partial x}{\partial L_1} \frac{\partial}{\partial x} + \frac{\partial y}{\partial L_1} \frac{\partial}{\partial y}$$
$$\frac{\partial}{\partial L_2} = \frac{\partial x}{\partial L_2} \frac{\partial}{\partial x} + \frac{\partial y}{\partial L_2} \frac{\partial}{\partial y} \tag{33}$$

or in matrix notation:

$$\tilde{\nabla} = \underline{J}\nabla$$

$$\begin{pmatrix} \frac{\partial}{\partial L_1} \\ \frac{\partial}{\partial L_2} \end{pmatrix} = \begin{pmatrix} \frac{\partial x}{\partial L_1} & \frac{\partial y}{\partial L_1} \\ \frac{\partial x}{\partial L_2} & \frac{\partial y}{\partial L_2} \end{pmatrix} \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix}, \tag{34}$$

where $\underline{J}$ represents the Jacobian matrix, ∇ the partial derivatives in Cartesian coordinates and $\tilde{\nabla}$ the partial derivatives in triangular coordinates. To get the derivatives in Cartesian form the upper equation must be multiplied with the inverse Jacobian matrix $\underline{J}^{-1}$:

$$\underline{J}^{-1} = \frac{1}{|\underline{J}|} \begin{pmatrix} \frac{\partial y}{\partial L_2} & -\frac{\partial y}{\partial L_1} \\ \frac{-\partial x}{\partial L_2} & \frac{\partial x}{\partial L_1} \end{pmatrix} \tag{35}$$

where $|\underline{J}|$ denotes the determinant of the Jacobian matrix. The conversion between triangular and Cartesian coordinates can be summarized as follows (see Figure 4 and [Ste15]):

$$L_1 + L_2 + L_3 = 1 \rightarrow L_3 = 1 - L_1 - L_2$$
$$x = x_1 L_1 + x_2 L_2 + x_3 L_3 = (x_1 - x_3)L_1 + (x_2 - x_3)L2 + x_3 \tag{36}$$
$$y = y_1 L_1 + y_2 L_2 + y_3 L_3 = (y_1 - y_3)L_1 + (y_2 - y_3)L2 + y_3$$

Considering equation (36) the Jacobian matrix can now be calculated:

$$J = \begin{pmatrix} \frac{\partial x}{\partial L_1} = x_1 - x_3 = x_{13} & \frac{\partial y}{\partial L_1} = y_1 - y_3 = y_{13} \\ \frac{\partial x}{\partial L_2} = x_2 - x_3 = x_{23} & \frac{\partial y}{\partial L_2} = y_2 - y_3 = y_{23} \end{pmatrix} = \begin{pmatrix} x_{13} & y_{13} \\ x_{23} & y_{23} \end{pmatrix} \tag{37}$$

and hence the inverse Jacobian matrix:

$$\underline{J}^{-1} = \frac{1}{2A_\triangle} \begin{pmatrix} y_{23} & -y_{13} \\ -x_{23} & x_{13} \end{pmatrix} \tag{38}$$

The determinant of the Jacobian matrix is two times the area of the triangle. With the help of equation (38), (34) can be reorganized

$$\nabla = \underline{J}^{-1}\tilde{\nabla} \tag{39}$$

and this finally yields to the new version of the differential operator $\underline{L}$ [Ste15]:

$$\underline{L} = \frac{1}{2A_\triangle} \begin{pmatrix} y_{23}\frac{\partial}{\partial L_1} - y_{13}\frac{\partial}{\partial L_2} & 0 \\ 0 & -x_{23}\frac{\partial}{\partial L_1} + x_{13}\frac{\partial}{\partial L_2} \\ -x_{23}\frac{\partial}{\partial L_1} + x_{13}\frac{\partial}{\partial L_2} & y_{23}\frac{\partial}{\partial L_1} - y_{13}\frac{\partial}{\partial L_2} \end{pmatrix} \tag{40}$$

Next, the strain-displacement matrix $\underline{B}$ can be calculated:

$$\underline{B} = \underline{L}\,\underline{N}$$

$$= \frac{1}{2A_\triangle}\begin{pmatrix} y_{23}\frac{\partial}{\partial L_1} - y_{13}\frac{\partial}{\partial L_2} & 0 \\ 0 & -x_{23}\frac{\partial}{\partial L_1} + x_{13}\frac{\partial}{\partial L_2} \\ -x_{23}\frac{\partial}{\partial L_1} + x_{13}\frac{\partial}{\partial L_2} & y_{23}\frac{\partial}{\partial L_1} - y_{13}\frac{\partial}{\partial L_2} \end{pmatrix}$$

$$\begin{pmatrix} L_1 & 0 & L_2 & 0 & 1-L_1-L_2 & 0 \\ 0 & L_1 & 0 & L_2 & 0 & 1-L_1-L_2 \end{pmatrix}$$

$$= \frac{1}{2A_\triangle}\begin{pmatrix} y_{23} & 0 & -y_{13} & 0 & y_{12} & 0 \\ 0 & -x_{23} & 0 & x_{13} & 0 & -x_{12} \\ -x_{23} & y_{23} & x_{13} & -y_{13} & -x_{12} & y_{12} \end{pmatrix} \tag{41}$$

With $\underline{B}$ known, one can insert equation (32) into (21) to get the stresses:

$$\vec{\sigma} = \underline{D}\,\underline{B}\vec{u} \tag{42}$$

Finally, every term of the plane element's functional (24) can be filled with the above discretized terms:

$$\begin{aligned}
\Pi &= \frac{1}{2}\int_V \vec{\varepsilon}^T\vec{\sigma}\,dV - \vec{u}^T\vec{F} \\
&= \frac{1}{2}\int_V \vec{u}^T\underline{B}^T\underline{D}\,\underline{B}\vec{u}\,dV - \vec{u}^T\vec{F} \\
&= \frac{1}{2}\vec{u}^T\int_V \underline{B}^T\underline{D}\,\underline{B}\,dV\vec{u} - \vec{u}^T\vec{F} \\
&= \frac{1}{2}\vec{u}^T\underline{K}\vec{u} - \vec{u}^T\vec{F}
\end{aligned} \tag{43}$$

with $\underline{K}$ the stiffness matrix and $\vec{F}$ the nodal force vector.

The variation of the functional (43) is as follows [Ste15]:

$$\begin{aligned}
\delta\Pi &= \frac{\partial\Pi}{\partial\vec{u}}\delta\vec{u} = 0 \\
&= \frac{1}{2}\delta\vec{u}^T\frac{\partial\vec{u}^T}{\partial\vec{u}^T}\underline{K}\vec{u} + \frac{1}{2}\vec{u}^T\underline{K}\frac{\partial\vec{u}}{\partial\vec{u}}\delta\vec{u} - \delta\vec{u}\frac{\partial\vec{u}^T}{\partial\vec{u}^T}\vec{F} \\
&= \delta\vec{u}^T\left(\underline{K}\vec{u} - \vec{F}\right) = 0
\end{aligned} \tag{44}$$

In order to satisfy this equation, the term in between the parenthesis must be zero ($\delta\vec{u}^T$ can have arbitrary values). This leads to the equilibrium equation of the triangular plane element as described in [Ste15]:

$$\underline{K}\vec{u} = \vec{F} \tag{45}$$

Since the thickness t of the element is constant per definition, it is $dV = t\,dA$ and therefore the integral of the stiffness matrix changes to:

$$\underline{K} = t\int_A \underline{B}^T\underline{D}\,\underline{B}\,dA = tA_\triangle\underline{B}^T\underline{D}\,\underline{B} \tag{46}$$

3.2.3 Quad-4 Plane Element

It is sometimes beneficial to use quadrilateral elements when describing certain detailed parts of a mesh or just to use less elements to describe a plane area. In contrast to triangles which always lie, due to their simple shape, in a plane, quadrilateral can have more complex forms. Such cases include for example: The fourth node does not lie in the plane defined by the other three or the shape is not convex. It is difficult to deal with such forms and one could be tempted to restrict the element to have rectangular shapes only, because these are easy to formulate and work with. But they are impractical when complicated geometry is to be modeled, especially if details should be emphasized in fine graduation.

One solution to this problem is the use of isoparametric elements. They can be non-rectangular. The trick is to use reference coordinates which map the physical element into a reference element that is a square. Thus, the physical element can have a more general shape, but a coordinate transformation and numerical integration is needed which brings in more mathematical complexity [CMPW02].

In this section a quadrilateral isoparametric elements consisting of four nodes is described and denoted by **Quad-4**. One can expand it to eight- or nine-node elements. Figure 5 shows the two abstraction layers: On the left side the original element is shown in physical space, on the right side the reference element is shown. The square has a side length of 2. The coordinate system with the ξ and η axis has its origin in the center of the square. Also note the numbering of the nodes is again counter-clockwise.
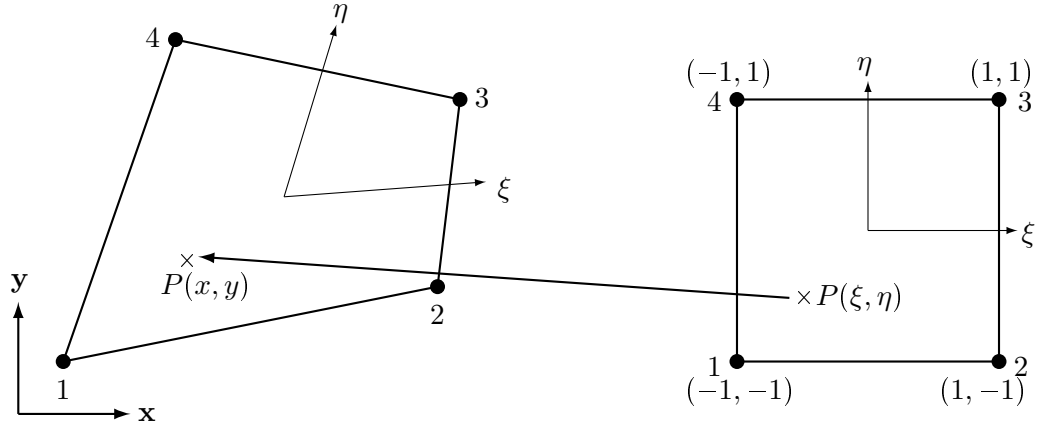
Figure 5: Coordinate transformation of four node quadrilateral element with physical space on the left side and reference space on the right side

Similarly to the triangular element, interpolating the displacement field as well as the element geometry is done by shape functions. They are defined in reference coordinates. The displacement of a point within the element can be expressed by the displacements at the nodes and shape functions $\underline{N}$. Also, the position of that point can be expressed in terms of the (global) nodal positions and shape functions $\underline{\tilde{N}}$. The element is called

isoparametric if $\underline{N}$ is identical to $\underline{\tilde{N}}$. If $\underline{\tilde{N}}$ is of lower degree than $\underline{N}$, the element is called *subparametric* and *superparametric* if it is the other way around [CMPW02].

Every node has two degrees of freedom: A displacement u along the x-axis and a displacement v along the y-axis. To find the shape functions it does not matter which variable to choose, so the following basis function was used for ϕ which can either represent u or v [Ste15]:

$$\phi(\xi, \eta) = a_0 + a_1\xi + a_2\eta + a_3\xi\eta = \begin{pmatrix} 1 & \xi & \eta & \xi\eta \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \vec{x}^T \vec{a} \qquad (47)$$

The interpolation conditions at the nodes are as follows:

$$\begin{aligned}
\phi(-1, -1) = \phi_1 &\rightarrow \phi_1 = a_0 - a_1 - a_2 + a_3 \\
\phi(1, -1) = \phi_2 &\rightarrow \phi_2 = a_0 + a_1 - a_2 - a_3 \\
\phi(1, 1) = \phi_3 &\rightarrow \phi_3 = a_0 + a_1 + a_2 + a_3 \\
\phi(-1, 1) = \phi_4 &\rightarrow \phi_4 = a_0 - a_1 + a_2 - a_3
\end{aligned} \qquad (48)$$

or in matrix notation:

$$\underline{A}\vec{a} = \vec{\phi}$$

$$\begin{pmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{pmatrix} \qquad (49)$$

Inversion of $\underline{A}$ yields the coefficients a_i:

$$\vec{a} = \underline{A}^{-1}\vec{\phi} = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{pmatrix} \qquad (50)$$

If the last equation is inserted into (47) one gets the shape functions $\vec{N}$ for the quadrilateral element:

$$\begin{aligned}
\phi &= \vec{x}^T \underline{A}^{-1} \vec{\phi} \\
&= \vec{N}^T \vec{\phi} \\
&= \frac{1}{4} \begin{pmatrix} 1 & \xi & \eta & \xi\eta \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix} \vec{\phi} \\
&= \begin{pmatrix} \frac{1}{4}(1-\xi)(1-\eta) & \frac{1}{4}(1+\xi)(1-\eta) & \frac{1}{4}(1+\xi)(1+\eta) & \frac{1}{4}(1-\xi)(1+\eta) \end{pmatrix} \vec{\phi} \qquad (51)
\end{aligned}$$

One can evaluate shape function i with the $\xi\eta$-coordinates of node i. If it evaluates to 1 while at any other node coordinates it evaluates to zero the shape function is correctly set. Now, the displacements can be expressed as follows:

$$\vec{u} = \begin{pmatrix} u \\ v \end{pmatrix} = \underline{N}\vec{u} = \begin{pmatrix} N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 & 0 \\ 0 & N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \end{pmatrix} \tag{52}$$

with $\underline{N}$ being the matrix containing the shape functions and $\vec{u}$ being the vector of the nodal displacements.

The assembly of the strain-displacement matrix $\underline{B}$ is more complicated with isoparametric elements. Due to the $\xi\eta$-coordinates one cannot easily describe an operator such as $\partial/\partial x$. The first step is to formulate a function $\phi = \phi(\xi, \eta)$. Like in the derivation on the shape functions ϕ can represent u or v. Derivatives with respect to ξ and η are as follows [CMPW02]:

$$\frac{\partial \phi}{\partial \xi} = \frac{\partial \phi}{\partial x}\frac{\partial x}{\partial \xi} + \frac{\partial \phi}{\partial y}\frac{\partial y}{\partial \xi}$$
$$\frac{\partial \phi}{\partial \eta} = \frac{\partial \phi}{\partial x}\frac{\partial x}{\partial \eta} + \frac{\partial \phi}{\partial y}\frac{\partial y}{\partial \eta} \tag{53}$$

or in matrix notation:
$$\vec{\phi} = \underline{J}\vec{\phi} \tag{54}$$

where $\underline{J}$ is the Jacobian matrix

$$\underline{J} = \begin{pmatrix} \sum N_{i,\xi} x_i & \sum N_{i,\xi} y_i \\ \sum N_{i,\eta} x_i & \sum N_{i,\eta} y_i \end{pmatrix} \tag{55}$$

and $N_{i,j}$ denotes the derivation of the i-th shape function with respect to j and x_i the i-th component of the $\vec{x}$ vector. The Jacobian matrix can be written out as follows:

$$\underline{J} = \frac{1}{4}\begin{pmatrix} -(1-\eta) & (1-\eta) & (1+\eta) & -(1+\eta) \\ -(1-\xi) & -(1+\xi) & (1+\xi) & (1-\xi) \end{pmatrix} \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{pmatrix}$$
$$= \begin{pmatrix} (x_{12} + x_{34})\eta - x_{12} + x_{34} & (y_{12} + y_{34})\eta - x_{12} + y_{34} \\ (x_{12} + x_{34})\xi - x_{13} - x_{24} & (y_{12} + y_{34})\xi - y_{13} + y_{24} \end{pmatrix} \tag{56}$$

Next, equation (54) can be rearranged to get the derivatives with respect to x and y:

$$\vec{\phi} = \underline{J}^{-1}\vec{\phi} \tag{57}$$

With the derivatives calculated, the strain-displacement relation (6) can be obtained [CMPW02]:

$$\vec{\varepsilon} = \begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}}_{\underline{L}} \begin{pmatrix} \partial u/\partial x \\ \partial u/\partial y \\ \partial v/\partial x \\ \partial v/\partial y \end{pmatrix} \tag{58}$$

$$\begin{pmatrix} \partial u/\partial x \\ \partial u/\partial y \\ \partial v/\partial x \\ \partial v/\partial y \end{pmatrix} = \underbrace{\begin{pmatrix} j_{11} & j_{12} & 0 & 0 \\ j_{21} & j_{22} & 0 & 0 \\ 0 & 0 & j_{11} & j_{12} \\ 0 & 0 & j_{21} & j_{22} \end{pmatrix}}_{\underline{\hat{J}}} \begin{pmatrix} \partial u/\partial \xi \\ \partial u/\partial \eta \\ \partial v/\partial \xi \\ \partial v/\partial \eta \end{pmatrix} \tag{59}$$

$$\begin{pmatrix} \partial u/\partial \xi \\ \partial u/\partial \eta \\ \partial v/\partial \xi \\ \partial v/\partial \eta \end{pmatrix} = \underbrace{\begin{pmatrix} N_{1,\xi} & 0N_{2,\xi} & 0 & N_{3,\xi} & 0 & N_{4,\xi} & 0 \\ N_{1,\eta} & 0N_{2,\eta} & 0 & N_{3,\eta} & 0 & N_{4,\eta} & 0 \\ 0 & N_{1,\xi} & 0N_{2,\xi} & 0 & N_{3,\xi} & 0 & N_{4,\xi} \\ 0 & N_{1,\eta} & 0N_{2,\eta} & 0 & N_{3,\eta} & 0 & N_{4,\eta} \end{pmatrix}}_{\underline{\hat{N}}} \begin{pmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \end{pmatrix} \tag{60}$$

where j_i denotes the i-th entry in the inverse Jacobian matrix. The composition of the previous three equations forms the matrix $\underline{B}$:

$$\underline{B} = \underline{L}\,\underline{\hat{J}}\,\underline{\hat{N}} \tag{61}$$

Together with the functional equation (43) and the material matrix $\underline{D}$ (eq. (21)), the stiffness matrix for the quadrilateral isoparametric element can be written as:

$$\underline{K} = \int_V \underline{B}^T \underline{D}\,\underline{B}\,dV = t\int_A \underline{B}^T \underline{D}\,\underline{B}\,dA = t\int_{-1}^{1}\int_{-1}^{1} \underline{B}^T \underline{D}\,\underline{B}\,|J|d\xi d\eta \tag{62}$$

For the Quad-4 element a Gaussian quadrature needs four Gauss integration points to satisfy the above equation [Ste15]. These four points are located at $\xi_i = \pm\frac{\sqrt{3}}{3}$ and $\eta_i = \pm\frac{\sqrt{3}}{3}$ with weight factors $\omega_i = 1$. The equation for the stiffness matrix can then be written in discretized form as follows:

$$\underline{K} = t\sum_{i=1}^{2}\sum_{j=1}^{2} \omega_i\omega_j\underline{B}(\xi_i,\eta_j)^T\underline{D}\,\underline{B}(\xi_i,\eta_j)|\underline{J}(\xi_i,\eta_j)| \tag{63}$$

3.3 Plate Bending Element

In contrast to the plane element, the load at the plate (bending) elements are applied transversal to the element's mid-surface producing plate bending. It has one deformation degree of freedom and two rotational degrees of freedom. Plate elements are often used to model floors or ceilings. In this section, the plate element is discussed in more detail and two discretizations are described: The three-node triangular plate element and the four-node quadrilateral plate element.

3.3.1 Problem Definition
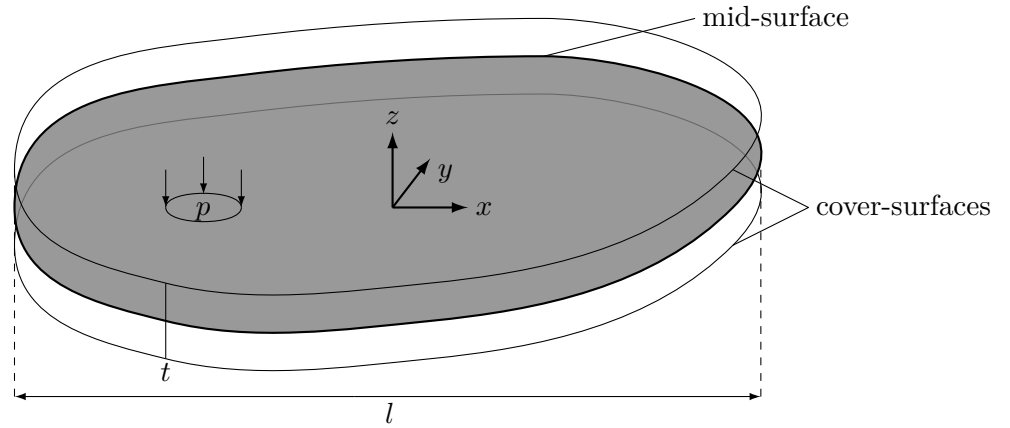
Figure 6: Schematic drawing of a Kirchhoff plate with its main dimension l, thickness t and loading p normal to the mid-surface

In contrast to a plane, where the load is located planar with respect to the plane, the load is perpendicular to the mid place at a plate. Therefore plate element problems are important for supporting structures of bridges or ceilings and floors in buildings, for example. In Figure 6 one can see a generalized plate object. It has a main dimension of l and a constant thickness t. With the assumption that $t \ll l$, the problem becomes two dimensional and, instead of the whole object, only the middle plane between the two surface areas will be considered. The object has a local coordinate system with its xy-plane the mid plane and its z-axis perpendicular to this plane. The surface areas are located at $z = \pm t/2$. As stated in the beginning, the load is applied in z-direction, i.e. normal to the mid-surface.

In this work Kirchhoff's theory of thin plates is used. For thick plates or laminated plates, the theory of Reissner-Mindlin is more applicable [Wer95]. The main difference is that with Reissner-Mindlin plates one takes the shear deformations into account. Thus, the normal to the mid-surface remains straight but not necessarily perpendicular to it; instead of a Kirchhoff plate: Here, the normal remains normal to the mid-surface even after deformation.

As a short summarize the following conditions must be satisfied for a Kirchhoff plate [Ste15]:

- The thickness t must be much smaller than the main dimension l: $t \ll l$.

- Straight lines normal to the mid-surface remain straight after deformation.

- Straight lines normal to the mid-surface remain normal to the mid-surface after deformation.

- There is only a small amount of deformation w, i.e. $w < t$ and it holds $w \neq w(z)$.

- The plate is symmetrical to the mid-surface and changes in thickness must be very small.

- Normal stresses in z-direction σ_{zz} will be neglected.

With [Kle13] and [Ste15] the following displacement terms can be formulated:

$$w = w(x, y) \tag{64}$$

$$u = -z\frac{\partial w}{\partial x} \tag{65}$$

$$v = -z\frac{\partial w}{\partial y} \tag{66}$$

The deformation w suffices to explain the whole displacement vector. The two derivatives in the equations above describe the torsions around the x- and y-axis.

Similar to the plane element, the Kirchhoff plate element can have a plane strain or plane stress, respectively [Ste15], i.e. equation (6) can be applied here, too:

$$\vec{u} = \begin{pmatrix} u \\ v \end{pmatrix} = -z\begin{pmatrix} \frac{\partial w}{\partial x} \\ \frac{\partial w}{\partial y} \end{pmatrix} = -z\nabla w$$

$$\vec{\varepsilon} = \underline{L}\vec{u} = -z\underline{L}\nabla w = -z\vec{\Delta}w = -z\vec{\kappa} \tag{67}$$

$$\vec{\Delta} = \underline{L}\nabla = \begin{pmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{pmatrix} \begin{pmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{pmatrix} = \begin{pmatrix} \frac{\partial^2}{\partial x^2} \\ \frac{\partial^2}{\partial y^2} \\ 2\frac{\partial^2}{\partial x\partial y} \end{pmatrix}$$

$$\vec{\kappa} = \vec{\Delta}w = \begin{pmatrix} \frac{\partial^2 w}{\partial x^2} \\ \frac{\partial^2 w}{\partial y^2} \\ 2\frac{\partial^2 w}{\partial x\partial y} \end{pmatrix} \tag{68}$$

Referring [Kle13] ($\sigma_{zz} = 0, \tau_{xz} = \tau_{yz} = 0$), equation (7) can be filled with the above information:

$$\vec{\sigma} = \underline{D}\vec{\varepsilon} = -z\underline{D}\vec{\kappa}$$

$$= -\frac{Ez}{1 - \nu^2}\begin{pmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{pmatrix}\begin{pmatrix} \kappa_x \\ \kappa_y \\ 2\kappa_{xy} \end{pmatrix} \tag{69}$$

The integration of the stresses $\vec{\sigma}$ over the thickness results in the vector of moments $\vec{M}^T = (M_{xx} \quad M_{yy} \quad M_{xy})$ [Ste15]:

$$\vec{M} = \int_{-t/2}^{t/2} z\vec{\sigma}dz = -\int_{-t/2}^{t/2} z^2 \underline{D}\vec{\kappa}dz = -\underline{D}\vec{\kappa} \int_{-t/2}^{t/2} z^2 dz = -\frac{t^3}{12}\underline{D}\vec{\kappa} = -\underline{D}_p\vec{\kappa} \qquad (70)$$

The above equation relates the moments with the curvatures of the plate. The integrals over the transverse stresses σ_{xz} and σ_{yz} lead to the following shear forces, as described in [Ste15]:

$$Q_x = \int_{-t/2}^{t/2} \sigma_{xz}dz = \int_{-t/2}^{t/2} \sigma_{xz}^{\max}\left(1 - 4\left(\frac{z}{t}\right)^2\right)dz = \frac{2}{3}\sigma_{xz}^{\max}t$$
$$= \frac{2}{3}\sigma_{xz}(z = 0)t \qquad (71)$$

$$Q_y = \int_{-t/2}^{t/2} \sigma_{yz}dz = \int_{-t/2}^{t/2} \sigma_{yz}^{\max}\left(1 - 4\left(\frac{z}{t}\right)^2\right)dz = \frac{2}{3}\sigma_{yz}^{\max}t$$
$$= \frac{2}{3}\sigma_{yz}(z = 0)t \qquad (72)$$

The transverse stress is distributed quadratically over the thickness t, i.e. they have their maximum at $z = 0$ and vanish at $z = \pm t/2$. The equilibrium of forces in z-direction leads to:
$$\frac{\partial Q_x}{\partial x} + \frac{\partial Q_y}{\partial y} + p = 0 \qquad (73)$$

with p the load applied perpendicular to the mid-surface. Additionally the equilibrium of moments around the x- and y-axis:

$$\frac{\partial M_{xx}}{\partial x} + \frac{\partial M_{xy}}{\partial y} + Q_x = 0$$
$$\frac{\partial M_{yy}}{\partial y} + \frac{\partial M_{xy}}{\partial x} + Q_y = 0 \qquad (74)$$

Putting equation (74) into (73) results in:

$$\frac{\partial^2 M_{xx}}{\partial x^2} + \frac{\partial^2 M_{yy}}{\partial y^2} + 2\frac{\partial^2 M_{xy}}{\partial x \partial y} = \vec{\Delta}^T \vec{M} = p \qquad (75)$$

Now, one can insert the kinematic equation (68) into equation (70) and then into the equilibrium relation (75):

$$\vec{\kappa} = \vec{\Delta}w$$
$$\vec{M} = -\underline{D}_p\vec{\kappa} = -\underline{D}_p\vec{\Delta}w$$
$$\vec{\Delta}^T\vec{M} = -\vec{\Delta}^T\underline{D}_p\vec{\Delta}w = p \qquad (76)$$

The last equation leads to the partial differential equation of the plate bending [Kle13]:

$$\frac{\partial^4 w}{\partial x^4} + \frac{\partial^4 w}{\partial y^4} + 2\frac{\partial^4 w}{\partial x^2 \partial y^2} = -\frac{12(1 - \nu^2)}{Et^3}p = \frac{p}{k} \tag{77}$$

with k denoted as *plate stiffness*.
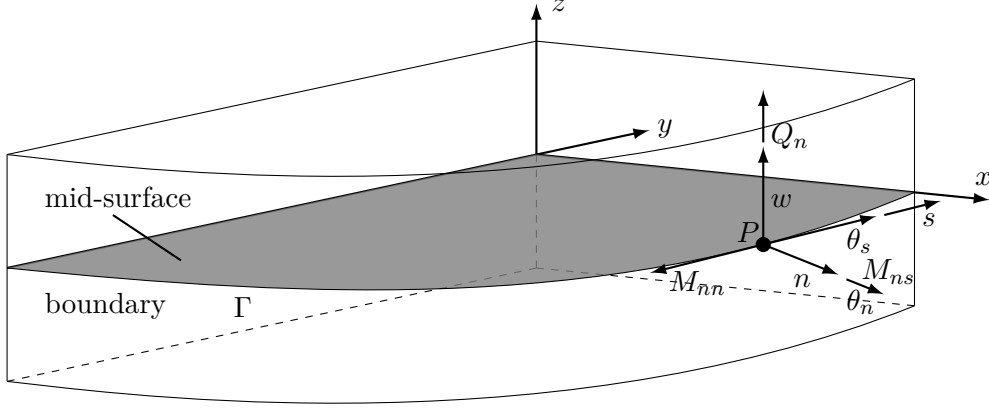
Figure 7: Part of a plate's boundary with its essential and natural boundary conditions

Let P be a point on the continuous boundary of the plate with a local Cartesian coordinate system as described in [Ste15] (see Figure 7): The n coordinate is perpendicular to the boundary surface, the s axis tangential to it. The third axis equals the global z-axis of the plate. There are three essential and three natural boundary conditions defined for P: The displacement w, the drills $\theta_n = \partial w / \partial s, \theta_s = -\partial w / \partial n$, the shear force Q_n and the moments M_{ns} and M_{nn}. Since this leads to an inconsistency with the differential equation above, [Ste15] stated that Kirchhoff introduced new forces:

$$V_n = Q_n - \frac{\partial M_{ns}}{\partial s} \tag{78}$$

With them, only the four conditions for w, θ_s, V_n and M_{nn} occur. The plate can be mounted in different ways:

- clamped: $w = 0, \theta_s = -\partial w / \partial n = 0$

- simple supported: $w = 0, M_{nn} = 0$

- symmetrical edge: $\theta_s = -\partial w / \partial n = 0, V_n = 0$

The plate's functional as described in [Ste15] is given below:

$$\frac{1}{2}\int_V \vec{\varepsilon}^T \vec{\sigma}\, dV \tag{79}$$

One can insert equation (67) and (69) into the functional:

$$\frac{1}{2}\int_V \vec{\varepsilon}^T \vec{\sigma}\, dV = \frac{1}{2}\int_V \vec{\kappa}^T \underline{D}\vec{\kappa} z^2\, dV = \frac{1}{2}\int_A \vec{\kappa}^T \underline{D}_p \vec{\kappa}\, dA \tag{80}$$

Together with the potential of the external forces the overall potential of the Kirchhoff plate is:

$$\Pi = \frac{1}{2} \int_A \vec{\kappa}^T \underline{D}_p \vec{\kappa} \, dA - \int_A p \, w \, dA - \int_\Gamma (V_n w - M_{nn} \theta_s) \, d\Gamma \qquad (81)$$

Klein [Kle13] states that for the plate element discretization additional conditions must be satisfied. They are: The bending $w(x, y)$ as well as the normal derivative $\partial w / \partial n$ at the element's boundary must be continuous to the neighboring elements. This would be the case if the bending and the normal derivative are explicitly determined by the nodal parameters at the border. Further, Klein lists requirements for a plate element ansatz:

- Totality of the displacement approach in order to guarantee good convergence.

- The terms $1, x, y, x^2, xy, y^2$ should be included to get variable strains, curvatures and rigid body motion.

Steinke [Ste15] expands the requirements as follows:

- Compatibility of the displacement variable at the element's boundary (conformity condition): If the steadiness of the deformation w and its first derivatives is not satisfied the bending surface between two elements can have a sharp bend at which the elements are overlapping at one side and diverge on the opposite side. If such a behavior is shown, the element is called *non-conforming.*

- Rigid body motions must not create strains and stresses in the element. This requires a constant term in the basis function for the translative part of the motion and a linear term for the rotatory.

- The basis function must provide constant plain strain and plain stress: If the element converges in its size until it becomes a point, a constant state of bending must be describable in this situation. Since the bending is described as second order derivatives of w, the basis function must include quadratic terms.

The following sections show details of two discretizations of plate elements: A triangular element with three nodes and a quadrilateral element with four nodes.

3.3.2 Tri-3 Plate Element

There exist many different types of triangular plate elements, for example Batoz et al. [BBH80], Tocher [Toc63] or Specht [Spe88]. The three node triangular element from [Toc63] has three degrees of freedom (d.o.f) (w, θ_x, θ_y) per node. His basis function was a complete cubic polynomial. The term xy was left out, because the polynomial has one coefficient more than the element has d.o.f. This leads to the problem that no constant state of bending can be described (non-conforming element) and this leads to wrong results at convergence [Ste15]. Therefore, Steinke challenges the practical use of this element. A possible way to use a complete cubic polynomial would be to add another node in the center of mass of the triangle and assign the only degree of freedom w to it [Ste15].

But the problem of non-conformity persist, as the nodal twists don't suffice to describe the twists along the element's edges, which are quadric. Here, additional nodes on the edges would be needed. To get a conforming element one can choose a basis function with a complete polynomial of fifth order. It has 21 coefficients and d.o.f. They are distributed as follows: Every node has six d.o.f $(w, \partial w/\partial x, \partial w/\partial y, \partial^2 w/\partial x^2, \partial^2 w/\partial y^2, \partial^2 w/\partial x \partial y)$ and the mid node of every edge gets the degree of freedom $\partial w/\partial n$. A conform element with continuous twists at its edges follows from that. But the 21 d.o.f. per element leads to high computational effort and second order derivatives at the boundaries are needed. Hence, Steinke advices against using it in practice [Ste15].

In this work an element from Specht [Spe88] was implemented which is also described in [Ste15]. It has three nodes and also three d.o.f. per node: The deformation w and the two twists θ_x and θ_y. The basis function for the deformation w is as follows:

$$
\begin{aligned}
w = & a_0 L_1 + a_1 L_2 + a_2 L_3 + a_3 L_1 L_2 + a_4 L_2 L_3 + a_5 L_3 L_1 \\
& + a_6 \left(L_2 L_1^2 + \frac{1}{2} L_1 L_2 L_3 \left(3(1 - \mu_3)L_1 - (1 + 3\mu_3)L_2 + (1 + 3\mu_3)L_3 \right) \right) \\
& + a_7 \left(L_3 L_2^2 + \frac{1}{2} L_1 L_2 L_3 \left(3(1 - \mu_1)L_2 - (1 + 3\mu_1)L_3 + (1 + 3\mu_1)L_1 \right) \right) \\
& + a_8 \left(L_1 L_3^2 + \frac{1}{2} L_1 L_2 L_3 \left(3(1 - \mu_2)L_3 - (1 + 3\mu_2)L_1 + (1 + 3\mu_2)L_2 \right) \right)
\end{aligned}
\tag{82}
$$

with

$$
\begin{aligned}
\mu_1 &= \frac{S_{21} - S_{31}}{S_{32}} \\
\mu_2 &= \frac{S_{32} - S_{21}}{S_{31}} \\
\mu_3 &= \frac{S_{31} - S_{32}}{S_{21}} \\
S_{32} &= x_{32}^2 + y_{32}^2 \\
S_{31} &= x_{31}^2 + y_{31}^2 \\
S_{21} &= x_{21}^2 + y_{21}^2
\end{aligned}
\tag{83}
$$

$$
\tag{84}
$$

S_{ij} denoted the square of the length of the edge between node i and j. This can be

written is vector form:

$$w = \vec{x}^T \vec{a}$$

$$\vec{x} = \begin{pmatrix} L_1 \\ L_2 \\ L_3 \\ L_1 L_2 \\ L_2 L_3 \\ L_3 L_1 \\ \left(L_2 L_1^2 + \frac{1}{2} L_1 L_2 L_3 \left(3(1 - \mu_3)L_1 - (1 + 3\mu_3)L_2 + (1 + 3\mu_3)L_3\right)\right) \\ \left(L_3 L_2^2 + \frac{1}{2} L_1 L_2 L_3 \left(3(1 - \mu_1)L_2 - (1 + 3\mu_1)L_3 + (1 + 3\mu_1)L_1\right)\right) \\ \left(L_1 L_3^2 + \frac{1}{2} L_1 L_2 L_3 \left(3(1 - \mu_2)L_3 - (1 + 3\mu_2)L_1 + (1 + 3\mu_2)L_2\right)\right) \end{pmatrix}$$

$$\vec{a}^T = \begin{pmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \end{pmatrix} \tag{85}$$

The twists θ_x and θ_y are to be described in Cartesian coordinates. They must be transformed into triangular coordinates with the help of equation (34):

$$\vec{\theta} = \begin{pmatrix} \theta_x \\ \theta_y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \nabla w = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \underline{J}^{-1} \tilde{\nabla} \vec{x}^T \vec{a} = \underline{G} \vec{a} \tag{86}$$

with $\underline{J}^{-1}$ the inverse Jacobian matrix and $\tilde{\nabla}$ the nabla operator in triangular coordinates. The matrix $\underline{G}$:

$$\underline{G} = \frac{1}{2A_\triangle} \begin{pmatrix} x_{32} & x_{13} \\ y_{32} & y_{13} \end{pmatrix} \begin{pmatrix} \frac{\partial x_1}{\partial L_1} & \frac{\partial x_2}{\partial L_1} & \cdots & \frac{\partial x_9}{\partial L_1} \\ \frac{\partial x_1}{\partial L_2} & \frac{\partial x_2}{\partial L_2} & \cdots & \frac{\partial x_9}{\partial L_2} \end{pmatrix} \tag{87}$$

Next, the interpolation conditions at the three nodes for the three unknowns can be set (cf. Figure 4). Following the notation of [Ste15]:

$$\underbrace{\begin{pmatrix} \vec{x}^T(1,0) \\ G_1(1,0) \\ G_2(1,0) \\ \vec{x}^T(0,1) \\ G_1(0,1) \\ G_2(0,1) \\ \vec{x}^T(0,0) \\ G_1(0,0) \\ G_2(0,0) \end{pmatrix}}_{\underline{A}} \vec{a} = \underbrace{\begin{pmatrix} w_1 \\ \theta_{x_1} \\ \theta_{y_1} \\ w_2 \\ \theta_{x_2} \\ \theta_{y_2} \\ w_3 \\ \theta_{x_3} \\ \theta_{y_3} \end{pmatrix}}_{\vec{w}} \tag{88}$$

$$\tag{89}$$

where $\underline{G}_i$ is the i-th row of matrix $\underline{G}$. The unknown coefficients a_i can be computed by

32

inverting the matrix $\underline{A}$: $\vec{a} = \underline{A}^{-1}\vec{w}$:

$$
\underline{A}^{-1} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
-1 & 0 & 0 & 1 & y_{12} & x_{21} & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 1 & y_{23} & x_{32} \\
1 & y_{31} & x_{13} & 0 & 0 & 0 & -1 & 0 & 0 \\
2 & y_{21} & x_{12} & -2 & y_{21} & x_{12} & 0 & 0 & 0 \\
0 & 0 & 0 & 2 & y_{32} & x_{23} & -2 & y_{32} & x_{23} \\
-2 & y_{13} & x_{31} & 0 & 0 & 0 & 2 & y_{13} & x_{31}
\end{pmatrix} \tag{90}
$$

where x_{ij} and y_{ij} denotes the differences of the node's coordinates $x_i - x_j$ and $y_i - y_j$. Now, the coefficients can be inserted into equation (85):

$$
w = \vec{x}^T \vec{a} = \vec{x}^T \underline{A}^{-1} \vec{w} = \vec{N}^T \vec{w} \tag{91}
$$

The vector $\vec{N}$ containing the shape functions N_i can then be calculated as follows:

$$
\vec{N} = \left(\underline{A}^{-1}\right)^T \vec{x} \tag{92}
$$

Since the shape functions follow a pattern, due to the regular order in the matrix $\underline{A}^{-1}$, one can summarize the nine shape functions into three groups; one for every node:

$$
N_i = \begin{cases}
\chi_i - \chi_{i+3} + \chi_{k+3} + 2\left(\chi_{i+6} - \chi_{k+6}\right) & \text{for d.o.f. } w \\
-y_{ki}\left(\chi_{k+6} - \chi_{k+3}\right) + y_{ji}\chi_{i+6} & \text{for d.o.f. } \theta_x \\
x_{ki}\left(\chi_{k+6} - \chi_{k+3}\right) - x_{ji}\chi_{i+6} & \text{for d.o.f. } \theta_y
\end{cases} \tag{93}
$$

The variables χ_i denotes the i-th component of the vector $\vec{x}$, the indexes i, j, k under χ are cyclic permutations of $1, 2, 3$. x_{ij} and y_{ij} denote the coordinate differences $x_i - x_j$ and $y_i - y_j$. The index under N is incremented in such a way, that N_1, N_4, N_7 describes the d.o.f. w, N_2, N_5, N_8 describes the d.o.f. θ_x and N_3, N_6, N_9 describes the d.o.f. θ_y. Similar to the plane elements, one can check the correctness of the shape functions by evaluating them at the triangular coordinates of the three triangle's nodes. For example, shape function N_7 will evaluate to 1 for the coordinates $(L_1 = 0, L_2 = 0)$ (node 3) and will be zero for $(L_1 = 1, L_2 = 0)$ (node 1) and $(L_1 = 0, L_2 = 1)$ (node 2).

The displacement-strain relation (67) introduced for the plate element contains an operator living in the Cartesian space. It has to be converted into triangular coordinates.

With equation (39) ($\nabla = \underline{J}^{-1}\tilde{\nabla}$) one can describe a second order derivative operator Δ:

$$\Delta = \nabla\nabla^T = \underline{J}^{-1}\tilde{\nabla}\left(\underline{J}^{-1}\tilde{\nabla}\right)^T = \underline{J}^{-1}\tilde{\nabla}\tilde{\nabla}^T\left(\underline{J}^{-1}\right)^T = \underline{J}^{-1}\tilde{\Delta}\left(\underline{J}^{-1}\right)^T \tag{94}$$

$$\Delta = \begin{pmatrix} \frac{\partial^2}{\partial x^2} & \frac{\partial^2}{\partial x \partial y} \\ \frac{\partial^2}{\partial y \partial x} & \frac{\partial^2}{\partial y^2} \end{pmatrix} \rightarrow \vec{\Delta} = \begin{pmatrix} \frac{\partial^2}{\partial x^2} \\ \frac{\partial^2}{\partial y^2} \\ 2\frac{\partial^2}{\partial x \partial y} \end{pmatrix}$$

$$\vec{\Delta} = \frac{1}{4A_\triangle^2} \begin{pmatrix} y_{32}^2 & y_{31}^2 & y_{23}y_{31} \\ x_{32}^2 & x_{31}^2 & x_{13}x_{32} \\ 2x_{32}y_{23} & 2x_{13}y_{31} & x_{32}y_{31}+x_{31}y_{32} \end{pmatrix} \begin{pmatrix} \frac{\partial^2}{\partial L_1^2} \\ \frac{\partial^2}{\partial L_2^2} \\ 2\frac{\partial^2}{\partial L_1 \partial L_2} \end{pmatrix}$$

$$\vec{\Delta} = \underline{Y}\vec{\tilde{\Delta}} \tag{95}$$

Next, equation (67) can be rewritten for triangular coordinates:

$$\vec{\varepsilon} = -z\vec{\Delta}w = -z\underline{Y}\vec{\tilde{\Delta}}w = -z\vec{\kappa} \tag{96}$$

And additionally, with the help of equation (91), this yields a new version of equation (68):

$$\vec{\kappa} = \vec{\Delta}w = \underline{Y}\vec{\tilde{\Delta}}\vec{N}^T\vec{w} = \underline{Y}\,\underline{\tilde{B}}\vec{w} = \underline{B}\vec{w} \tag{97}$$

$$\underline{\tilde{B}} = \vec{\tilde{\Delta}}\vec{N}^T = \begin{pmatrix} \frac{\partial^2 N_1}{\partial L_1^2} & \frac{\partial^2 N_2}{\partial L_1^2} & \cdots & \frac{\partial^2 N_9}{\partial L_1^2} \\ \frac{\partial^2 N_1}{\partial L_2^2} & \frac{\partial^2 N_2}{\partial L_2^2} & \cdots & \frac{\partial^2 N_9}{\partial L_2^2} \\ 2\frac{\partial^2 N_1}{\partial L_1 \partial L_2} & 2\frac{\partial^2 N_2}{\partial L_1 \partial L_2} & \cdots & 2\frac{\partial^2 N_9}{\partial L_1 \partial L_2} \end{pmatrix} \tag{98}$$

With the help of equation (97), the first term (denoted as Π_1) of the plate element's functional (81) can be written out:

$$\begin{aligned} \Pi_1 &= \frac{1}{2}\int_A \vec{\kappa}^T \underline{D}_p \vec{\kappa} \; dA \\ &= \frac{1}{2}\vec{w}^T \int_A \underline{B}^T \underline{D}_p \underline{B} \; dA\vec{w} \\ &= \frac{1}{2}\vec{w}^T \underline{K}\vec{w} \end{aligned} \tag{99}$$

where $\underline{K}$ describes the stiffness matrix for the three node triangular plate element. The stiffness matrix must be integrated in triangular coordinates. This will be done by a Gaussian quadrature with the Gauss points located at: $(L_{1_1} = 1/6, L_{2_1} = 1/6), (L_{1_2} = 2/3, L_{2_2} = 1/6), (L_{1_3} = 1/6, L_{2_3} = 2/3)$ and weights $\omega_i = 1/6$ for all three points. For an exact integration one would accumulate four sampling points, but [Ste15] states that this leads to an element, that is too stiff; with only three samplings a more natural element results.

$$\underline{K} = 2A_\triangle \sum_{i=1}^{3} \omega_i \underline{B}^T(L_{1_i}, L_{2_i})\underline{D}_p\underline{B}(L_{1_i}, L_{2_i}) \tag{100}$$

The plate's functional (81) has two more terms including the surface load p and edge loads V_n. These two can now be written as follows (see also [Ste15]):

$$\int_A p\, w\, dA = \vec{w}^T \vec{F}_p = \vec{w}^T p \int_A \vec{N}\, dA = 2\vec{w}^T A_\triangle p \int_0^1 \left(\int_0^{1-L_1} \vec{N}\, dL_2 \right)\, dL_1 \qquad (101)$$

where $\vec{F}_p$ is a $1{\times}9$ vector containing forces and moments emerging from the surface load p. As an example, an edge load V_n is applied to edge S_{13}. This can be described as follows [Ste15]:

$$\int_{\Gamma_V} V_n\, w\, d\Gamma = \int_{\Gamma_V} V_n \vec{w}^T \vec{N}(L_2 = 0)\, d\Gamma \qquad (102)$$

with $d\Gamma = S_{13} dL_1$. With V_n being constant all over the edge:

$$\int_{\Gamma_V} V_n \vec{w}^T \vec{N}(L_2 = 0)\, d\Gamma = \vec{w}^T S_{13} V_n \int_0^1 \vec{N}\, dL_1 = \vec{w}^T \vec{F}_v \qquad (103)$$

The edge load applies forces and moments contained in $\vec{F}_v$ to the nodes forming that edge. The above equation can be applied to every other edge.

3.3.3 Quad-4 Plate Element

The Quad-4 element implemented in this work is the so-called Discrete Kirchhoff Quadrilateral (DKQ) element, introduced by Batoz et al. [BT82]. It is a four-node, 12 degrees-of-freedom quadrilateral element for thin plates. It is based on a generalization of the Discrete Kirchhoff Triangular (DKT) element which is a three-node, 9 d.o.f. triangular element. Like the triangular element of the previous section, all the DKQ elements nodes have three degrees of freedom: The displacement w and the rotations θ_x and θ_y around the element's local x- and y-axis. Figure 8 shows an example of such an element.

The formulation of the DKQ element by Batoz et al. [BT82] uses the discrete Kirchhoff technique. It is based on the discretization of the strain energy and neglects the transverse shear energy. This results in the following functional:

$$\Pi = \frac{1}{2} \int_A \vec{\kappa}^T \underline{D}_p \vec{\kappa}\, dA \qquad (104)$$

where $\underline{D}_p$ is the material matrix as defined previously (equation (70)) and $\vec{\kappa}$ denotes:

$$\vec{\kappa} = \begin{pmatrix} \frac{\partial \beta_x}{\partial x} \\ \frac{\partial \beta_y}{\partial y} \\ \frac{\partial \beta_x}{\partial y} + \frac{\partial \beta_y}{\partial x} \end{pmatrix} \qquad (105)$$

β_i is the rotation of the normal to the undeformed mid-surface in x-z-plane and y-z-plane, respectively. For Π only C^0 continuity is required [BT82]. Further, Batoz et al. states that β_x and β_y must be related to w in such a way, that the final element satisfies the following requirements:
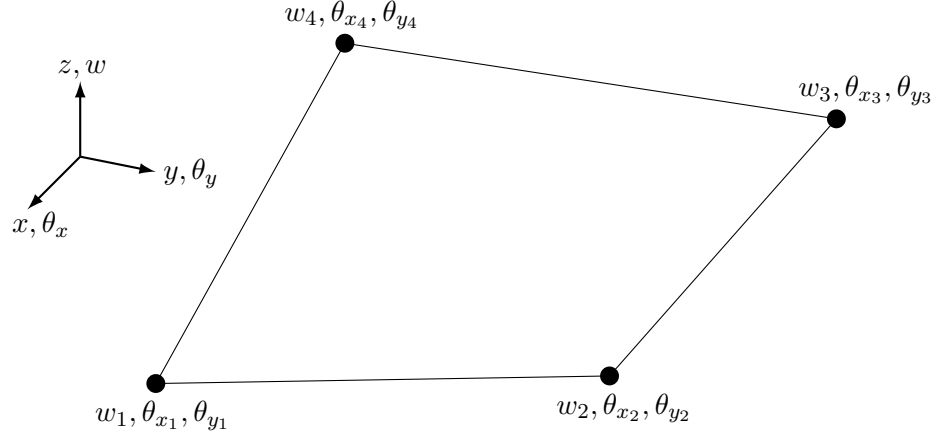
Figure 8: 4-node quadrilateral plate element DKQ with 12 degrees-of-freedom, three per node.

- The nodal variables must be w, θ_x and θ_y with respect to x and y at the four element's nodes ($\theta_x = \partial w/\partial y, \theta_y = -\partial w/\partial x$)

- The Kirchhoff boundary conditions must be satisfied.

Two incomplete cubic polynomial expressions define β_x and β_y:

$$\beta_x = \sum_{i=1}^{8} N_i \beta_{x_i} \tag{106}$$

$$\beta_y = \sum_{i=1}^{8} N_i \beta_{y_i} \tag{107}$$

$N_i(\xi, \eta)$ are here the shape functions with isoparametric coordinates ξ and η. They are the same as of the "eight-node Serendipity" element, described for example in [ZT00], or [Bra07] and seen in Figure 9. The shape functions of this element are achieved by products of linear Lagrangian polynomials of the form $\frac{1}{4}(\xi + 1)(\eta + 1)$. For the eight

node element the following shape functions result:

$$N_1(\xi, \eta) = \frac{1}{4}(1 - \xi)(1 - \eta)(-\xi - \eta - 1)$$

$$N_2(\xi, \eta) = \frac{1}{4}(1 + \xi)(1 - \eta)(\xi - \eta - 1)$$

$$N_3(\xi, \eta) = \frac{1}{4}(1 + \xi)(1 + \eta)(\xi + \eta - 1)$$

$$N_4(\xi, \eta) = \frac{1}{4}(1 - \xi)(1 + \eta)(-\xi + \eta - 1)$$

$$N_5(\xi, \eta) = \frac{1}{2}(1 - \xi^2)(1 - \eta)$$

$$N_6(\xi, \eta) = \frac{1}{2}(1 + \xi)(1 - \eta^2)$$

$$N_7(\xi, \eta) = \frac{1}{2}(1 - \xi^2)(1 + \eta)$$

$$N_8(\xi, \eta) = \frac{1}{2}(1 - \xi)(1 - \eta^2)$$

β_{x_i} and β_{y_i} are transitory nodal variables at the four nodes and mid-sides of the element.

Figure 9: Eight-node quadrilateral element ("Seredipity"-element) with local ξ, η-coordinates.

Next, Batoz et al. described the Kirchhoff assumptions at the corner nodes (cf. Figure 9 for the following):

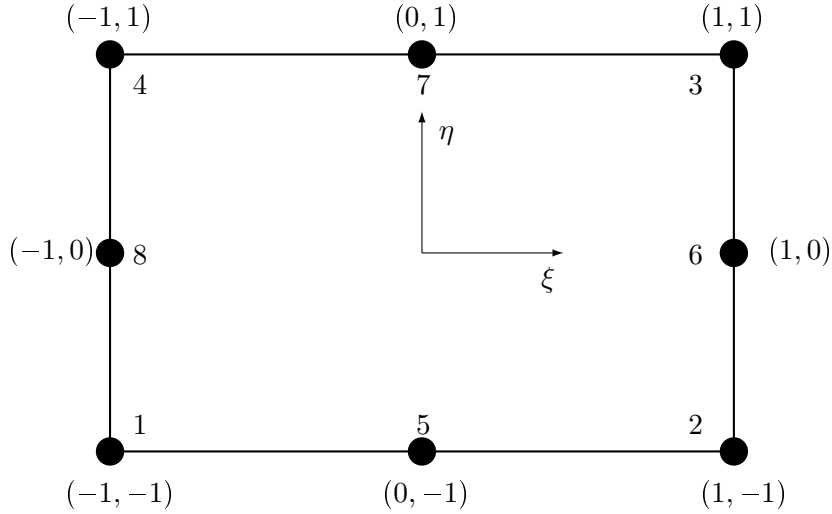$$\begin{pmatrix} \beta_{x_i} + \partial w/\partial x_i \\ \beta_{y_i} + \partial w/\partial y_i \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \qquad i = 1, 2, 3, 4 \tag{108}$$

and at the mid-nodes:

$$\beta_{s_k} + \partial w/\partial s_k = 0, \qquad k = 5,6,7,8 \tag{109}$$

where s denotes the coordinate along the element boundary and $\partial w/\partial s_k$ is the derivative of the displacement w with respect to the mid-node k:

$$\frac{\partial w}{\partial s_k} = -\frac{3}{2l_{ij}}(w_i - w_j) - \frac{1}{4}\left(\frac{\partial w}{\partial s_i} + \frac{\partial w}{\partial s_j}\right) \tag{110}$$

with $k = 5,6,7,8$ being the mid-node of side $ij = 12, 23, 34, 41$ and l_{ij} denotes the length of side ij. β_n varies linearly along the sides:

$$\beta_{n_k} = \frac{1}{2}\left(\beta_{n_i} + \beta_{n_j}\right) = -\frac{1}{2}\left(\frac{\partial w}{\partial n_i} + \frac{\partial w}{\partial n_j}\right) \tag{111}$$

with k same as before. β_x and β_y can be rewritten as follows:

$$\beta_x = H^x(\vec{\xi}, \eta)^T \vec{w} \tag{112}$$

$$\beta_y = H^y(\vec{\xi}, \eta)^T \vec{w} \tag{113}$$

$$\vec{w}^T = \begin{pmatrix} w_1 & \theta_{x_1} & \theta_{y_1} & w_2 & \theta_{x_2} & \theta_{y_2} & w_3 & \theta_{x_3} & \theta_{y_3} \end{pmatrix}$$

with

$$\vec{H^x}^T = \begin{pmatrix} H_1^x & \ldots & H_{12}^x \end{pmatrix}$$

$$H_{[1,4,7,10]}^x = \frac{3}{2}\left(a_{[5,6,7,8]}N_{[5,6,7,8]} - a_{[8,5,6,7]}N_{[8,5,6,7]}\right)$$

$$H_{[2,5,8,11]}^x = b_{[5,6,7,8]}N_{[5,6,7,8]} + b_{[8,5,6,7]}N_{[8,5,6,7]}$$

$$H_{[3,6,9,12]}^x = N_{[1,2,3,4]} - c_{[5,6,7,8]}N_{[5,6,7,8]} - c_{[8,5,6,7]}N_{[8,5,6,7]}$$

$$\vec{H^y}^T = \begin{pmatrix} H_1^y & \ldots & H_{12}^y \end{pmatrix}$$

$$H_{[1,4,7,10]}^y = \frac{3}{2}\left(d_{[5,6,7,8]}N_{[5,6,7,8]} - d_{[8,5,6,7]}N_{[8,5,6,7]}\right)$$

$$H_{[2,5,8,11]}^y = -N_{[1,2,3,4]} + e_{[5,6,7,8]}N_{[5,6,7,8]} + e_{[8,5,6,7]}N_{[8,5,6,7]}$$

$$H_{[3,6,9,12]}^y = -b_{[5,6,7,8]}N_{[5,6,7,8]} - b_{[8,5,6,7]}N_{[8,5,6,7]}$$

The function notation $H_{[i,j,k,l]}^x$ groups four functions together. The first function of the group gets the first index of the squared brackets, the second function the second index,

and so on. The coefficients a, b, c, d and e are as follows:

$$a_k = -\frac{x_{ij}}{l_{ij}^2}$$

$$b_k = \frac{3}{4}\frac{x_{ij}y_{ij}}{l_{ij}^2}$$

$$c_k = \frac{\frac{x_{ij}^2}{4} - \frac{y_{ij}^2}{2}}{l_{ij}^2}$$

$$d_k = -\frac{y_{ij}}{l_{ij}^2}$$

$$e_k = \frac{\frac{y_{ij}^2}{4} - \frac{x_{ij}^2}{2}}{l_{ij}^2}$$

where $k = 5, 6, 7, 8$ for the sides $ij = 12, 23, 34, 41$, $x_{ij} = x_i - x_j$, $y_{ij} = y_i - y_j$ and $l_{ij}^2 = x_{ij}^2 + y_{ij}^2$. For more details about the derivation of these coefficients and functions H^x and H^y see Batoz et al. [BT82].

Next, the Jacobian matrix $\underline{J}$ can be assembled, that is:

$$\underline{J} = \frac{1}{4}\begin{pmatrix} (x_{12} + x_{34})\eta - x_{12} + x_{34} & (y_{12} + y_{34})\eta - x_{12} + y_{34} \\ (x_{12} + x_{34})\xi - x_{13} - x_{24} & (y_{12} + y_{34})\xi - y_{13} + y_{24} \end{pmatrix} = \begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix} \quad (114)$$

With its determinant and inverse:

$$|\underline{J}| = J_{11}J_{22} - J_{12}J_{21} \quad (115)$$

$$\underline{J}^{-1} = \frac{1}{|\underline{J}|}\begin{pmatrix} J_{22} & -J_{12} \\ -J_{21} & J_{11} \end{pmatrix} = \begin{pmatrix} j_{11} & j_{12} \\ j_{21} & j_{22} \end{pmatrix} \quad (116)$$

The strain-displacement matrix can now be obtained:

$$\underline{B} = \begin{pmatrix} \vec{H}_x^x \\ \vec{H}_y^y \\ \vec{H}_y^x + \vec{H}_x^y \end{pmatrix} = \begin{pmatrix} j_{11} & j_{12} & 0 & 0 \\ 0 & 0 & j_{21} & j_{22} \\ j_{21} & j_{22} & j_{11} & j_{12} \end{pmatrix} \begin{pmatrix} \vec{H}_\xi^x \\ \vec{H}_\eta^x \\ \vec{H}_\xi^y \\ \vec{H}_\eta^y \end{pmatrix} \quad (117)$$

The expressions $\vec{H}_\xi^x, \vec{H}_\eta^x, \vec{H}_\xi^y$ and $\vec{H}_\eta^y$ are vectors containing the derivatives of the corresponding components of the vectors $\vec{H}^x$ and $\vec{H}^y$ with respect to ξ and η, respectively. And the matrix $\underline{B}$ can then be inserted into the displacement-strain relation, like equation (97):

$$\vec{\kappa} = \underline{B}\vec{w} \quad (118)$$

Next, $\vec{\kappa} = \underline{B}\vec{w}$ can be used in the functional to get the first term like equation (99):

$$
\begin{aligned}
\Pi_1 &= \frac{1}{2} \int_A \vec{\kappa}^T \underline{D}_p \vec{\kappa} \ dA \\
&= \frac{1}{2} \vec{w}^T \int_A \underline{B}^T \underline{D}_p \underline{B} \ dA \vec{w} \\
&= \frac{1}{2} \vec{w}^T \underline{K} \vec{w}
\end{aligned}
$$

with the stiffness matrix of the DKQ element $\underline{K}$:

$$
\begin{aligned}
\underline{K} &= \int_A \underline{B}^T \underline{D}_p \underline{B} \ dA \\
&= \int_{-1}^{1} \int_{-1}^{1} \underline{B}^T \underline{D}_p \underline{B} \ |\underline{J}| \ d\xi d\eta
\end{aligned} \tag{119}
$$

The stiffness matrix can be numerically integrated with a 2×2 Gaussian integration scheme. Batoz et al. states that four sampling points are enough, although a 3×3 point scheme would be necessary for exact integration on a rectangular element [BT82]. Those four sampling points are located at $\xi_i = \pm \frac{\sqrt{3}}{3}$ and $\eta_i = \pm \frac{\sqrt{3}}{3}$ with weight factor $\omega_i = 1$ equivalent to all four. The equation for the stiffness matrix can then be written in discretized form as follows:

$$
\underline{K} = \sum_{i=1}^{2} \sum_{j=1}^{2} \omega_i \omega_j \underline{B}(\xi_i, \eta_j)^T \underline{D}_p \underline{B}(\xi_i, \eta_j) \ |\underline{J}(\xi_i, \eta_j)| \tag{120}
$$

When all nodal values $\vec{w}$ are known, the moments $\vec{M}$ at point (x, y) in the element can be calculated:

$$
\vec{M}(x, y) = \underline{D}_p \underline{B}(x, y)\vec{w} \tag{121}
$$

with

$$
\vec{M} = \begin{pmatrix} M_x \\ M_y \\ M_{xy} \end{pmatrix} \tag{122}
$$

3.4 Coordinate Transformation

The nodes and elements in the mesh are defined in a global three dimensional coordinate system. The elements need to be transformed into a two dimensional local coordinate system in order to be able to construct their local stiffness matrices. This local stiffness matrix must then be transformed back into the global system before adding it to the global stiffness matrix. This section describes the building of the transformation matrix, that will be used in the following section for the addressed transformation steps. First the transformation of an arbitrary triangle defined in 3D space is described.

Figure 10: Arbitrary triangle with nodes A, B and C, defined in global xyz coordinate system. After transformation a new, local, $\tilde{x}\tilde{y}\tilde{z}$ coordinate system with node A in its origin, is created.

Given a triangle with vertices $\vec{A} = (a_x, a_y, a_z)^T, \vec{B} = (b_x, b_y, b_z)^T$ and $\vec{C} = (c_x, c_y, c_z)^T$ ordered in counterclockwise direction, as shown in Figure 10. Let $\vec{u}$ be the vector from node $\vec{A}$ to $\vec{B}$ and $\vec{v}$ be the vector from node $\vec{A}$ to $\vec{C}$:

$$\vec{u} = \vec{B} - \vec{A} = \begin{pmatrix} b_x - a_x & b_y - a_y & b_z - a_z \end{pmatrix}^T$$
$$\vec{v} = \vec{C} - \vec{A} = \begin{pmatrix} c_x - a_x & c_y - a_y & c_z - a_z \end{pmatrix}^T$$

First local unit vector:

$$\vec{\tilde{x}} = \frac{1}{|\vec{u}|}\vec{u}$$

Second local unit vector:

$$\vec{\tilde{z}} = \vec{u} \times \vec{v}$$
$$\vec{\tilde{z}} \leftarrow \frac{1}{|\vec{\tilde{z}}|}\vec{\tilde{z}}$$

Third local unit vector:

$$\vec{\tilde{y}} = \vec{\tilde{z}} \times \vec{\tilde{x}}$$

Define transformation matrix $\underline{T}$ as follows:

$$\underline{T} = \begin{pmatrix} \vec{\tilde{x}}^T \\ \vec{\tilde{y}}^T \\ \vec{\tilde{z}}^T \end{pmatrix} = \begin{pmatrix} \tilde{x}_x & \tilde{x}_y & \tilde{x}_z \\ \tilde{y}_x & \tilde{y}_y & \tilde{y}_z \\ \tilde{z}_x & \tilde{z}_y & \tilde{z}_z \end{pmatrix} \tag{123}$$

Assembly of element's stiffness matrix needs partial derivatives. In order to get these derivatives with less computational effort, every triangle can be translated in such a way,

that node $\vec{A}$ lies in the global origin before transforming it to local coordinates. Node $\vec{A}$ stays at $(0, \ 0, \ 0)$ coordinates which then simplifies getting the derivatives (see section 4.2.4). It follows:

$$\vec{\tilde{A}} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$$
$$\vec{\tilde{B}} = \begin{pmatrix} \tilde{b}_x & 0 & 0 \end{pmatrix}^T$$
$$\vec{\tilde{C}} = \begin{pmatrix} \tilde{c}_x & \tilde{c}_y & 0 \end{pmatrix}^T$$

Node $\vec{A}$ will not be changed by the transformation with $\underline{T}$, $\vec{B}$ will be projected onto the local x-axis that is defined to be the normalized vector between $\vec{A}$ and $\vec{B}$. Node $\vec{C}$ will be projected onto the local xy-plane. One can see that the z-component vanishes by transforming into local space, thus generating the two dimensional local space.
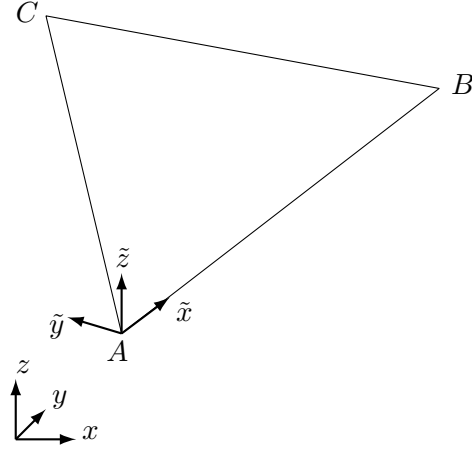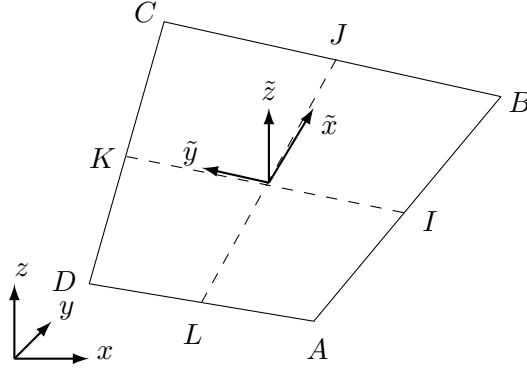

Figure 11: Quadrilateral with nodes A, B, C and D, defined in global xyz coordinate system. After transformation a new, local, $\tilde{x}\tilde{y}\tilde{z}$ coordinate system is created.

The other element described in this work is the quadrilateral. Let an arbitrary quadrilateral be given with vertices $\vec{A} = (a_x, \ a_y, \ a_z)^T$, $\vec{B} = (b_x, \ b_y, \ b_z)^T$, $\vec{C} = (c_x, \ c_y, \ c_z)^T$, $\vec{D} = (d_x, \ d_y, \ d_z)^T$ ordered in counter-clockwise direction, cf. Figure 11. Next, let $\vec{I}$ be the midpoint of edge $\overline{AB}$:

$$\vec{I} = \vec{A} + \frac{1}{2}\left(\vec{B} - \vec{A}\right)$$

Analogously let $\vec{J}, \vec{K}$ and $\vec{L}$ be the midpoints of the edges $\overline{BC}, \overline{CD}$ and $\overline{DA}$:

$$\vec{J} = \vec{B} + \frac{1}{2}\left(\vec{C} - \vec{B}\right)$$
$$\vec{K} = \vec{C} + \frac{1}{2}\left(\vec{D} - \vec{C}\right)$$
$$\vec{L} = \vec{D} + \frac{1}{2}\left(\vec{A} - \vec{D}\right)$$

Let then $\vec{u}$ be the vector from node $\vec{L}$ to $\vec{J}$ and $\vec{v}$ be the vector from node $\vec{I}$ to $\vec{K}$:

$$\vec{u} = \vec{J} - \vec{L} = \begin{pmatrix} j_x - l_x & j_y - l_y & j_z - l_z \end{pmatrix}^T$$
$$\vec{v} = \vec{K} - \vec{I} = \begin{pmatrix} k_x - i_x & k_y - i_y & k_z - i_z \end{pmatrix}^T$$

First local unit vector:

$$\vec{\tilde{x}} = \frac{1}{|\vec{u}|}\vec{u}$$

Second local unit vector:

$$\vec{\tilde{z}} = \vec{u} \times \vec{v}$$
$$\vec{\tilde{z}} \leftarrow \frac{1}{|\vec{\tilde{z}}|}\vec{\tilde{z}}$$

Third local unit vector:

$$\vec{\tilde{y}} = \vec{\tilde{z}} \times \vec{\tilde{x}}$$

Define transformation matrix T as follows:

$$\underline{T} = \begin{pmatrix} \vec{\tilde{x}}^T \\ \vec{\tilde{y}}^T \\ \vec{\tilde{z}}^T \end{pmatrix} = \begin{pmatrix} \vec{\tilde{x}}_x & \vec{\tilde{x}}_y & \vec{\tilde{x}}_z \\ \vec{\tilde{y}}_x & \vec{\tilde{y}}_y & \vec{\tilde{y}}_z \\ \vec{\tilde{z}}_x & \vec{\tilde{z}}_y & \vec{\tilde{z}}_z \end{pmatrix} \tag{124}$$

Remark: In order to transform a quadrilateral element from 3D to a local two dimensional space, the nodes of the original element must all be situated on a common plane. Otherwise such a transformation cannot be performed. Such shaped quadrilaterals cannot be used as shell elements.

3.5 Shell Element

Shell elements combine the capability of both, plane and plate elements. Every time a thin walled structure like a car body, dome structure or container with multi-axial pressures is to be simulated, shell elements provide a good solution. In this work only the so-called flat shell elements are described and used in the implementation. Details about curved shell elements, shells of revolution and general shells can be found in Cook et al. [CMPW02].

Flat shell elements have a state of bending and membrane stress that can be described by superposition of the plane and plate element [Kle13]. Figure 12 shows the superposition of plane and plate elements to a flat shell element. The degrees of freedom of plane and plate at every node are combined at the node of the shell element. Obviously the plane and plate element must be of the same finite element's type, for example three node triangular or eight node quadrilateral. The plane has displacements u and v with dedicated forces F_x and F_y. The plate has the deformation w with assigned normal force F_z and the two twists θ_x and θ_y with assigned moments M_x and M_y. Through the linking of the elements the shell element node has now five natural degrees of freedom. An additional twist around the local z-axis θ_z will be introduced [Ste15], increasing the
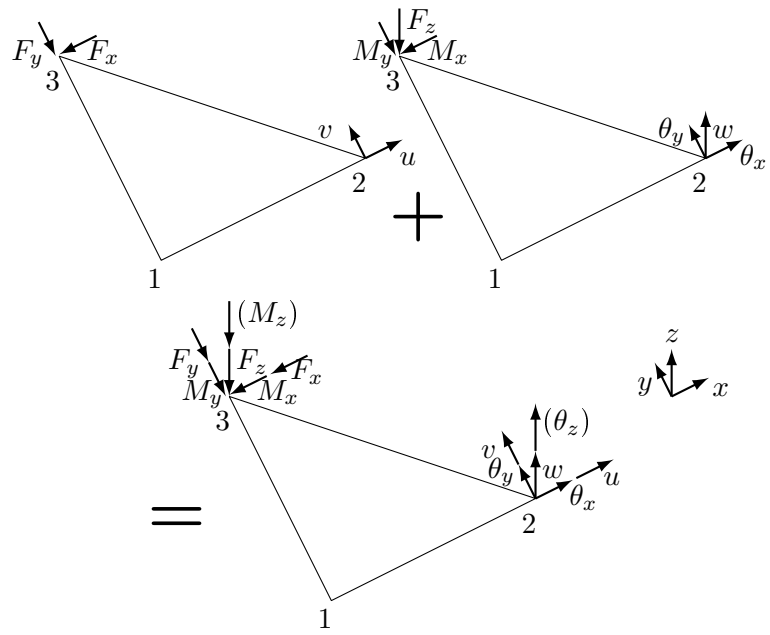
Figure 12: Creation of a triangular flat shell element by superimposing a triangular plane and a triangular plate element. The elements are described in the same local coordinate system with their degrees of freedom shown exemplary at one node per element.

number to six degrees of freedom per node. In vector notation the resulting displacement vector $\vec{u}_i$ of a shell element's node i is:

$$
\vec{u} = \underbrace{\begin{pmatrix} u \\ v \\ w \\ \theta_x \\ \theta_y \\ \theta_z \end{pmatrix}}_{\text{shell}} = \underbrace{\begin{pmatrix} u \\ v \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{\text{plane}} + \underbrace{\begin{pmatrix} 0 \\ 0 \\ w \\ \theta_x \\ \theta_y \\ 0 \end{pmatrix}}_{\text{plate}} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \theta_z \end{pmatrix} \tag{125}
$$

$$
\tag{126}
$$

For the two different finite elements in this work - the three node triangular element and the four node quadrilateral element - the stiffness matrix for the shell element is described by a block matrix of either 3×3 submatrices for the triangular case or 4×4 submatrices for the quadrilateral. The following equation shows the latter case as example:

$$
\underline{K}\vec{u} = \vec{F}
$$

$$
\begin{pmatrix} \underline{K}_{11} & \underline{K}_{12} & \underline{K}_{13} & \underline{K}_{14} \\ \underline{K}_{21} & \underline{K}_{22} & \underline{K}_{23} & \underline{K}_{24} \\ \underline{K}_{31} & \underline{K}_{32} & \underline{K}_{33} & \underline{K}_{34} \\ \underline{K}_{41} & \underline{K}_{42} & \underline{K}_{43} & \underline{K}_{44} \end{pmatrix} \begin{pmatrix} \vec{u}_1 \\ \vec{u}_2 \\ \vec{u}_3 \\ \vec{u}_4 \end{pmatrix} = \begin{pmatrix} \vec{F}_1 \\ \vec{F}_2 \\ \vec{F}_3 \\ \vec{F}_4 \end{pmatrix} \tag{127}
$$

where the vectors $\vec{u}_i$ are the same as in equation (125). The single submatrices $\underline{K}_{ij}$ of $\underline{K}$ were created by the superposition of the stiffness matrices of the plane and the plate:

$$
\underline{K}_{ij} = \left(\hat{\underline{K}}_{ij} \right)_m + \left(\hat{\underline{K}}_{ij} \right)_p \tag{128}
$$

The submatrix $\underline{K}_{ij}$ has the following structure:

$$
\begin{array}{cccccc} u & v & w & \theta_x & \theta_y & \theta_z \end{array}
$$
$$
\begin{pmatrix} \circ & \circ & 0 & 0 & 0 & 0 \\ \circ & \circ & 0 & 0 & 0 & 0 \\ 0 & 0 & \star & \star & \star & 0 \\ 0 & 0 & \star & \star & \star & 0 \\ 0 & 0 & \star & \star & \star & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} |\,u \\ |\,v \\ |\,w \\ |\,\theta_x \\ |\,\theta_y \\ |\,\theta_z \end{matrix} = \begin{pmatrix} \circ & \circ & 0 & 0 & 0 & 0 \\ \circ & \circ & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} |\,u \\ |\,v \\ |\,w \\ |\,\theta_x \\ |\,\theta_y \\ |\,\theta_z \end{matrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \star & \star & \star & 0 \\ 0 & 0 & \star & \star & \star & 0 \\ 0 & 0 & \star & \star & \star & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} |\,u \\ |\,v \\ |\,w \\ |\,\theta_x \\ |\,\theta_y \\ |\,\theta_z \end{matrix}
$$
$$
\tag{129}
$$

$\left(\hat{\underline{K}}_{ij} \right)_m$ describes the submatrix of the stiffness matrix of the plane element for node i and j and is marked with the $\circ$-symbol, $\left(\hat{\underline{K}}_{ij} \right)_p$ describes the corresponding matrix part

of the plate element's stiffness matrix and is symbolized with a $\star$. The degree of freedom θ_z does not exists in both, the plane and the plate element, and were introduced with the shell element. Therefore $(\underline{K}_{ij})_{66}$ is zero. The sixth degree of freedom is necessary because the missing stiffness regarding a rotation around the axis normal to the element could produce singularities in the overall stiffness matrix. This happens for example, if all neighboring elements of a node lie in the same plane, i.e. they are coplanar. A singularity can lead to a non-solvable system, so this case needs be excluded [Ste15]. One way is to introduce this sixth degree of freedom and give it a value that is so small that it does not influence the displacements and stresses too much. The number of this values varies: Werkle suggests a value of 1/10000 of the smallest diagonal entry of $\underline{K}_{ij}$ [Wer95], whereas [Kan04] used 1/1000 of the smallest diagonal entry of $\underline{K}_{ij}$. The value must be small, but big enough to prevent the singularities. Since this value is an approximation, one has to modify it, if the solution is not as expected or one cannot get a solution at all due to the addressed singularities.

The stiffness matrix for the shell element was constructed in a local coordinate system as described in section 3.2, 3.3 and 3.4. The overall stiffness matrix containing information about all elements need to be described in a global coordinate system. Before the element stiffness matrix is added to the global stiffness matrix, it has to be transformed from local to global. This can be achieved by transforming the single blocks of $\underline{K}$ from equation (127) with the relation:

$$\underline{\check{K}}_{ij}\vec{\check{u}}_j = \vec{\check{F}}_i \tag{130}$$

where " $\check{}$ " denotes that the matrix and vector are represented in local coordinates. With the help of the transformation matrix $\underline{\tilde{T}}$, the globally described displacement vector $\vec{u}_j$ and load vector $\vec{F}_i$ can be represented in local coordinates:

$$\vec{\check{u}}_j = \underline{\tilde{T}}\vec{u}_j \tag{131}$$

$$\vec{\check{F}}_i = \underline{\tilde{T}}\vec{F}_i \tag{132}$$

Since the load vector is to be defined in a global coordinate system and the resulting displacements are to be defined globally, too, equation (130) will be multiplied by $\underline{\tilde{T}}^T$ from left:

$$\underline{\tilde{T}}^T \underline{\check{K}}_{ij}\underline{\tilde{T}}\vec{u}_j = \underline{\tilde{T}}^T \underline{\tilde{T}}\vec{F}_i = \vec{F}_i \tag{133}$$

Hence, the two vectors will be represented in global coordinates and only the local element stiffness matrix need to be transformed. The addressed 6×6 transformation matrix $\underline{\tilde{T}}$ is made up of the 3×3 transformation matrix $\underline{T}$ from section 3.4:

$$\underline{\tilde{T}} = \begin{pmatrix} \underline{T} & 0 \\ 0 & \underline{T} \end{pmatrix} \tag{134}$$

In order to get the local element stiffness matrix $\underline{K}$ transformed to the global coordinate system, one has to transform the single submatrices $\underline{K}_{ij}$ as follows:

$$\underline{K}_{ij} = \underline{\tilde{T}}^T \underline{\check{K}}_{ij}\underline{\tilde{T}} \tag{135}$$

for $1 \leqslant i, j \leqslant 3$ in the case of the triangular element and $1 \leqslant i, j, \leqslant 4$ for the quadrilateral element, respectively.

4 FEM Code Implementation

In this chapter the FEM framework "libMesh" is presented at first that was used in the development of the program. Then, details on important parts of the implementation like the system's matrix assembly or mesh format requirements for import are illustrated. The parallelization with MPI concludes this chapter, pointing out details about libMesh's requirements for MPI usage and important modifications to the program's code.

4.1 Introduction to libMesh

The libMesh finite element library was stared as part of the Ph.D. thesis of Benjamin Kirk [Kir07]. It is a tool for numerical simulation of partial differential equations on serial and parallel platforms and uses the finite element method. A major goal is to provide data structures and algorithms for applications that need implicit numerical methods, parallel computing, adaptive mesh refinement techniques, or, a combination of them. Further, it simplifies many programming details for the user such as: Importing mesh structures, providing and initialize data structures for the equation systems, solving the discretized system and writing out the results [KPS13].

LibMesh allows discretization of one, two and three dimensional problems using several geometric element types, including: Edges, quadrilaterals, triangles, tetrahedra, hexahedra, pyramids, prisms and some infinite elements of quadrilaterals or hexahedra. Finite elements include traditional first and second order Lagrange, as well as arbitrary order hierarchical bases, and Nédélec elements of first type.

Mesh partitioning is available in libMesh through interfaces to several external packages, but also some internal partitioning algorithms are provided: Linear and centroid partitioner as examples of internal algorithms, Metis and ParMetis [KK98] as examples for external partitioner. In addition to these two, libMesh includes interfaces to solver libraries such as PETSc [pet] and LASPack [las]. Thus, libMesh offers several linear equation solvers such as GMRES, CG, Bi-CGSTAB, QMR, and preconditioners like Jacobi, incomplete LU factorization and incomplete Cholesky factorization. The choice of an appropriate solver and preconditioner can be made by the user at runtime or directly coded into the program.

A wide variety of mesh formats are supported by libMesh to facilitate the handling of complex geometries. The following is an incomplete list of supported input and output formats: Nemesis, TetGen, I-deas Universal UNV, AVS's ASCII UCD, Visualization Toolkit VTK, libMesh formats XDR/XDA, ExodusII, GMSH, LANL's General Mesh Viewer GMV, GnuPlot (only output), Matlab (only input) [KPS13].

An example program using the libMesh library would look like listing 1.

Listing 1: Example libMesh program

```
1  #include "libmesh/libmesh.h"
2  #include "libmesh/additional_libmesh_components"
3
4  using namespace libMesh;
5
```

```
 6  void assemble_something(EquationSystems& es, const std::string& system_name);
 7
 8  int main (int argc, char** argv)
 9  {
10      LibMeshInit init(int argc, char** argv);
11
12      Mesh mesh( init.comm() );
13
14      // mesh generation via MeshTools::Generation::build_... or mesh import from file via
            mesh.read(std::string filename)
15
16      EquationSystems es(mesh);
17
18      LinearImplicitSystem& system = es.add_system<LinearImplicitSystem> ("example_system");
19
20      system.add_variable ("a", FIRST);
21      system.add_variable ("b", SECOND, LAGRANGE);
22
23      system.attach_assemble_function (assemble_something);
24
25      es.init();
26
27      system.solve();
28
29      VTKIO (mesh).write_equation_systems ("out.pvtu", es);
30
31      return 0;
32  }
```

In fact, this is the base construction of a usual libMesh program. It starts with including
libMesh (*libmesh.h*) and all its components that are needed by the program, e.g. *mesh.h,
equation_systems.h, fe.h.* Then, the library needs to be initialized (line 10). This is
necessary because it may depend on a number of other external applications like MPI
or PETSc that require initialization before use. Furthermore, if the `LibMeshInit` object
goes out of scope, the other libraries are finalized automatically by libMesh. Next, a `Mesh`
object is created (lines 12-14) on the default MPI communicator even if the program is
executed single-threaded. The mesh can either be read from file or created by internal
mesh generation tools. In line 16 an `EquationSystems` object is created. It can contain
multiple different systems and manage them. Here, only one linear implicit system is
added to the object (line 16). Any `...System` object can contain multiple variables of
different approximation orders (see lines 20/21). Many systems require a user-defined
function that will assemble the (linear) system (lines 6 and 23). Now, the data structures
for the equation system must be initialized which is done in line 25. The solving of all
or just some of the systems added to the equation systems object is done in line 27 of
the code. This single line of code calls the assemble function defined earlier and invokes
the default numerical solver. If the external library PETSc is installed, the solver's
properties can be controlled from the command line by the user (cf. 4.2.1). After

solving the system, the solution can be written to file; here, for example, the results are written to a VTK-formatted plot file (line 29).

4.2 Implementation Details

This section contains details about the program's implementation supported by libMesh. Different parts of the code like the initialization, loading of the mesh or the system's matrix assembly are described. The focus is put on the interaction between the libMesh library and the user's code. Requirements regarding mesh formats and user arguments are pointed out as well.

4.2.1 Initialization

The program expects a few parameters set by the user through the command line at start. The ordering of these parameters are not relevant; some are optional. Here is a complete list of all arguments that can be set in the command line:

- **-nu**: The Poisson's ratio ν is required by the material matrices. A value in the range $0.0 < \nu \leqslant 0.5$ is recommended for most scenarios.

- **-e**: The elastic modulus or Young's modulus E is also required by the material matrices. Here, a value $E \gg 0$ is recommended.

- **-t**: The thickness t of the mesh. It is used at both, the material matrices and the strain-displacement matrices and thus a required parameter to be set by the user.

- **-d**: If set to "1" additional messages regarding transformation matrix entries, strain-displacement matrices, force load vectors and other internal mathematic structures are put out on the console. This parameter is optional, as it only gives the user more information in case of detecting errors. Since it slows down the calculation, it should only be set if needed. To turn the messages off, it can simply be left out or set to "0".

- **-mesh**: The file name of the mesh file to be imported. A required parameter, because no default mesh is provided by the program to be used. The relative path to the file (+ extension) must be specified. Allowed file formats are: libMesh format *xda* (ASCII) and *xdr* (binary) as well as GMSH format *msh*. For more details, see the following section.

- **-out**: The relative path and filename (*without* extension) for the output of the resulting mesh. This parameter is optional. If not set, no output file will be created. The path to the filename must exist, otherwise no file can be created and libMesh will terminate the program with an error message.

If the external library PETSc is installed and libMesh is configured to use it, the user can set additional optional command line arguments [BAA+15]. In fact, if one uses PETSc, it will look through all parameters to find those it can process by itself. The

following list is therefore limited to parameters that directly coincide with the need of this program. For more PETSc command line arguments see [BAA$^+$15].

- **-ksp_type**: Specifies the Krylov subspace method. Options are: `richardson`, `chebyshev`, `cg`, `gmres`, `tcqmr`, `bcgs`, `cgs`, `tfqmr`, `cr`, `lsqr`, `bicg`, `preonly`.

- **-pc_type**: To employ a particular preconditioning method used with the Krylov space method, the user can select one using this argument. Options are: `none`, `jacobi`, `bjacobi`, `sor`, `eisenstat`, `icc`, `ilu`, `asm`, `gasm`, `gamg`, `bddc`, `ksp`, `composite`, `lu`, `cholesky`, `shell`.

4.2.2 Mesh file import

The mesh geometry needs to be defined in a mesh file. LibMesh can import meshes from many different formats, including its own formats XDA and XDR, the first one stored in human-readable ASCII format, the latter one written in binary code. Another one is the GMSH format *msh*. There are other formats libMesh can import, but only the three mentioned are currently supported by the thesis' program. A mesh file must provide the following information such that the program can work with it:

- A list of vertices. Every vertex must be specified with its xyz-coordinates defined in the global coordinate system.

- A list of elements the mesh consists of. The elements are normally defined by their type, for example a three-node triangle or a four-node quadrilateral, and a list of vertex identifiers representing the element's nodes.

- A list of boundary conditions. The program provides two different types of boundary conditions. The type (or ID) has to be specified in form of identifiers on element's nodes or edges. In the latter case the boundary condition is used on both nodes defining the edge.

Listing 2 shows a short example of a mesh defined in the xda-format. It displays the unit square with its center at the global origin, composed of two three-node triangles. It has different boundary conditions on the bottom and top edge.

Listing 2: Example xda mesh file

```
 1  libMesh-0.7.0+
 2  2        # number of elements
 3  4        # number of nodes
 4  .        # boundary condition specification file
 5  n/a      # subdomain id specification file
 6  n/a      # processor id specification file
 7  n/a      # p-level specification file
 8  2        # n_elem at level 0, [ type (n0 ... nN-1) ]
 9  3 0 1 2        # 3 -> triangle with 3 nodes, 0 1 2 -> vertices 0, 1 and 2
10  3 1 3 2        #                             1 3 2 -> vertices 1, 3 and 2
11  -1.0 -1.0  0.0   # x y z coordinates of vertex 0
```

```
12   1.0 -1.0  0.0   #                      vertex 1
13  -1.0  1.0  0.0   #                      vertex 2
14   1.0  1.0  0.0   #                      vertex 3
15  2                # number of boundary conditions
16  0 0 1            # 0 -> element 0, 0 -> edge 0 (between vertex 0 and 1), bc-type 1
17  1 1 0            # 1 -> element 1, 1 -> edge 1 (between vertex 3 and 2), bc-type 0
```

Listing 3 shows the same example mesh but in the GMSH format. Here, libMesh has some requirements on how the GMSH mesh file has to be structured: Every line defined in the `$Elements`-section contains severals numbers, ordered as follows: Element index, element type, number of tags, physical entity number, geometrical entity, additional list of tags, list of node indices [gms]. LibMesh requires the number of tags to be at least two. The first tag (physical entity) will be used by libMesh to identify the boundary condition ID; the second tag will be ignored - at least the author could not find where libMesh uses this value. LibMesh also treats the element type in different ways: The highest dimensional element types, for example 2D elements, like triangle and quadrilaterals, will act as the mesh defining elements. Every element that has lower dimension, (e.g. nodes or edges), will be seen as boundary condition definitions by libMesh. See listing 3 for clarification: There are six elements defined. Two triangles and four single nodes. The mesh only exists of the two triangles. The four nodes will be used by libMesh to set boundary conditions at the corresponding nodes of the mesh. In this case node 1 and 2 gets boundary conditions with ID 0, node 3 and 4 with ID 1. This behavior of libMesh must be kept in mind when dealing with GMSH mesh files.

Listing 3: Example GMSH mesh file

```
1   $MeshFormat
2   2.2 0 8
3   $EndMeshFormat
4   $Nodes
5   4
6   1 -1.0 -1.0  0.0
7   2  1.0 -1.0  0.0
8   3 -1.0  1.0  0.0
9   4  1.0  1.0  0.0
10  $EndNodes
11  $Elements
12  6
13  1 2 2 0 0 1 2 3
14  2 2 2 0 0 2 4 3
15  3 15 2 0 0 1
16  4 15 2 0 0 2
17  5 15 2 1 0 3
18  5 15 2 1 0 4
19  $EndElements
```

The program features two different types of boundary conditions whose identifiers must be set in the mesh file:

- Simply supported boundaries are of type "0". The boundary cannot be moved but is free to rotate and have no moment resistance. In mathematical notation: $u = v = w = 0, M_x = M_y = 0$.

- Clamped boundary has type "1". Here, the boundary is completely fixed with no movement and no rotation possible. Mathematically: $u = v = w = 0, \theta_x = \theta_y = 0$

In the stand-alone version of the program no coupled fluid solver provides it with pressures/forces and moments at the nodes. Therefore, these values must be imported via file, too. The structure of such a file is simple: Listing 4 shows an example corresponding to the mesh of Listing 2. The first line defines the number n of nodes/vertices the mesh has (in this case $n = 4$). The second line holds a floating point number representing a global factor that is multiplied by every force/moment component defined below. A value of 1.0 has no effect on the load values. Lines 3 to $n + 2$ are the xyz-components of the single forces put on the corresponding mesh nodes followed by three values for the moments M_x, M_y, M_z. The ordering is the same as for the vertices in the mesh file. The xyz-coordinates must also be specified in the global coordinate system. In the example a load is applied on the first and third node. The first load is directed along the negative z-axis, the second load along the positive y-axis. The other two nodes have no forces and all nodes have no moments applied.

Listing 4: Example force file

```
4
1.0
0.0  0.0  -0.65  0.0  0.0  0.0
0.0  0.0   0.0   0.0  0.0  0.0
0.0  2.34  0.0   0.0  0.0  0.0
0.0  0.0   0.0   0.0  0.0  0.0
```

The implementation's code to import a mesh file is rather short (Listing 5).

Listing 5: Loading mesh and prepare for use

```
Mesh mesh(init.comm(), 2);
mesh.allow_renumbering(false);
mesh.read(in_filename);
mesh.print_info();
```

The first line creates a 2D mesh distributed across the default MPI communicator (gathered by the LibMeshInit-object). The mesh is read in line 3 from the file located at the place defined by in_filename. In line 4 information about the mesh is printed to the console. A special detail is line 2: It is not guaranteed that the ordering of the nodes defined in the mesh file is the same after libMesh has imported the mesh file. Since the program uses the additional force file to apply the loads onto the mesh nodes, this mapping could be destroyed. Therefore the function call in line 2 forbids libMesh to automatically renumber the nodes of the mesh and let the ordering be as defined in the mesh file.

4.2.3 System setup

After libMesh and possible external libraries are initialized and the mesh was created and initialized, too, the equation system must be set up. Listing 6 shows the relevant part of the implementation.

Listing 6: Setting up the equation system

```
1  EquationSystems equation_systems (mesh);
2  LinearImplicitSystem& system = equation_systems.add_system<LinearImplicitSystem> ("
       Elasticity");
3
4  unsigned int u_var  = system.add_variable("u",  FIRST, LAGRANGE);
5  unsigned int v_var  = system.add_variable("v",  FIRST, LAGRANGE);
6  unsigned int w_var  = system.add_variable("w",  FIRST, LAGRANGE);
7  unsigned int tx_var = system.add_variable("tx", FIRST, LAGRANGE);
8  unsigned int ty_var = system.add_variable("ty", FIRST, LAGRANGE);
9  unsigned int tz_var = system.add_variable("tz", FIRST, LAGRANGE);
10
11 system.attach_assemble_function (assemble_elasticity);
12
13 std::set<boundary_id_type> boundary_ids;
14 boundary_ids.insert(0);
15 std::vector<unsigned int> variables;
16 variables.push_back(u_var);
17 variables.push_back(v_var);
18 variables.push_back(w_var);
19 ConstFunction<Number> cf(0.0);
20 DirichletBoundary dirichlet_bc(boundary_ids, variables, &cf);
21
22 boundary_ids.clear();
23 boundary_ids.insert(1);
24 variables.push_back(tx_var);
25 variables.push_back(ty_var);
26 variables.push_back(tz_var);
27 DirichletBoundary dirichlet_bc2(boundary_ids, variables, &cf);
28
29 system.get_dof_map().add_dirichlet_boundary(dirichlet_bc);
30 system.get_dof_map().add_dirichlet_boundary(dirichlet_bc2);
31
32 equation_systems.init();
33 equation_systems.print_info();
```

In line 1 an `EquationSystems`-object is created. It contains and controls all equation systems defined for a mesh, that is passed as parameter in its constructor. It can have multiple systems or just one like in this case. Here, a linear implicit system is to be used. LibMesh offers exactly such a system. In line 2 the system is created named "Elasticity" and added to the equation systems object. As discussed in Section 3.5, the system has six variables, namely: $u, v, w, \theta_x, \theta_y, \theta_z$. These variables are added to the system in line 4 to 9. All of them are of first polynomial order and members of the Lagrange

finite element family. The `add_variable`-function returns a unique number identifying the variable just added. In order to assemble the system matrix and the right-hand side, a user-defined function must be attached to the system. This is done in line 11. The assemble function will be discussed in section 4.2.4. The only part missing is the definition of the different boundary conditions. This is done between the lines 13 and 30. As stated in the previous section, the program features two types of boundary conditions: Simply supported and clamped with the ID "0" and "1", respectively. Because libMesh allows multiple IDs representing the same boundary condition type, in line 13 a `set` is created and filled with the 0-ID in the next line. After that, a vector containing the IDs of the system's variables must be created. In the case of a simply supported boundary only the three displacement variables u, v, w are affected (lines 15-18). Before creating the **DirichletBoundary**-object, a function supplying the Dirichlet value must be defined. In this case the value must be zero at the boundary. A `ConstFunction` initialized with the value zero is therefore created in line 19. The constructor for the Dirichlet object takes the ID-set, the variables-vector and the value-function as parameters and copies their content into the boundary object. The set and vector will be reused for the second boundary type. Now the 1-ID is inserted in the cleared set in line 23 and the twist variables $\theta_x, \theta_y, \theta_z$ are added to the existing variables in the vector (lines 24ff.). Another `DirichletBoundary` object is created with the new initialization parameters. Finally, the two boundary types are added to the system in line 29f. When the preparation steps are finished, the system can finally be initialized and information about the system can be print to the console.

4.2.4 Matrix and vector assembly

As stated in Section 4.1, libMesh tries to do as much programming tasks as possible on its own and let the user concentrate on the mathematical/physical problems to model. The last sections showed that this is often true, since the user often only need to set parameters, create predefined objects or call libMesh functions. To solve the system, libMesh calls a user-defined assembly function. This function is the part where the user gets involved at most, because here the system matrix and the right-hand side (RHS) must be assembled. Nevertheless, libMesh helps the user with many auxiliary function as can be seen further down. The main part of the assembly function is described in Listing 7. The first step is to get a reference to the system whose matrix and vector needs to be assembled. In this case the "Elasticity" system is used (line 1). Next, a reference to a special object is get from the system: The **DofMap**-object handles the numbering of degrees of freedom on a mesh and manages the mapping of the local matrices to the right positions in the overall system matrix. In line 3 to 9 several variables are defined: `Ke`, `Ke_m` and `Ke_p` are matrices representing the shell element's stiffness matrix, plane (or **m**embrane) stiffness matrix part and **p**late stiffness matrix part. The values of the element's RHS are stored in the vector `Fe`. The other three matrices are needed for the transformation of the element to local coordinates (`trafo`), the storage of the transformed nodes (`transUV`) and the partial derivatives (`dphi`). The element's area is stored in the variable of the same name.

Listing 7: Assemble System Matrix and RHS

```
1  LinearImplicitSystem& system = es.get_system<LinearImplicitSystem>("Elasticity");
2  const DofMap& dof_map = system.get_dof_map();
3
4  DenseMatrix<Number> Ke, Ke_m, Ke_p;
5  DenseVector<Number> Fe;
6  DenseMatrix<Real> trafo;
7  DenseMatrix<Real> transUV;
8  DenseMatrix<Real> dphi;
9  Real area = 0.0;
10
11 std::vector<dof_id_type> dof_indices;
12 std::unordered_set<dof_id_type> processedNodes;
13 processedNodes.reserve(mesh.n_local_nodes());
14
15 MeshBase::const_element_iterator el = mesh.active_local_elements_begin();
16 const MeshBase::const_element_iterator end_el = mesh.active_local_elements_end();
17 for (; el != end_el; ++el)
18 {
19     const Elem* elem = *el;
20     dof_map.dof_indices (elem, dof_indices);
21     ElemType type = elem->type();
22
23     initElement(&elem, transUV, trafo, dphi, &area);
24
25     calcPlane(type, transUV, dphi, &area, Ke_m);
26     calcPlate(type, dphi, &area, Ke_p);
27     constructStiffnessMatrix(type, Ke_m, Ke_p, Ke);
28     localToGlobalTrafo(type, trafo, Ke);
29     contribRHS(&elem, Fe, &processedNodes);
30
31     dof_map.constrain_element_matrix_and_vector(Ke, Fe, dof_indices);
32
33     system.matrix->add_matrix (Ke, dof_indices);
34     system.rhs->add_vector (Fe, dof_indices);
35 }
```

The assembly function creates the local stiffness matrix and RHS for every single finite element and adds it to the global system matrix and system right-hand side. Therefore, one has to iterate over all the mesh elements. This is done in line 15 to 35. The following steps are made in the exact same order:

- Get the mapping of the element's degrees of freedom to their positions in the system matrix (line 20).

- Transform the element from global to local coordinates, calculate its partial derivatives and its area (line 23).

- Assemble the plane stiffness matrix part of the shell element (line 25).

- Assemble the plate stiffness matrix part of the shell element (line 26).

- Construct the shell stiffness matrix for the current element (line 27).

- Transform the local shell stiffness matrix back to global coordinates (line 28).

- Apply possible forces to the element in form of contributing to the local RHS (line 29).

- Constrain the local element's stiffness matrix and RHS according to the set boundary conditions (line 31): LibMesh provides a function that automatically constrains the system matrix and right-hand side vector due to the boundary condition definitions in the initialization step.

- Add the element's stiffness matrix to the global system matrix (line 33): The final element's stiffness matrix is added to the overall system matrix through the function provided by libMesh. The vector storing the mappings of the degrees of freedom ensures that the matrix is added at the correct position in the system matrix.

- Add the element's RHS to the global RHS (line 34): Same as before but now for the right-hand side vector.

In the following, some of the steps stated above are discussed in more detail, namely the coordinate transformation, the construction of the shell element stiffness matrix and the contribution to the right-hand side vector.

The transformation from global to local coordinates is described in Section 3.4. Let $\tilde{A} = \begin{pmatrix} x_1 = 0 & y_1 = 0 \end{pmatrix}^T, \tilde{B} = \begin{pmatrix} x_2 & y_2 = 0 \end{pmatrix}^T, \tilde{C} = \begin{pmatrix} x_3 & y_3 \end{pmatrix}$ be the nodes of a transformed triangular element. Then the partial derivatives are as follows:

$$
\begin{aligned}
x_{12} = x_1 - x_2 = -x_2 && y_{12} = y_1 - y_2 = 0 \\
x_{31} = x_3 - x_1 = x_3 && y_{31} = y_3 - y_1 = y_3 \\
x_{23} = x_2 - x_3 && y_{23} = y_2 - y_3 = -y_3
\end{aligned}
$$

For a transformed quadrilateral the derivatives are straightforward: $x_{ij} = x_i - x_j, y_{ij} = y_i - y_j$, because no implicit positions can be assumed for the transformed nodes.

The area of a triangle can easily be calculated during the creation of the transformation matrix. The cross product between the spanning vectors $\vec{u}$ and $\vec{v}$ (cf. Section 3.4) is 2 times the area of the triangle. Thus, the area is:

$$A_\triangle = \frac{1}{2} \left| \vec{u} \times \vec{v} \right|$$

For the quadrilateral area $A_\square$ one can use the Gauss' area formula:

$$A_\square = \frac{1}{2} \sum_{i=1}^{4} \left| \begin{pmatrix} x_i & x_{(i+1)\%4} \\ y_i & y_{(i+1)\%4} \end{pmatrix} \right|$$

where "%" denotes the modulo-operator.

After the assembly of the plane stiffness part described in Section 3.2.2 for triangular elements and in 3.2.3 for quadrilateral elements, a matrix K_m with $2n \times 2n$ entries results, n being the number of nodes the element has:

$$K_m = \begin{matrix} & \underline{u_1} & \underline{v_1} & \underline{u_2} & \cdots & \underline{u_n} & \underline{v_n} & \\ & \begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & \cdots & m_{1,n-1} & m_{1,n} \\ m_{2,1} & m_{2,2} & m_{2,3} & \cdots & m_{2,n-1} & m_{2,n} \\ m_{3,1} & m_{3,2} & m_{3,3} & \cdots & m_{3,n-1} & m_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \cdots & \vdots \\ m_{n-1,1} & m_{n-1,2} & m_{n-1,3} & \cdots & m_{n-1,n-1} & m_{n-1,n} \\ m_{n,1} & m_{n,2} & m_{n,3} & \cdots & m_{n,n-1} & m_{n,n} \end{pmatrix} & \begin{matrix} | u_1 \\ | v_1 \\ | u_2 \\ | \vdots \\ | u_n \\ | v_n \end{matrix} \end{matrix}$$

The same holds for the plate stiffness part described in Sections 3.3.2 and 3.3.3. Here, the matrix has $3n \times 3n$ entries, n also the number of elements nodes:

$$K_p = \begin{matrix} & \underline{w_1} & \underline{\theta_{x_1}} & \underline{\theta_{y_1}} & \underline{w_2} & \cdots & \underline{\theta_{x_n}} & \underline{\theta_{y_n}} & \\ & \begin{pmatrix} p_{1,1} & p_{1,2} & p_{1,3} & p_{1,4} & \cdots & p_{1,n-1} & p_{1,n} \\ p_{2,1} & p_{2,2} & p_{2,3} & p_{2,4} & \cdots & p_{2,n-1} & p_{2,n} \\ p_{3,1} & p_{3,2} & p_{3,3} & p_{3,4} & \cdots & p_{3,n-1} & p_{3,n} \\ p_{4,1} & p_{4,2} & p_{4,3} & p_{4,4} & \cdots & p_{4,n-1} & p_{4,n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \cdots & \vdots \\ p_{n-1,1} & p_{n-1,2} & p_{n-1,3} & p_{n-1,4} & \cdots & p_{n-1,n-1} & p_{n-1,n} \\ p_{n,1} & p_{n,2} & p_{n,3} & p_{n-1,4} & \cdots & p_{n,n-1} & p_{n,n} \end{pmatrix} & \begin{matrix} | w_1 \\ | \theta_{x_1} \\ | \theta_{y_1} \\ | w_2 \\ | \vdots \\ | \theta_{x_n} \\ | \theta_{y_n} \end{matrix} \end{matrix}$$

In Section 3.5, it is described that the two stiffness matrices from plane and plate can be superimposed to the shell element's stiffness matrix. This results in a matrix K that consists of $n \times n$ submatrices ($6n \times 6n$ entries per submatrix):

$$K = \begin{pmatrix} K_{11} & K_{12} & \cdots & K_{1n} \\ K_{21} & K_{22} & \cdots & K_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ K_{n1} & K_{n2} & \cdots & K_{nn} \end{pmatrix}$$

Every submatrix describes the stiffness for one node of the shell element, i.e.:

$$K_{ij} = \begin{pmatrix} m_{2i,2j} & m_{2i,2j+1} & 0 & 0 & 0 & 0 \\ m_{2i+1,2j} & m_{2i+1,2j+1} & 0 & 0 & 0 & 0 \\ 0 & 0 & p_{3i,3j} & p_{3i,3j+1} & p_{3i,3j+2} & 0 \\ 0 & 0 & p_{3i+1,3j} & p_{3i+1,3j+1} & p_{3i+1,3j+2} & 0 \\ 0 & 0 & p_{3i+2,3j} & p_{3i+2,3j+1} & p_{3i+2,3j+2} & 0 \\ 0 & 0 & 0 & 0 & 0 & d_{ij} \end{pmatrix}$$

where d_{ij} is a thousandth of the maximum of the diagonal entries of K_{ij} (cf. Section 3.5):

$$d_{ij} = \frac{1}{1000} \max \{m_{2i,2j}, m_{2i+1,2j+1}, p_{3i,3j}, p_{3i+1,3j+1}, p_{3i+2,3j+2}\}$$

After this step, the local stiffness matrix for the shell element is finally constructed. Before it can be added to the overall system stiffness matrix, it must be transformed back to the global coordinate system as discussed in Section 3.4. For this purpose let $\tilde{T}$ be the 6×6 transformation matrix from Equation (134). Now, every submatrix K_{ij} from K must be transformed in the following way (Eq. (135)):

$$\bar{K}_{ij} = \tilde{T}^T K_{ij} \tilde{T}$$

for $1 \leqslant i, j \leqslant n$.

The resulting global coordinate matrix $\bar{K}_{ij}$ has the same structure as its local ancestor. In order to add it to the system matrix, one has to modify this structure. Until now, the ordering of columns and rows were as follows: All variables for node 1, followed by all variables for node 2, etc. $(u_1, v_1, w_1, \ldots, \theta_{y_n}, \theta_{z_n})$. LibMesh wants to system matrix to be in another format: The first variable for all nodes, then the second variable for all nodes, etc. $(u_1, u_2, \ldots, u_n, v_1, v_2, \ldots, \theta_{z_n})$. This change in format is achieved by the following code (Listing 8):

Listing 8: Bring stiffness matrix into libMesh conform format

```
for (α = 0..5)
    for (β = 0..5)
        for (i = 0..n-1)
            for (j = 0..j-1)
                K_libmesh(αn+i, βn+j) = K̄(6i+α, 6j+β);
```

The right-hand side of the shell element is constructed in the function described in Listing 9.

Listing 9: Contribute RHS function

```
1  void contribRHS(const Elem **elem, DenseVector<Real> &Fe, std::unordered_set<unsigned int> *
       processedNodes)
2  {
3      unsigned int nsides = (*elem)->n_sides();
4      Fe.resize(6*nsides);
```

```
 5
 6     DenseVector<Real> arg;
 7     for (unsigned int side = 0; side < nsides; side++)
 8     {
 9         Node* node = (*elem)->get_node(side);
10         dof_id_type id = node->id();
11
12         if (processedNodes->find(id) == processedNodes->end())
13         {
14             processedNodes->insert(id);
15             arg = forces[id];
16             Fe(side) = arg(0);
17             Fe(side+nsides) = arg(1);
18             Fe(side+nsides*2) = arg(2);
19         }
20     }
21 }
```

The local vector storing the forces and moments has $6n$ entries: Three forces and three moments for each of the n nodes ($n = 3$ for the triangular element, $n = 4$ for the quadrilateral). The values are stored in a global vector that was filled at the beginning of the program with the entries of the force file. At first (line 3 and 4) the number of sides of the element is get (equals the number of element's nodes n) and the RHS-vector is resized accordingly. Then, one has to iterate over the sides/nodes of the element. Here, a problem occurs due to the iteration of the assembly function over all mesh elements: The majority of mesh nodes belongs to more than one element. In order to keep the global RHS data consistent such a node must be processed only once. In Listing 7 an `unordered_set` structure were created in line 12. This set keeps record of the IDs of already processed nodes. In line 12 of the RHS contribution function it is checked if the current element node's ID is already existing in that set. If so, the node is skipped and the function continues with the next one. If not, that node's ID is inserted into the set and the corresponding force/moment entries from the global vector is written to the local RHS-vector.

4.2.5 Solving the system

The system's solving is done in libMesh with only one call: `equation_systems.solve()`. This function performs two actions: It calls the user-defined assembly function that constructs the system matrix and right-hand side and after that it starts the solver. The rest is handled by libMesh or libraries like PETSc, respectively, though the user has some options to control the solver's behavior. Listing 10 shows the code section responsible for the system's solving.

Listing 10: Solve the system and build solution

```
1 // optionally set solver parameters:
2 const unsigned int max_iter  = equation_systems.parameters.get<unsigned int>("linear_solver
       maximum_iterations");
```

```
3  const Real          tolerance = equation_systems.parameters.get<Real>("linear_solver_
       tolerance");
4  equation_systems.parameters.set<unsigned int>("linear_solver_maximum_iterations")= max_iter;
5  equation_systems.parameters.set<Real>        ("linear_solver_tolerance")        = tolerance;
6
7  equation_systems.solve();
8
9  std::vector<Number> sols;
10 equation_systems.build_solution_vector(sols);
11
12 MeshBase::const_node_iterator no = mesh.nodes_begin();
13 const MeshBase::const_node_iterator end_no = mesh.nodes_end();
14 for (; no != end_no; ++no)
15 {
16      Node *nd = *no;
17      int id = nd->id();
18      Real displ_x = sols[6*id];
19      Real displ_y = sols[6*id+1];
20      Real displ_z = sols[6*id+2];
21      (*nd)(0) += displ_x;
22      (*nd)(1) += displ_y;
23      (*nd)(2) += displ_z;
24 }
```

In line 7 the addressed solve-function is called. Before, one can modify the maximum
number of iterations the solver will do and the residual's tolerance signaling when the
solver should stop. In this case nothing is changed, since the original parameter values
are set. It was added to the source code to show the syntax and mention the possibility,
but was not used in the program. The equation systems object stores the calculated
solution internally. In order to get access to the results, one has to build a vector (line
9f.) where the solutions can be stored in. LibMesh resizes the vector automatically and
fills it with the solution values. The order hereby is as follows: First all six values for
the d.o.f. for the first node in the order of adding them to the system, then all six values
for the second node and so on, i.e. $\vec{sols} = \begin{pmatrix} u_1 & v_1 & w_1 & \theta_{x_1} & \theta_{y_1} & \theta_{z_1} & u_2 & \dots & \theta_{z_n} \end{pmatrix}$
for a mesh with n nodes. The nodes are ordered as defined in the mesh file. Until this
position in the code, only the displacements and twists are calculated. What is left is to
apply the displacements to the mesh. This is done between line 12 and 24. One has to
iterator over all mesh nodes. LibMesh provides a special iterator-type for nodes to do
this. The current node's ID is stored in a variable in line 17 and then the displacements
for the x-, y- and z-direction is get from the vector. Note, that the values are represented
in global coordinates like the coordinates of the nodes in the mesh object. Hence, no
transformation is necessary. The single direction displacements are then added to the
existing absolute values of the mesh nodes.

4.2.6 Output

When the system is solved and the results are applied to the mesh, the program can write the displaced mesh with its additional data to a file. Listing 11 shows the relevant function. First, the function checks if the user does not wish any output to be made. If this is the case nothing has to be done. Otherwise, an output stream is created with the filename specified by the user as command line argument plus the ExodusII extension ".e". Finally, the mesh with all additional data is written to the specified file.

Listing 11: Store results in mesh file

```
void writeOutput(Mesh &mesh, EquationSystems &es)
{
    if (!isOutfileSet)
        return;

    std::ostringstream file_name;
    file_name << out_filename << ".e";
    ExodusII_IO (mesh).write_equation_systems(file_name.str(), es);
}
```

The ExodusII-format was chosen for two reasons: First, libMesh can handle parallel output for this format on multiple processes. This would not be the case for VTK, for instance. Parallel output is required when the program is executed on multiple processes. Second, the ExodusII format can be opened and analyzed by ParaView - an open source multiple-platform application for interactive, scientific visualization - which gives the user more flexibility for post-processing work and analysis.

4.3 Parallelization with MPI

When the complexity of the mesh gets larger or the program is coupled with other solvers and is executed multiple times per time step, it is important to parallelize the computations in order to solve the system in shorter time. The parallelization process with MPI has three major parts: The library itself, the partition of the mesh and the solving of the system. In the following sections these parts are described in more detail. Additionally, it will be stated what code changes had to be performed in order to make the program code working with MPI.

4.3.1 libMesh requirements

Although MPI is used internally throughout the libMesh library as soon as the program runs in MPI-mode, not everything will work as expected. If one tries to run a theoretically fully MPI-compatible libMesh program and tries to solve a system like the linear implicit system in this case, wrong solutions will result. The problem occurs, when libMesh is built and configured without PETSc support. Then, libMesh will use an Eigen sparse solver [eig]; either a version installed in the user's environment or a built-in version from libMesh itself. These solvers are only serial and cannot be used with MPI,

referring to an answer on the libMesh-mailing list from Dr. John Peterson, one of the libMesh developers. The solution would be to install PETSc and build libMesh against it. The usage of PETSc when libMesh is executed in parallel is a strong recommendation from the developers of libMesh.

4.3.2 Partitioning the mesh

The parallelization of the solving step is directly linked to the partitioning of the mesh. Every process gets a part of the mesh and solves the system on its partition. LibMesh provides different partitioning algorithms the user can choose from, namely:

- `LinearPartitioner`: The linear partitioning algorithm is the simplest of all available partitioners. It takes the element list and splits it into equally-sized parts and assigns them to each processor.

- `CentroidPartitioner`: The centroid partitioner partitions simply based on the locations of element centroids. It must be defined how to sort the element centroids, i.e. the distance in the x, y or z-direction or the radial distance.

- `MetisPartitioner`: This partitioner uses the METIS graph partitioner to partition the elements. This partitioner will be used by libMesh at default if no other partitioner is specified.

- `ParmetisPartitioner`: Here, the ParMETIS graph partitioner will be used to partition the elements.

- `HilbertSFCPartitioner`: The Hilbert SFC partitioner uses a Hilbert space filling curve to partition the elements.

- `MortonSFCPartitioner`: Same as before but with a Morton space filling curve to partition the elements.

It is easy to use any of these partitioners in code. Listing 12 shows an example of the ParMETIS partitioner. Instead of this partitioner, every other listed class can be placed at line 7 with the corresponding header file included (line 1).

Listing 12: Mesh partitioning example

```
1  #include "libmesh/partmetis_partitionier.h"
2  // (...)
3  Mesh mesh(init.comm(), 2);
4  mesh.allow_renumbering(false);
5  mesh.read(in_filename);
6
7  ParmetisPartitioner partitioner;
8  partitioner.partition(mesh);
9  // (...)
```

4.3.3 Local elements

LibMesh uses a collection of different iterator-types to go through the mesh elements and nodes. This collection can be split into two groups: Local and global iterators. In the serial case there is no difference between the two, since all elements and nodes exist on the same process. If the program is run with multiple processes on a partitioned mesh, it gets important to use local iterators. Thus, the single processes only iterate over their "own" elements and nodes without risking to interfere with the neighboring processes. This behavior gets especially important in the assembly function (cf. Section 4.2.4). Listing 13 shows a short code part of this function. Here, the for-loop header is shown that will iterate over the mesh elements in order to assemble the element's stiffness matrix and right-hand side and add them to the global system matrix and right-hand side, respectively. Note that it will be iterated over `active_local_elements`. In the serial case it could also be possible to only use `active_elements`. "active" in this case means, that only non-deactivated elements from the mesh will be processed. Since the deactivation of mesh parts is not used in this implementation, one can ignore this term.

Listing 13: Local elements iterator

```
1  // (...)
2  MeshBase::const_element_iterator       el    = mesh.active_local_elements_begin();
3  const MeshBase::const_element_iterator end_el = mesh.active_local_elements_end();
4  for (; el != end_el; ++el)
5  {
6       // (...)
7  }
```

An example where global iterators are still possible and useful is after the solving step (see Section 4.2.5. Listing 14 shows parts of the code after the system gets solved.

Listing 14: Global nodes iterator

```
1   // (...)
2   std::vector<Number> sols;
3   equation_systems.build_solution_vector(sols);
4
5   if (global_processor_id() == 0)
6   {
7        MeshBase::const_node_iterator            no = mesh.nodes_begin();
8        const MeshBase::const_node_iterator end_no = mesh.nodes_end();
9        for (; no != end_no; ++no)
10       {
11            // (...)
12       }
13       // (...)
14  }
15  // (...)
```

LibMesh produces the solution vector only on the master process which has ID 0 per definition. Even when running the program with multiple processes, the solution vector on process 0 contains the values for **all** mesh nodes. Every other process does not have access to the solution vector. In order to apply the displacements onto the mesh nodes, process 0 must iterate over all mesh nodes. Instead of `local_nodes`, in line 7 and 8 `nodes` is used which represents all the nodes of the mesh.

4.3.4 Assembly changes

In Section 4.2.4 it was stated that a problem occurs at the creation of the local right-hand side's vector: Since nodes can be part of multiple elements, they can be processed multiple times and contribute more than one time to the RHS. This was prevented by creating an `unordered_set` that stores the ID of already processed nodes. In the serial case this solves the problem. In case of parallel execution, every process generates such a set for itself without knowing what the other processes have already stored in theirs. This way, it cannot be decided if a node that is situated on the boundary between two mesh partitions was already processed by one of these processes. One solution would be to communicate the processed node IDs between the processes such that the information is consistent throughout the mesh. But that would produce a lot of communication overhead and would slow down the program down. With the help of libMesh this problem can be solved without any inter-process communication at all. Listing 15 contains extracts from the assembly function and the auxiliary function to create the local right-hand side.

Listing 15: Process local nodes only

```
1  // in the assemble_elasticity function:
2  std::unordered_set<dof_id_type> processedNodes;
3  processedNodes.reserve(mesh.n_local_nodes());
4
5  //----------------------------------------------------------------
6
7  // in the contribRHS function:
8  Node* node = (*elem)->get_node(side);
9  int   id   = node->id();
10
11 if (node->processor_id() != global_processor_id())
12     continue;
13
14 if (processedNodes->find(id) == processedNodes->end())
15 {
16     processedNodes->insert(id);
17     // (...)
18 }
19 // (...)
```

Here, one can see in line 3 that every process will create an set with a size reserved to only the number of local elements that process has access to. When the mesh file is

imported, LibMesh stores the ID of the process in all nodes that are part of the partition assigned to that process. This is exploited in line 11: If the stored ID of the node does not match the process' ID than it will be continued with the next node of the element. This can only be the case for nodes situated on the partition's boundary with another partition neighboring. If the node does not belong to the calling process, it must belong to the neighboring process. It is impossible to skip a node, since all nodes belong to at least one process. With this line of code one can ensure that a node is processed only once: If it is not owned by the calling process, it is not processed at all. Otherwise it will be processed once and than marked in line 16 such that the same process will not use it in the future.

5 Coupling through preCICE

When a physical problem becomes too complex, one often split it into smaller pieces that are better manageable. These pieces, or physical fields, can be used to get separate solutions which can then be combined to an overall solution. This approach is called "partitioned approach" [Gat15]. It allows to reuse simulation code for the single fields and simultaneously provides the possibility to encapsulate the coupling functionality itself as a reusable component. This allows minimal access to solver codes; treating them as black boxes. At the same time the solver code does not have to include the whole coupling code making it less application dependent. The coupling tool preCICE (**pre**cise **C**ode **I**nteraction **C**oupling **E**nvironment) offers coupling functionality to develop a multi-physics simulation environment using existing solvers. In this work a fluid-structure interaction (FSI) is to be simulated. The thesis' program represents the structure part whereas the fluid part can be dynamically exchanged due to the coupling with preCICE. This Chapter gives an overview of preCICE and its main components and shows the code modifications that were necessary to integrate preCICE and prepare the program for coupling.

5.1 Overview of preCICE

The goal of preCICE is to provide all functionality to realize a multi-physics simulation environment working with existing single-physics solvers. This includes simulations like fluid-structure, fluid-acoustics, fluid-solid thermodynamics and porous-free flow interactions, for instance. It provides technical inter-code communication via MPI or TCP/IP, methods for data mapping between different grids and coupling methods based on quasi-Newton methods to ease the development process. preCICE supports parallel solvers through efficient point-to-point communication without the need of a server instance. It also features a high-level API making its integration into existing solver code minimal invasive [bun].

It supports partitioned coupling of black box solvers with focus on FSI and provides a geometry interface for Cartesian grid solvers. Its API is available for C++, C and Fortran and consists of methods enabling solvers to use coupling functionality in a flexible way. After the preCICE integration into a solver's code, a peer-to-peer communication without central control instance is generated. The single solver codes can be run serial or parallel without major modifications to the integrated API. Furthermore, the concrete coupling algorithms in the simulation can be selected via an XML configuration file that optimizes the solver adaption [Gat15].

Figure 13 shows a schematic view of the main functionality groups of preCICE. Two solvers (A and B) are coupled through the preCICE tool. The three main functionalities of preCICE are shown in the middle: Fix-point acceleration methods, point-to-point solver process communication and data mapping between non-matching grids at the coupling surface. Each of the solver runs in parallel with a master process controlling tasks such as convergence for the corresponding solver as well as for the whole simulation. The following sections describe each of the three addressed main functionalities in more
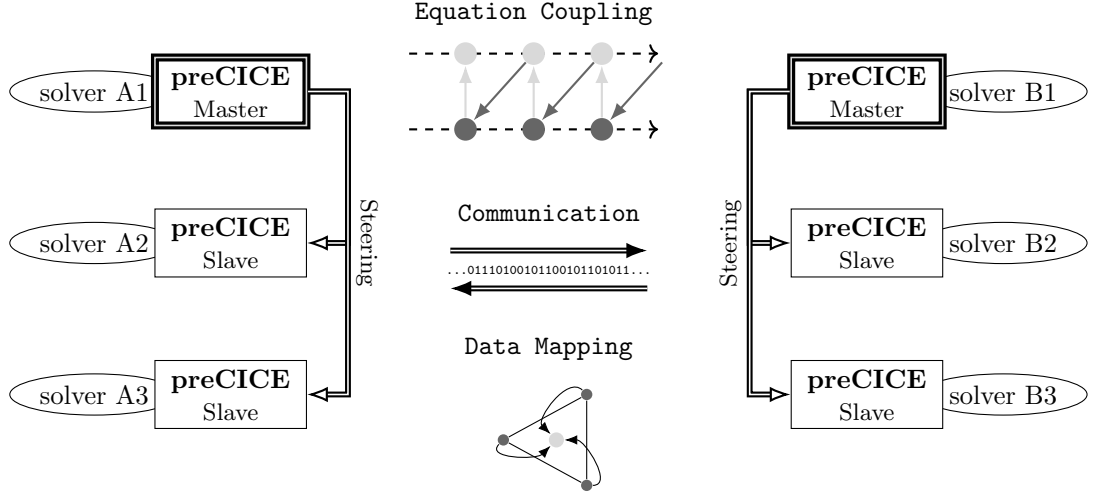
Figure 13: Schematic view of a partitioned multi-physics simulation with two solvers (A and B) coupled through preCICE. In the middle are the three main functionalities of preCICE shown, that steer coupling iterations: Fix-point acceleration methods, point-to-point communication and data mapping between non-matching grids at the coupling interface. Picture courtesy of Florian Lindner [bun]

detail.

5.2 Coupling Methods

preCICE distinguishes two possible coupling schemes: Explicit and implicit coupling. A single time step of multi-physics simulation partitioned into several solvers can be done with only a small and fixed number of calls of time steps per solver. This is described by an explicit coupling scheme. The alternative is an iterative procedure that achieves convergence with respect to the monolithic (all solvers in one program) solution of the system in each time step. An implicit coupling scheme then iterates over a coupling equation until convergence is reached. Based on [bun], it is assumed that two solvers S_1 and S_2 make up a coupled system. Vector spaces X_1 and X_2 describe data at the coupling interface to be mapped between the two solvers in a single time step. The output of S_1 is required as an input by S_2 and vice versa; it follows:

$$S_1 : X_1 \leftarrow X_2 \quad \text{and} \quad S_2 : X_2 \leftarrow X_1 \tag{136}$$

The fluid-structure coupling done in this work, is an example for a Dirichlet-Neumann type coupling. The displacements at the coupling interface of the structure solver is the input for the fluid solver that gives forces to the structure solver as its output.

68

5.2.1 Explicit Coupling Schemes

Two different implementations of explicit coupling schemes exist in preCICE: The staggered scheme and the parallel scheme. The staggered scheme uses the old time step's values $x_1^{(n)}$ at the coupling surface for the execution of the nth time step $(t_n \leftarrow t_{n+1})$ to achieve $x_2^{(n+1)}$. These new values act as boundary values for the nth time step of S_2:

$$x_2^{(n+1)} = S_1^{(n)}\left(x_1^{(n)}\right) \quad \text{and} \quad x_1^{(n+1)} = S_2^{(n)}\left(x_2^{(n+1)}\right) \tag{137}$$

Since the two solvers are executed staggered, this scheme is not optimal with respect to load balancing. In contrast, the parallel scheme does not have this drawback [bun]. It uses the old time step values $x_1^{(n)}$ and $x_2^{(n)}$ as inputs for both solvers:

$$x_2^{(n+1)} = S_1^{(n)}\left(x_1^{(n)}\right) \quad \text{and} \quad x_1^{(n+1)} = S_2^{(n)}\left(x_2^{(n)}\right) \tag{138}$$

According to [bun] both explicit schemes yield consistent time steps but cause instabilities that cannot be avoided, even if the time step length is reduced.

5.2.2 Implicit Coupling Schemes

All implicit coupling schemes in preCICE are based on fixed-point iterations using the staggered or the parallel explicit scheme. The fixed-point iteration regarding the staggered scheme is as follows:

$$x_2^{(n+1),i+1} = S_1^{(n)}\left(x_1^{(n+1),i}\right) \quad \text{and} \quad x_1^{(n+1),i+1} = S_2^{(n)}\left(x_2^{(n+1),i+1}\right) \tag{139}$$

The fixed-point iteration corresponding to the parallel scheme can be written as:

$$x_2^{(n+1),i+1} = S_1^{(n)}\left(x_1^{(n+1),i}\right) \quad \text{and} \quad x_1^{(n+1),i+1} = S_2^{(n)}\left(x_2^{(n+1),i}\right) \tag{140}$$

Here, the new iterates for $x_2^{(n+1),i+1}$ and $x_2^{(n+1),i+1}$ are computed in parallel. preCICE offers simple underrelaxation, adaptive Aitken underrelaxation and various quasi-Newton solvers in order to solve these fixed-point equations as robust and stable as possible [bun]. In the following a generic fixed-point equation

$$x = H(x) \tag{141}$$

is considered. Every fixed-point equation solver in preCICE is a combination of a fixed-point iteration and a post-processing step that modifies the result of the fixed-point iterator. Underrelaxed fixed-point iterations are the simplest solvers [bun]:

$$x^{i+1} = H(x^i) + (\omega - 1)\left(H(x^i) - x^i\right) \tag{142}$$

with $\omega \in \mathbb{R}, 0 < \omega \leqslant 1$ either a user-defined value or additionally adapted by the solver throughout the iterations using Aitken underrelaxation. In order to get the full

possibility of a parallel scheme, preCICE provides a class of convergence acceleration methods, called "quasi-Newton methods". With these methods a fast convergence is achieved in particular for difficult problems when using parallel fixed-point iterations [bun]. All quasi-Newton methods in preCICE accelerate the fixed-point iteration by a subsequent Newton step:

$$x^{k+1} = \underbrace{H(x^k)}_{=:\,\tilde{x}^k} - J_{\tilde{R}}^{-1} \underbrace{\left(H(x^k) - x^k \right)}_{=\,\tilde{x}^k - H^{-1}(\tilde{x}^k)} \tag{143}$$

where $\tilde{R} = I - H^{-1}$ maps $\tilde{x}^k$ to the residual $r^k = \tilde{R}(\tilde{x}^k) = H(x^k) - x^k$. The inverse of the Jacobian $J_{\tilde{R}}^{-1}$ can be approximated ($\hat{J}_{\tilde{R},k}^{-1}$) in different ways in preCICE [bun]:

- The classical interface quasi-Newton (IQN) approach uses the approximated inverse Jacobian with minimal Frobenius norm:

$$\left\| \hat{J}_{\tilde{R},k}^{-1} \right\|_F \leftarrow \min \tag{144}$$

- The multi-vector quasi-Newton (IMVJ) approach minimizes the distance between $\hat{J}_{\tilde{R},k}^{-1}$ and the approximate $\hat{J}_{\tilde{R},k}^{-1,(n)}$ from the last time step:

$$\left\| \hat{J}_{\tilde{R},k}^{-1} - \hat{J}_{\tilde{R},k}^{-1,(n)} \right\|_F \leftarrow \min \tag{145}$$

For more details, see [bun] and [Gat15].

Additional specifications can be made by the user to adapt the coupling method to the problem. A statement of extrapolation in time can be made to get a better initial guess for the next time step solution and with subcycling several small time steps are done in one of the solvers while the others use a larger time step.

5.3 Data Mapping

Due to the fact that independent solvers are used for coupling in this case, it can happen that the single solver meshes do not match each other at the coupling interface. This requires a mapping of data that is sent by the above iterative coupling methods from the coupling surface of one solver domain to the surface of the other solver domain. The case where the two meshes matching each other is not very common when dealing with two distinct solvers and would result in a simple copying of data values for transfer without any interpolation or projection. In the case of non-conforming meshes, interpolation algorithms are necessary and when the two grids overlap or gaps exist even projections in some form are additionally required [Gat15]. preCICE offers some data mapping methods but also provides the possibility for user-defined mapping implementations for special cases [bun].

5.3.1 Conservative vs. Consistent

The mapping of displacements and pressures, for example, usually requires a **consistent mapping**, i.e. constants should be interpolated exactly. Let $\underline{H}_{AB} \in \mathbb{R}^{n_A \times n_B}$ be the matrix mapping values between variables from solver A to B. Then consistent mapping can be expressed as follows:

$$\underline{\zeta}_B = \underline{H}_{AB}\underline{\zeta}_A \tag{146}$$

with the arbitrary variable ζ whose values are represented as matrix $\zeta_{A,i} = \zeta_{B,j} = \zeta = \text{const}, 1 \leqslant i \leqslant n_A, 1 \leqslant j \leqslant n_B$. The property of exact constant interpolation is satisfied if and only if the row sums of the mapping matrix are equal to 1 [Gat15]:

$$\zeta_{B,i} = \sum_{j=1}^{n_A} H_{AB,ij}\zeta_{A,j} = \zeta_{A,j} = \zeta \qquad \forall i \tag{147}$$

A **conservative mapping** on the other hand is important for integral values like forces. This mapping approach requires the sum of the data values to be equal on both sides:

$$\sum_{i=1}^{n_B} \zeta_{B,i} = \sum_{j=1}^{n_A} \zeta_{A,j} \tag{148}$$

This property holds only for the column sums of $\underline{H}_{AB}$ to be equal to 1:

$$\sum_{i=1}^{n_B} \zeta_{B,i} = \sum_{i=1}^{n_B} \left(\underline{H}_{AB}\underline{\zeta}_A\right)_i = \sum_{i=1}^{n_B}\sum_{j=1}^{n_A} H_{AB,ij}\zeta_{A,j} = \sum_{j=1}^{n_A} \left(\sum_{i=1}^{n_B} H_{AB,ij}\right)\zeta_{A,j} = \sum_{j=1}^{n_A} 1\zeta_{A,j} \tag{149}$$

Further details to the mathematical background of the two mapping approaches can be found in [Gat15]. Both mappings are available for all methods described below.

5.3.2 Nearest-Neighbor

The Nearest-Neighbor mapping method works locally and requires only vertex positions, i.e. only the data nodes of the solver surface meshes are required for the mapping. The mapping itself is simple: In order to map values from mesh A to mesh B, for every node of mesh B the geometrically closest neighbor to a node of mesh A needs to be determined. Then, the data value from the closest node in mesh A is copied to the corresponding node in mesh B. There are special cases for this projection method: A value from mesh A can be copied to more than one node of mesh B if the node in mesh A is closest to all of the B nodes. On the other hand, nodes in A can be omitted if no projection partner in B were found, since the search for nearest neighbors is done on mesh B only. This is a general issue of all projection methods [Gat15].

If the meshes have matching vertex positions then this mapping method is an adequate choice. For non-conforming meshes the first order accuracy of Nearest-Neighbor makes it a rather bad choice and other mapping methods should be taken into account [bun].

5.3.3 Nearest-Projection

The Nearest-Projection mapping requires not only vertex positions but also topology information of the source mesh. It searches for mesh nodes on the target mesh and creates geometrical projections to a matching set of elements on the source mesh. An interpolation is employed from nodes to mesh elements and vice versa. The finding of closest neighbors is similar to Nearest-Neighbor, but Nearest-Projection uses faces instead of points for its projections and several data nodes can be involved [Gat15]. Just as for the Nearest-Neighbor mapping, nodes can be omitted here if the mesh widths differ too much locally. This method is also of first order in theory due to the projection step. In practice, due to the fact that often the distance between the two meshes normal to the coupling interface is smaller than the mesh widths, a second order accuracy can be observed [bun].

5.3.4 Radial Basis Function

This mapping method uses radial basis functions (RBF) centered at the mesh nodes of the source mesh. It does not require topological information, projections or search-algorithms. It works well on general non-conforming meshes, where overlapping meshes or gaps between them can occur. Several different basis functions are implemented in preCICE, including Gaussian, (Inverse) Multiquadrics, Thin Plate Splines and Volume Splines (see [bun]); further functions can be added by the user. The global support of some RBFs (Gaussian and Thin Plate, for instance) can be limited to a smaller area by introducing a cut-off radius. This reduces the density of the system matrix for interpolation and thus reduces the amount of communication between the solvers and the computational complexity of the data mapping. This allows local communication while one still benefit from the properties of the radial basis functions [bun].

5.4 Communication

During a multi-physics simulation the different solvers requires an efficient communication between each other. preCICE offers a communication per interaction of two or more participants. Either MPI ports or low level TCP/IP sockets are available for communication in preCICE. A fully parallel point-to-point data transfer is possible, since it analyzes the mesh decomposition of all participants and constructs only local communication channels where they are needed [bun].

5.4.1 MPI Communication

The message passing interface (MPI) is used in preCICE with a multiple program multiple data (MPMD) paradigm, i.e. two different codes are communicating with their own data. Two MPI-based communication methods are implemented which differ in the way how they set up the communication space. With the first method all executables are put into the same communication space, called "communicator world". All processes of each participant are then grouped into separate communicators. An inter-communicator

is used as the actual channel for data exchange. The second method establishes a connection between processes started individually and in different communication spaces. The exchange of connection information is done through a commonly accessible file that stores the port names of the participants. In order to use the port variant, a MPI version of 2 or greater is required [Gat15]. The result is the same as with the first method, namely an inter-communicator.

5.4.2 Socket Communication

The communication by TCP/IP increases compatibility to closed-source software that might be restricted to some MPI implementations [bun]. To abstract from platform dependent socket interfaces like Pthreads or Winsock, the `Boost.Asio` (asynchronous network and low-level input/output) library was used for implementing the socket communication in preCICE [Gat15]. The sending and receiving of data is managed by the transfer control protocol (TCP) which is designed for failsafe data transfer and synchronization between sender and receiver. Communication via TCP is not typical for multi-physics simulations that often run on supercomputers, because it involves a synchronization step and additional communication overhead compared to MPI. Other difficulties are that ports used for socket communication may be blocked by default and each supercomputer node might have a different network address which requires automated checks on a low-level socket layer [Gat15].

5.5 Implementation

The program developed within the scope of this thesis is to be coupled with a fluid solver in a fluid-structure interaction simulation. The coupling is done with the help of preCICE. This section contains the steps that were necessary to adapt the original code to preCICE, including a general preCICE integration example, the introduction of additional boundary conditions, the partitioning of the coupling surface and details on the actual API integration.

5.5.1 preCICE Code Example

One goal of preCICE is that its API can be integrated into existing solver code with minimal modifications to the original solver in order to make it part of a multi-physics coupled simulation. Listing 16 shows an integration example of preCICE into an arbitrary solver code.

Listing 16: preCICE Integration Example

```
1  // ... solver specific initialization steps
2
3  precice::SolverInterface precice("SolverName", solverRank, solverThreadSize);
4  precice.configure("precice-config.xml");
5  int dataID = precice.getDataID("DataName");
6  int* vertexIDs = new int[n_nodes];
7  precice.setMeshVertices(meshID, dataSize, coordinates, vertexIDs);
```

```
8
9   // ... setup solver data structures like forces, displacements, etc.
10
11  double preciceMaxDt = precice.initialize();
12
13  while (precice.isCouplingOngoing())
14  {
15        dt = min(preciceMaxDt, solverDt);
16
17        // ... computer solver time step
18
19        precice.writeBlockVectorData(dataID, dataSize, dataIndices, data);
20        preciceMaxDt = precice.advance(dt);
21        precice.readBlockVectorData(dataID, dataSize, dataIndices, data);
22  }
23
24  precice.finalize();
25  // ... solver specific finalization steps
```

The main parts of the preCICE code can be summarized as follows: The creation of an **SolverInterface** object in line 3 and the loading of the XML-configuration file the line below, the setup of the interface structures (line 5-7), the while-loop that is executed as long as the simulation has not finished (line 13) and the exchange of data in line 19 and 21. The call of the `precice.advance`-function (line 20) executes the preCICE coupling numerics, interpolations and communications, as stated in the previous sections. More details on the single preCICE steps are described in the following when the actual preCICE integration into the thesis' program code is shown.

5.5.2 Additional Boundary Conditions

When the solver is part of a coupled simulation, a part of the mesh (or the whole) is acting as the interface surface to the other side. preCICE uses this coupling interface to exchange the data values between the two sides. In order to be flexible with respect to the fluid solver part, this interface region is to be defined arbitrarily within the mesh file that is imported by the structure solver. The definition which node is part of the coupling interface and which not, is done by a separate boundary condition ID. This guarantees no additional modifications to the mesh import, as only the mesh files itself need to be changed accordingly. Such modified files can even be used without the coupled version of the solver as the added boundary conditions will simply be ignored by the stand-alone version of the solver.

The structure solver accepts two different types of boundary conditions in the unmodified version: Simply supported (ID = 0) and clamped (ID = 1). These are defined at the mesh's boundary in general. With the coupling update three more boundary condition IDs are required:

- **ID 2**: A node with this ID is defined to be part of the preCICE coupling interface. It can be positioned at the mesh's boundary and within the mesh.

- **ID 20**: Such a node has a simply supported boundary condition **and** is additionally part of the coupling interface. This ID is necessary, because the boundary condition is required by the structure solver to work correctly and the coupling interface definition is required by preCICE to exchange data at this node with the fluid solver.

- **ID 21**: Here, the node has a clamped boundary condition **and** is part of the coupling interface. The reason for this ID is the same as for the above.

The new definitions add only two more lines to the original code, as can be seen in Listing 17. LibMesh uses a "set" data structure to be able to assign multiple different IDs to one **DirichletBoundary**-object, which is exactly exploited here.

Listing 17: Additional boundary condition IDs

```
1  // (...)
2  std::set<boundary_id_type> boundary_ids;
3  boundary_ids.insert(0);
4  boundary_ids.insert(20); // NEW
5  // (...)
6  boundary_ids.clear();
7  boundary_ids.insert(1);
8  boundary_ids.insert(21); // NEW
9  // (...)
```

5.5.3 Partitioned Coupling Surface

As stated in the section above, preCICE expects a coupling interface region on the solver's mesh. The definition of this region happens in the mesh file that is imported by the solver at the beginning of the simulation. The solver stores the mesh as an internal libMesh **Mesh**-object. For the interface region, an own grid-like structure needs to be defined that stores the vertex positions of the region nodes. Listing 18 is an extract of the coupled program code. First, an advantage of libMesh and preCICE can be seen in line 3 and 4: The code is usable with and without MPI without any code modifications: With the usage of the prefix "local" for the node iterators, every MPI process only searches its own mesh partition for coupling interface nodes. If no MPI is used, local is equal to global and the only existing process goes through all mesh nodes. On the other hand does preCICE not distinguish between a serial or parallel in the solver programming code - although it handles parallel solvers different to serial solvers internally (see [Gat15]).

Listing 18: Partition Coupling Surface

```
1  // (...) preCICE successfully configured
2  MeshBase::const_node_iterator          no = mesh.local_nodes_begin();
3  const MeshBase::const_node_iterator end_no = mesh.local_nodes_end();
4  int n_nodes = 0;
5  BoundaryInfo info = mesh.get_boundary_info();
```

```
 6  info.build_node_list_from_side_list();
 7  std::vector<const Node*> preCICEnodes;
 8  for (; no != end_no; ++no)
 9  {
10      const Node *nd = *no;
11      if (info.has_boundary_id(nd,2)  ||
12          info.has_boundary_id(nd,20) ||
13          info.has_boundary_id(nd,21))
14      {
15          preCICEnodes.push_back(nd);
16      }
17  }
18  n_nodes = preCICEnodes.size();
19  int dimensions = interface.getDimensions();
20  double* grid = new double[dimensions * n_nodes];
21  // (...)
22  std::vector<const Node*>::iterator iter = preCICEnodes.begin();
23  for (int i = 0; iter != preCICEnodes.end(); ++iter,++i)
24  {
25      const Node *nd = *iter;
26      // (...)
27      grid[i*dimensions]   = (*nd)(0);
28      grid[i*dimensions+1] = (*nd)(1);
29      grid[i*dimensions+2] = (*nd)(2);
30  }
31  // (...)
32  int meshID  = interface.getMeshID("Structure_Nodes");
33  int *vertexIDs = new int[n_nodes];
34  interface.setMeshVertices(meshID, n_nodes, grid, vertexIDs);
```

In line 6 of Listing 18 a **BoundaryInfo**-object is created. This class contains information relevant to boundary conditions: It can mark element faces and nodes with IDs useful for identifying the type of boundary condition. The next line is needed when a mesh file written in the libMesh-format XDR/XDA was imported. In such a file the boundary conditions can only be defined at elements sides. The BoundaryInfo-object then contains only information regarding edges, not nodes. The function called in line 7 applies these information to the mesh nodes, too, such that the boundary condition IDs can be accessed at the nodes. The for-loop between line 9 and 18 goes through all local nodes of the mesh partition. Inside of it, it is checked whether the current node has a boundary condition ID that makes the node part of the coupling interface. If so, a pointer to this node is stored in a vector for further processing. After the for-loop, the number of nodes of the interface part is saved.

When it is known how many and which nodes are part of the coupling interface, the grid-like structure needs to be created and filled. This is done in line 21 and 24 to 31. An array of **double**-values represents the structure. It has `dimensions * n_nodes` entries. preCICE expects the ordering to be as follows: $p_{1_x}, p_{1_y}, p_{1_z}, p_{2_x}, \ldots, p_{n_y}, p_{n_z}$, with p_i being the ith node defined in 3D space. The for-loop beginning in line 24 simply

iterates over the previously stored nodes and assigns their geometrical positions to the corresponding entries in the grid-structure.

preCICE holds an ID for every mesh a solver contributes to the simulation. The ID of the structure mesh is get from preCICE in line 32. Additionally, preCICE holds an internal representation of the vertices of the coupling interface. Every such vertex has a globally (throughout the solvers) unique ID. These IDs may be different to the node IDs of the structure mesh. For the data exchange with preCICE to function properly the IDs must be get from preCICE. This is done in line 33 and 34. According to the single geometric vertex positions, preCICE finds the corresponding internal vertex and returns its ID.

5.5.4 Integration of preCICE

The changes in the mesh import and the creation of the coupling interface partitions were preprocessing steps that laid the foundation of the actual preCICE integration into the solver code. Before any preCICE function can be used, a **SolverInterface**-object of preCICE must be created with the specification of the solver's name, the ID of the calling process and the overall number of processes used for this solver (see Listing 19). Next, the XML-configuration file must be loaded by preCICE.

Listing 19: preCICE Integration Part 1

```
// (...) libMesh Initialization and mesh import
SolverInterface interface(solverName, global_processor_id(), global_n_processors());
interface.configure(config_filename);
```

The structure solver in its current state gets forces as input from the fluid solver and produces displacements as output sent back to the fluid solver. In order to accelerate the data exchange, all data is sent at once and received at once. Therefore, in Listing 20 two arrays are created (line 1 and 2) with as many entries as the process owns nodes of the coupling interface multiplied by the number of dimensions (typically 3). These variables were defined by the XML-configuration file and are present in preCICE, too. For the data exchange their IDs must be get from preCICE, which is done in line 5 and 6. Inside the for-loop that also fills the grid-like structure from Listing 18 with vertex position, initial values for the displacements and forces are set. This must be done by one of the coupled solvers; which one is defined in the XML-configuration file.

Listing 20: preCICE Integration Part 2

```
double *displ = new double[dimensions*n_nodes];
forces = new double[dimensions*n_nodes];
// (...)
int meshID  = interface.getMeshID("Structure_Nodes");
int displID = interface.getDataID("Displacements", meshID);
int forceID = interface.getDataID("Forces", meshID);
// (...)
for (int i = 0 ; iter != preCICEnodes.end(); ++iter,++i)
{
```

```
10          const Node *nd = *iter;
11          for (int dims = 0; dims < dimensions; dims++)
12          {
13                  displ[i*dimensions+dims]  = 0.0;
14                  forces[i*dimensions+dims] = 0.0;
15          }
16          // (...)
17  }
```

As stated in Section 5.5.3, the vertex IDs of preCICE does not necessarily match the IDs of the structure's mesh node IDs. To be able to apply the correct forces to the correct mesh nodes, a mapping is necessary. Listing 21 displays this in the lines 1 to 6: A for-loop iterates over the coupling interface nodes and stores the linking of the mesh nodes IDs to the preCICE vertex IDs in a **unordered_map**-data structure, which is efficient in adding elements and finding them later. The **id_map** is used when the right-hand side of the single elements is constructed and is described in further detail in Listing 24.

Listing 21: preCICE Integration Part 3

```
1   iter = preCICEnodes.begin();
2   for (int i = 0 ; iter != preCICEnodes.end(); ++iter,++i)
3   {
4           std::pair<dof_id_type, int> pair( (*iter)->id(), vertexIDs[i] );
5           id_map.insert(pair);
6   }
7
8   interface.initialize();
9   if ( interface.isActionRequired(actionWriteInitialData()) )
10  {
11          interface.writeBlockVectorData(displID, n_nodes, vertexIDs, displ);
12          interface.fulfilledAction(actionWriteInitialData());
13  }
14  interface.initializeData();
15  if ( interface.isReadDataAvailable() )
16          interface.readBlockVectorData(forceID, n_nodes, vertexIDs, forces);
17
18  // (...) libMesh equation system setup etc.
```

With line 8 preCICE gets finally initialized. In line 9 to 13 the initial data set in Listing 20) is sent to preCICE. The call in line 14 tells preCICE to communicate all initial data to every solver, including the calling one. In line 16 possible available initial force values are read.

Listing 22 contains the main while-loop that is executed until the complete simulation is finished (`isCouplingOngoing()`) or a critical error happened which quits the simulation, too. The lines 3 and 4 are only required when an implicit coupling scheme was chosen in the preCICE configuration. This requires so-called "Checkpoints" to be created. Because the coupled solvers are only sub-components of the overall simulation, they cannot access all information needed to control the simulation. preCICE is the control

78

instance that decides what kind of computation step has to be performed next by a solver. This includes the responsibility for synchronized writing of outputs when a time step convergenced or restarting a simulation step. For this, the old time step state needs to be saved and this is realized by checkpoints. When a checkpoints is written to or read from is managed by preCICE and the solver only does so when the corresponding action is required.

In line 6 the linear elastic problem with the given mesh and forces is solved by the program. As addressed in Section 4.2.5, the solution is only accessible for the master process. In the stand-alone version of the program this was not a problem, because the master process can write the solution into a file or put it onto the console. When the program is part of a coupled simulation, every single process must tell preCICE its own displacement solutions. After building the solution vector in line 9, a broadcasting of the results must be performed: Every slave process reserves enough space in its solution vector to store the data. The master process broadcasts its solution to the slave processes. If the program is not executed in parallel this step is ignored through the two if-clauses. When every process got the solution, they can assign their displacements to the array that is used for data exchange.

Listing 22: preCICE main while-loop part 1

```
while ( interface.isCouplingOngoing() )
{
    if ( interface.isActionRequired(actionWriteIterationCheckpoint()) )
        interface.fulfilledAction(actionWriteIterationCheckpoint());

    equation_systems.solve();

    std::vector<Number> sols;
    equation_systems.build_solution_vector(sols);
    if (global_processor_id() > 0)
        sols.reserve(mesh.n_nodes()*6);
    if (global_n_processors() > 1)
        mesh.comm().broadcast(sols);

    std::vector<const Node*>::iterator iter = preCICEnodes.begin();
    for (int i = 0 ; iter != preCICEnodes.end(); ++iter,++i)
    {
        int id = (*iter)->id();
        displ[i*dimensions]   = sols[6*id];
        displ[i*dimensions+1] = sols[6*id+1];
        displ[i*dimensions+2] = sols[6*id+2];
    }
```

The actual data exchange is taken place in the lines 23 and 25. First, the solver writes its solution to preCICE. Then the advance-function of preCICE is called with a parameter that represents the actually used time step length of the solver. The function updates the coupling state, maps data between non-matching grids and exchanges data between the coupled solvers. Further, it applies iteration acceleration schemes (cf. 5.2.2) and

measures the convergence of coupling iterations when implicit coupling is used (like in this case) [Gat15]. In line 25 the new force values are read and assigned to the corresponding array.

Listing 23: preCICE main while-loop part 2

```
23    interface.writeBlockVectorData(displID, n_nodes, vertexIDs, displ);
24    interface.advance(dt);
25    interface.readBlockVectorData(forceID, n_nodes, vertexIDs, forces);
26
27    if (interface.isActionRequired(actionReadIterationCheckpoint()))
28          interface.fulfilledAction(actionReadIterationCheckpoint());
29    else
30    {
31          t++;
32          if (global_processor_id() == 0)
33          {
34                MeshBase::const_node_iterator no = mesh.nodes_begin();
35                const MeshBase::const_node_iterator end_no = mesh.nodes_end();
36                for (int i = 0 ; no != end_no; ++no,++i)
37                {
38                      Node *nd = *no;
39                      (*nd)(0) += sols[6*i];
40                      (*nd)(1) += sols[6*i+1];
41                      (*nd)(2) += sols[6*i+2];
42                }
43          }
44          writeOutput(mesh, equation_systems, t);
45    }
46 }
47 interface.finalize();
```

The if-clause in line 27 checks whether the coupling iteration has already converged. This is the case when **no** action for reading the iteration checkpoint is required. If the iteration has converged then the else-part beginning from line 29 becomes active. Here, the current time step is finally finished by increasing the local time variable (line 31) and the displacement solution gets applied to the program's own mesh (lines 32 to 43). This is done by the master-process only, iterating over the complete collection of mesh nodes.

For every finished time step an output file is written (line 45), as long as the user put the corresponding argument to the command line. Line 46 is the closing bracket of the big while-loop. If the program proceeds to this point, the simulation was finished, either correctly or quit due to errors. In both cases the solver interface to preCICE must be correctly finalized, which is done by calling the appropriate function in line 47.

In Listing 21 a data structure was created that stores mappings from libMesh node IDs to preCICE vertex IDs. These mappings are necessary because the vertex ID of preCICE and the ID of the mesh node at the corresponding geometrical position does not has to be equal. When the right-hand side for a single element is generated, the

forces received from the fluid solver are processed. Listing 24 shows mostly the same code as in the stand-alone version of the solver. New are the lines 9 and 10 where the preCICE-ID linked with the mesh node "`id`" is to be found in the map. If such an entry is found then the resulting iterator to this entry is valid and the forces at the corresponding position in the force-array can be assigned to the right-hand side vector.

Listing 24: Modification of contribRHS-function

```
1  // (...)
2  Node* node = (*elem)->get_node(side);
3  dof_id_type id = node->id();
4  // (...)
5  if (processedNodes->find(id) == processedNodes->end())
6  {
7        processedNodes->insert(id);
8
9        std::unordered_map<dof_id_type,int>::const_iterator preCICE_id = id_map.find(id);
10       if (preCICE_id != id_map.end())
11       {
12             arg(0) = forces[preCICE_id->second*3];
13             arg(1) = forces[preCICE_id->second*3+1];
14             arg(2) = forces[preCICE_id->second*3+2];
15       }
16  }
17  // (...)
```

The actual integration of preCICE into the solver's code required not significantly more lines of code than shown in the example in Listing 16. The use of implicit coupling made it necessary to work with checkpoints that would not have been used with an explicit coupling scheme and added a few more lines of code. Nevertheless, the goal of preCICE to be minimal invasive with respect to integration into existing code was confirmed in this particular case.

6 Validation

The developed program implemented flat shell elements. Two different discretizations of shell elements are provided: A three-node triangular element (Tri-3) and a four-node quadrilateral element (Quad-4). A plane and plate bending element were superimposed to the final shell element. This chapter attends the validation with respect to accuracy and convergence properties of the two shell element discretizations as well as their components, the plane and plate elements. Several example problems were chosen that show the correctness of the finite element types. The tests were taken from different sources, namely [Kan04], [MH85], [Wil96] and [Jin94]. Many example problems have analytical results to be compared with, the examples from [Kan04] used a commercial software called *SAP2000* [sap00] as a benchmark for comparison. Since this software is used in practice for over 30 years, it will be used here as a benchmark as well.

6.1 Test A: Tri-3 Plane Element Displacement

The three-node triangular plane element **Tri-3** is validated with a cantilever beam shown in Figure 14. The example problem was taken from [Kan04] (Test Example 2).
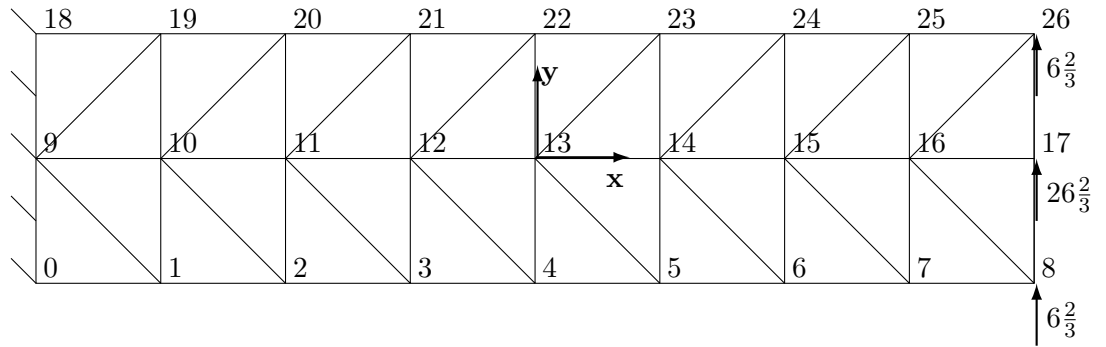

Figure 14: Cantilever beam consisting of 32 triangular elements, clamped at the left side and a total force of 40 kips applied in positive y-direction at the right side. The numbers next to the elements denote the node IDs.

- **Mesh dimensions**
  Length $l = 48 \ in$
  Depth $h = 12 \ in$
  Thickness $t = 1 \ in$

- **Material properties**
  Young's Modulus $E = 30000 ksi$
  Poisson's ratio $\nu = 0.25$

- **Boundary conditions**
  Clamped boundary conditions at node 0, 9 and 18, i.e. left side of the cantilever beam.

- **Loading**
  A concentrated load of $40kips$ in total. Node 8 and 26 has a load of $6\frac{2}{3}$, node 17 has a load of $26\frac{2}{3}$.

Results: The displacements in x- and y-direction at node 22 and 26 are presented in Table 1 together with the results from the *SAP2000* software showed in [Kan04]. The displacements of the thesis' program deviate from the commercial software in all cases for at most 0.027%. The triangle orientation in the example mesh contains both, the square diagonal facing the upper right as well as the lower right corner of the square. To show that the mixed usage of these orientation types increases the accuracy, two additional tests were made, one with triangles only having their hypotenuse facing the upper right corner of the square (▧) and one with triangles of the other type (◹) only. The results of these tests can only be compared to the first test results of the program, since no benchmark values are available. What can first be seen is that either of the new variants is less accurate than the mixed version. Second, the ▧-variant is more accurate than the ◹-variant and third, the accuracy in x-direction is better than in y-direction, particularly for the ◹-orientation.

Node	Displacement	Results from program[a]	Results from SAP2000[b]	Difference (%)
22	u_x	-0.0255988	-0.025605	0.024%[b]
	u_y	0.0629549	0.062971	0.026%[b]
26	u_x	-0.0342621	-0.034271	0.027%[b]
	u_y	0.1944070	0.194456	0.025%[b]
▧ 22	u_x	-0.0243863	-	4.97%[a]
	u_y	0.0552195	-	14.01%[a]
▧ 26	u_x	-0.0328891	-	4.17%[a]
	u_y	0.1829420	-	6.27%[a]
◹ 22	u_x	-0.0235617	-	8.65%[a]
	u_y	0.0440028	-	43.07%[a]
◹ 26	u_x	-0.0322955	-	6.09%[a]
	u_y	0.1564130	-	24.29%[a]

Table 1: Displacements and deviations for Test A

6.2 Test B: Quad-4 Plane Element Displacement

The four-node quadrilateral plane element **Quad-4** is validated with the same cantilever beam that was used in Test A. The mesh configuration is shown in Figure 15. The

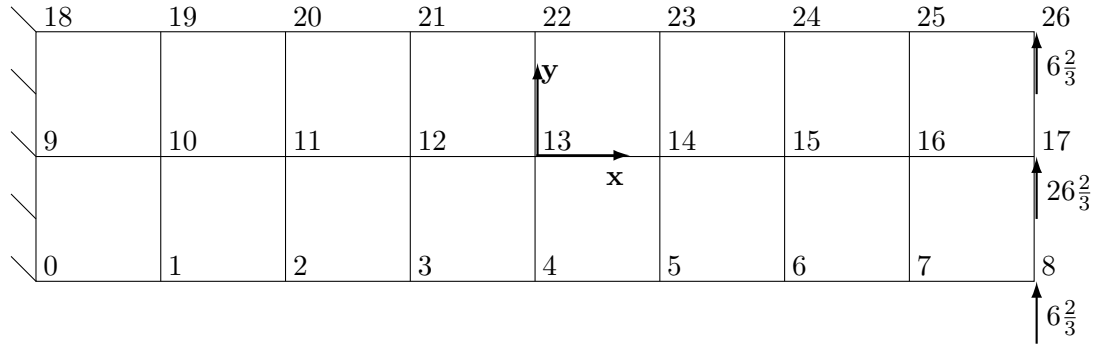example problem was also taken from [Kan04] (Test Example 5).

Figure 15: Cantilever beam consisting of 16 quadrilateral elements, clamped at the left
side and a total force of 40 kips applied in positive y-direction at the right
side. The numbers next to the elements denote the node IDs.

- **Mesh dimensions**
  Length $l = 48\ in$
  Depth $h = 12\ in$
  Thickness $t = 1\ in$

- **Material properties**
  Young's Modulus $E = 30000ksi$
  Poisson's ratio $\nu = 0.25$

- **Boundary conditions**
  Clamped boundary conditions at node 0, 9 and 18, i.e. left side of the cantilever
  beam.

- **Loading**
  A concentrated load of $40kips$ in total. Node 8 and 26 has a load of $6\frac{2}{3}$, node 17
  has a load of $26\frac{2}{3}$.

Results: The displacements in x- and y-direction at node 22 and 26 are presented
in Table 2. The results of *SAP2000* were used for comparison. The displacements
calculated by the thesis' program deviate from the commercial software in all cases for
at most 0.03%.

6.3 Test C: Tri-3 Plate Element Displacement

In this test the three-node triangular plate element **Tri-3** was validated. The example
problem was taken from [Wil96]. The geometry of the mesh is shown in Figure 16. Four
different tests were made: At first the square plate were subdivided into 4×4 squares.

Node	Displacement	Results from program	Results from SAP-2000	Difference (%)
22	u_x	-0.0427728	-0.042774	0.028%
	u_y	0.1012620	0.101265	0.030%
26	u_x	-0.0570728	-0.057074	0.021%
	u_y	0.3160560	0.316064	0.025%

Table 2: Displacements and deviations for Test B

Each square were divided into two triangles by a diagonal. Here, two possibilities are given: From the upper left to the lower right (◳) corner or from the upper right to the lower left (◲) corner. The former variant is shown in Figure 16. The other two tests were made with the same triangle variants but with a 16×16 mesh subdivision.
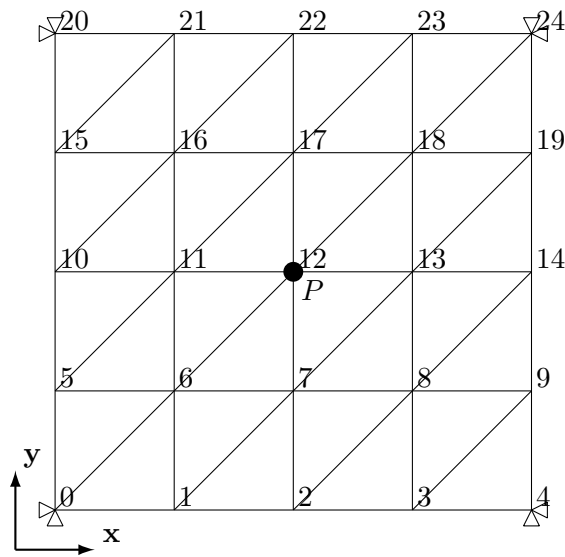

Figure 16: A square plate simply supported on all four sides with a mesh subdivision of 4×4 squares. Each square is divided into two triangles through a diagonal going from the lower left to the upper right corner ("◲"). In the center of the plate, a concentrated load P is applied. The numbers denote the node IDs as defined in the mesh file.

- **Mesh dimensions**
  Side length $l = 10.0$
  Thickness $t = 1.0$

- **Material properties**
  Young's Modulus $E = 10.92$

85

Poisson's ratio $\nu = 0.3$

- **Boundary conditions**
  All sides of the square plate are simply supported.

- **Loading**
  A concentrated load of 1.0 is applied on the center node of the square.

Results: Wilson [Wil96] states that the exact thin-plate displacement of the central node for this problem is $w_c = 1.16$. The results of the test is presented in Table 3. The program's result is compared to the exact value as well as to the results of the Dircrete Kirchhoff Element (DKE) presented in [Wil96]. The DKE is also a three-node triangular plate bending element. At first, the results show that no difference exists between the different orientations of the triangles. For the 4×4 mesh subdivision the difference between the program's result and the benchmark is 10.69% while it is only 8.69% compared with the exact value. For the 16×16 mesh subdivision the difference shrinks to only 0.72% compared to the exact value and 0.97% to the benchmark.

Mesh variant	Displacement at center node	Results from program	Results of DKE	Difference to 1.16 (%)
4×4 mesh subdivision				
◿	$w_{c_{12}}$	1.06723	1.195	8.69%
◺	$w_{c_{12}}$	1.06723	1.195	8.69%
16×16 mesh subdivision				
◿	$w_{c_{144}}$	1.15169	1.163	0.72%
◺	$w_{c_{144}}$	1.15169	1.163	0.72%

Table 3: Displacements and deviations for Test C

6.4 Test D: Quad-4 Plate Element Displacement

This example problem tests the four-node quadrilateral plate element **Quad-4**. The test was taken from [Jin94]. Several different configurations were made: The square plate was subdivided into 4, 8 and 16 square elements on each direction. Each mesh subdivision was then tested with a concentrated load of 30000 applied on the central node and with a uniformly distributed load of 300 per square unit. The geometry of one of the six test configurations is shown in Figure 17.

- **Mesh dimensions**
  Side length $a = 10.0$
  Thickness $t = 0.5$

- **Material properties**
  Young's Modulus $E = 10^7$
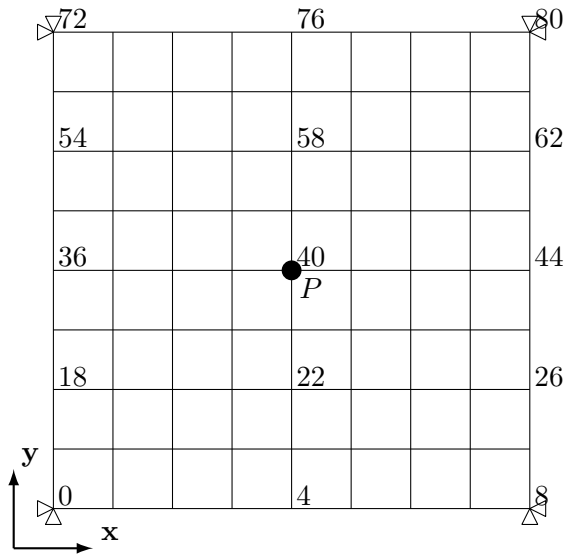 Poisson's ratio $\nu = 0.3$

Figure 17: The mesh of test D is shown with a subdivision level of 8×8 and concentrated central loading configuration. The mesh can be subdivided coarser or finer and the loading can be applied uniformly over the whole plate. The square plate is in all cases simply supported on all four sides.

- **Boundary conditions**

  All sides of the square plate are simply supported.

- **Loading**

  A uniform load of 300 is applied on the whole plate in a first test, while a concentrated load of 30000 is applied on the center node of the square in a second test.

Results: For the test with the uniform load, [Jin94] states that the displacement of the central node w_c^* can be evaluated exactly with the help of the plate theory, that is:

$$w_c^* = \alpha \frac{q_0 a^4}{D} \tag{150}$$

With $\alpha = 0.00406$ and $q_0 = 300$ being the uniform load, $a = 10$ the length of the square plate and $D = \frac{Et^3}{12(1-\nu^2)}$ the material property, the central node's displacement follows as:

$$w_c^* = 0.00406 \frac{300 * 10^4}{\frac{10^7 0.5^3}{12(1-0.3^2)}} = 0.1064045$$

The results with the different mesh subdivision levels are very accurate, the 8×8-subdivision is less than a thousands percents different to the analytical solution and even the coarse level with only 4 elements per direction has only 0.35% difference to the exact value.

For the concentrated load test, [Jin94] also proposes an analytic solution, namely:

$$w_c^* = \alpha \frac{Pa^2}{D} \tag{151}$$

where $\alpha = 0.0115999$ and $P = 30000$ is the concentrated load at the center node. The rest of the variables stays the same as in the uniform load test. The analytical solution to this example problem is:

$$w_c^* = 0.0115999 \frac{30000 * 10^2}{\frac{10^7 0.5^3}{12(1-0.3^2)}} = 0.30401019$$

The program's results are far more inaccurate compared to the uniform load test with a maximum difference of 8.62%. The fine subdivision of 16×16 elements is though acceptably accurate with less than a percent difference to the analytically exact value.

6.5 Test E: Shell Elements Displacement

In Test A to D the single components of the Tri-3 and Quad-4 shell elements were validated. This test shows the accuracy of the shell elements by an example problem taken from [Kan04]. An I-beam is clamped on one end while loads are applied at the flanges of the other end's corners. Figure 18 shows details on the mesh geometry. The force causes the beam to get twisted around the global x-axis. The I-beam will have deformations in every direction for which shell elements are qualified. The Tri-3 element is validated as well as the Quad-4 with the same geometry and load configuration.

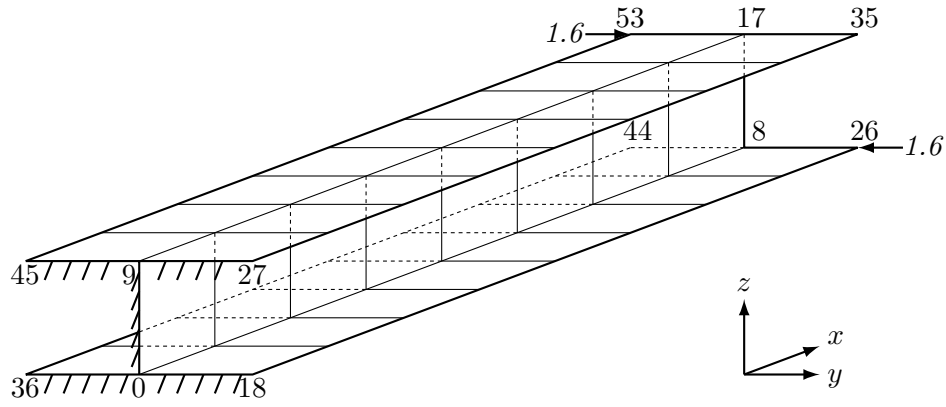| Mesh sub-divisions	Displacement at center node	Results from program	Analytical solution	Difference (%)
Uniform loading				
4×4	$w_{c_{12}}$	0.106032		0.35%
8×8	$w_{c_{40}}$	0.106405	0.1064045	0.00047%
16×16	$w_{c_{144}}$	0.106454		0.046%
Concentrated loading				
4×4	$w_{c_{12}}$	0.332677		8.62%
8×8	$w_{c_{40}}$	0.312851	0.30401019	2.83%
16×16	$w_{c_{144}}$	0.306664		0.86%

Table 4: Displacements and deviations for Test D

Figure 18: The I-beam from Test E in a quadrilateral discretization. The I-beam is clamped at the left side while two point loads of value 1.6 are applied at the other end's corners of the flanges in opposite directions.

- **Mesh dimensions**
  I-beam length $l = 40.0$
  I-beam depth $d = 10.0$
  I-beam height $h = 5.0$
  Thickness $t = 0.25$

- **Material properties**
  Young's Modulus $E = 10000$
  Poisson's ratio $\nu = 0.3$

- **Boundary conditions**
  The left end of the I-beam is clamped, i.e. nodes 0, 9, 18, 27, 36 and 45.

- **Loading**
  Two separate loads of 1.6 are applied on the flanges of the non-clamped end's corner. One on the top of the I-beam (node 53), the other on the bottom mirrored at the xz-plane (node 26). The forces pointing horizontally to the middle of the I-beam, cf. 18.

Results: In comparison to the commercial software SAP2000 shown in Table 5 the thesis' elements does not provide as good accuracy as can be. The Tri-3 element has a difference of at most 3.31% for the planar displacements u_x and u_y and at most 5.67% for the vertical displacement u_z. The Quad-4 element is with about 10% difference less accurate.

Node	Displacement	Results from program	Results from SAP2000	Difference (%)
Triangular element				
35	u_x	-0.0152698	-0.014921	2.34%
	u_y	0.0879212	0.085471	2.87%
	u_z	0.1543450	0.146070	5.67%
44	u_x	-0.0153249	-0.014834	3.31%
	u_y	-0.0878749	-0.085475	2.81%
	u_z	-0.1518410	-0.144533	5.06%
Quadrilateral element				
35	u_x	-0.0246001	-0.027162	10.41%
	u_y	0.1373650	0.151049	9.96%
	u_z	0.2320180	0.255308	10.04%
44	u_x	-0.0246001	-0.027162	10.41%
	u_y	-0.1373650	-0.151049	9.96%
	u_z	-0.2320180	-0.255308	10.04%

Table 5: Displacements and deviations for Test E

6.6 Test F: Convergence (increasing number of elements)

In this test an example problem from [MH85] is used. The convergence of the Quad-4 element is validated as an example. For this, the number of elements of a rectangular plate is increased from one test to the other. The mesh has four different configurations: A clamped and simply supported boundary condition variant as well as a uniform and a concentrated loading at the central node of the mesh. Each of the four configurations are tested with a mesh subdivision level of $n = 2, 4, 8, 16, 32, 64$, i.e. the mesh is subdivided into n elements in each direction. An example for a 4×4 subdivision can be seen in Figure 19.
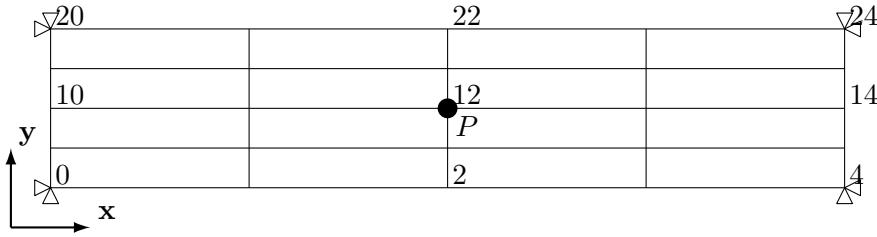

Figure 19: Example of the mesh geometry of Test F: A rectangular simply supported plate with a side length ratio of $1 : 5$. In this configuration a concentrated load P is applied to the center node (12) and the mesh is subdivided into 4 elements per axis. For better visibility, only some of the node IDs are drawn.

- **Mesh dimensions**
  Plate long side length $a = 10.0$
  Plate short side length $b = 2.0$
  Thickness $t = 0.01$

- **Material properties**
  Young's Modulus $E = 1.7472 \times 10^7$
  Poisson's ratio $\nu = 0.3$

- **Boundary conditions**
  SPL-configuration: All sides of the plate are simply supported.
  CLA-configuration: All sides of the plate are clamped.

- **Loading**
  UNI-configuration: A uniform load of 10^{-4} per square unit is applied to the entire plate.
  CON-configuration: A concentrated load of 4×10^{-4} is applied to the center node of the plate. For the different subdivision levels, that is in increasing order the node with ID 4, 12, 40, 144, 544 and 2112.

Results: For the analytical solution to these problems one can use equation (150) for the uniform loadings and (151) for the concentrated loadings, that is:

$$D = \frac{Et^3}{12\left(1 - \nu^2\right)} = 1.6$$

For the four configurations the following exact solutions can be calculated:

$$w_c^* = \alpha \frac{q_0 a^4}{D} = 0.01297 \frac{10^{-4} * 2^4}{1.6} = 12.97 \times 10^{-6} \quad \text{(simply supported, uniform load)}$$

$$w_c^* = \alpha \frac{q_0 a^4}{D} = 0.00256 \frac{10^{-4} * 2^4}{1.6} = 2.56 \times 10^{-6} \quad \text{(clamped, uniform load)}$$

$$w_c^* = \alpha \frac{P a^2}{D} = 0.01695 \frac{4 * 10^{-4} * 2^2}{1.6} = 16.95 \times 10^{-6} \quad \text{(simply supported, concentr. load)}$$

$$w_c^* = \alpha \frac{P a^2}{D} = 0.00725 \frac{4 * 10^{-4} * 2^2}{1.6} = 7.25 \times 10^{-6} \quad \text{(clamped concentr. load)}$$

These values are the benchmarks that were compared with the program's solutions. The results are displayed in Table 6. A subdivision of only 2 elements on every axis leads to a very poor accuracy that lies in the range from 11.03% to 57.81% difference to the exact value. The accuracy gets better for each increase in the subdivision level with two exceptions: The accuracy for the tests with a uniform loading and clamped boundary conditions stuck at an accuracy of about 1.7%. The other exception is the 4×4 subdivision of the simply supported plate with concentrated loading: The accuracy is much better than the following and a refinement of 32 is the first mesh that has better accuracy. Both phenomena cannot be explained so far, since it is neither caused by the type of boundary condition nor by the type of loading. Otherwise the other two configurations would not show as expected: A monotonically increasing accuracy with the level of refinement. What can be said is that with a mesh subdivided into 64×64 elements, the analytically exact value is approximated well with nearly all tests below 1% of difference to it.

One difference between the two boundary conditions can be seen in the results: The simply supported tests are more and faster accurate than the ones with clamped boundary condition. This is due to the discretization of the element: The Quad-4 element were designed with Kirchhoff's plate theory. This ignored transverse shear deformation. When the plate is simply supported the elements at the boundary can receive twists while the clamped elements cannot get any deformation. This increase of stiffness at the boundary cannot be simulated good enough with the thin-plate theory of Kirchhoff. With increasing number of elements the boundary part of the mesh gets smaller and so the bias are weighted less in the overall result.

6.7 Test G: MPI (increasing number of processes)

This test aims to show the benefits of a parallel execution of the thesis' program with respect to runtime optimization. As example problem the same mesh as in Test D

Boundary condition	Displacement	Results from program (10^{-6})	Analytical result (10^{-6})	Difference (%)
Uniform Loading				
SPL	w_{c_4}	14.4005		11.03%
	$w_{c_{12}}$	12.6269		2.65%
	$w_{c_{40}}$	12.8565	12.97	0.88%
	$w_{c_{144}}$	12.9431		0.21%
	$w_{c_{544}}$	12.9640		0.05%
	$w_{c_{2112}}$	12.9691		0.0069%
CLA	w_{c_4}	3.82366		47.06%
	$w_{c_{12}}$	2.45355		4.16%
	$w_{c_{40}}$	2.60137	2.56	1.62%
	$w_{c_{144}}$	2.60384		1.71%
	$w_{c_{544}}$	2.60414		1.72%
	$w_{c_{2112}}$	2.60420		1.73%
Concentrated Loading				
SPL	w_{c_4}	11.5204		32.03%
	$w_{c_{12}}$	17.3048		2.09%
	$w_{c_{40}}$	18.1158	16.95	6.88%
	$w_{c_{144}}$	17.4961		3.22%
	$w_{c_{544}}$	17.1495		1.18%
	$w_{c_{2112}}$	17.0215		0.42%
CLA	w_{c_4}	3.05893		57.81%
	$w_{c_{12}}$	6.06564		16.34%
	$w_{c_{40}}$	7.78902	7.25	7.43%
	$w_{c_{144}}$	7.66573		5.73%
	$w_{c_{544}}$	7.40552		2.14%
	$w_{c_{2112}}$	7.29681		0.65%

Table 6: Displacements and deviations for Test F

(6.4 was used, too. A refinement of 64×64 elements was chosen in the test with the Quad-4 element. For the Tri-3 element test, every such square element were divided at its diagonal creating two triangular elements. The testing machine's hardware was as follows: Intel Core i3 CPU M380 @ 2.53GHz $\times$ 4 (Dual-Core + Intel Hyper-Threading), 4 GB RAM, 64-bit ubuntu 14.04 LTS. It was tested with 1, 2, 3 and 4 processes. In order to measure the time, the built-in performance logging from libMesh was included in the program. It collects data concerning the number a specified part of the code is processed and the time it takes for this. Three time measurements were made for this test: First, the overall execution time from the beginning of the program to the last line of code. Second, the time needed to assemble the system matrix and right-hand side and third, the time the system solver needs to calculate the solution. What is not explicitly measured, is the time for the initialization and mesh importing step, the setup of the system, the applying of the resulting displacements to the mesh and the writing of the output file. These parts are basically independent from the parallelization. While the solving step is done by libMesh, or more precisely by PETSc which is used in parallel execution (cf. Section 4.3.1), the assembly code is program-specific.

- **Mesh dimensions**
  Side length $a = 10.0$
  Thickness $t = 0.5$

- **Material properties**
  Young's Modulus $E = 10^7$
  Poisson's ratio $\nu = 0.3$

- **Boundary conditions**
  All sides of the plate are simply supported.

- **Loading**
  A concentrated load of 30000 is applied to the center node of the plate.

Results: The exact solution to this problem is the same as in 6.4: $w_c^* = 0.1064045$. Independently from the number of processes the solutions are the same for each of the two elements, namely: 0.106465 for Quad-4 (0.056% difference) and 0.106413 for Tri-3 (0.008% difference). Each of the four test runs for each element was repeated five times. Thus, a minimum, a maximum and an average time followed from each test series. The results are illustrated in Table 7 and visualized as graphs in Figure 21, 20 and 22.

What can be seen without further analysis is that the runtime for every studied code part is shorter when using multiple processes in comparison to the sequential execution. Although the mesh for the Quad-4 and Tri-3 was mainly the same, the test with the triangular element has twice as much elements and the structure of the two system matrices differ. Therefore the time for solving the system with the Quad-4 element cannot be compared to the time for solving the Tri-3-system. The times for both elements were nonetheless combined in the resulting graphs. The reduction in time for the solving step by using multiple processes is largely dependent on the programming of PETSc and

cannot be reasoned, but only displayed in the framework of this thesis. One can see a benefit of using multiple processes: Every time measurement for tests with multiple processes is shorter than the serial execution test. A phenomenon that can be seen in all graphs is an relative increase in time for three processes compared to two processes. This might be due to the processor configuration used for the tests: It is a dual-core processor that uses hyper-threading. When using four processes the hyper-threading is beneficial: The two logical cores further reduce the execution time for each code part. The improvement is not as much as if four physical cores would be used, as the hyper-threading technology can only enhance the efficiency of a core and not replace an extra physical processor.

The biggest improvement in performance can be seen for the assembly time. Here, two processes nearly halved the time required for every element to contribute to the system matrix and the system right-hand side. The two elements differ slightly in time, mostly due to the complexer mathematics for the Quad-4 element (cf. 3.3.2 and 3.3.3) and the one extra node to be considered.

No. processes	Assembly time (s)			Solver time (s)			Overall time (s)		
	min	avg	max	min	avg	max	min	avg	max
Tri-3 element									
1	1.93	1.97	2.09	49.35	49.78	50.96	51.57	52.52	53.61
2	0.98	0.98	0.99	35.65	36.48	37.36	38.39	39.24	40.14
3	1.02	1.04	1.05	40.47	41.13	41.57	43.67	44.33	44.23
4	0.79	0.80	0.82	33.31	34.29	34.93	36.27	37.26	37.90
Quad-4 element									
1	1.99	2.01	2.04	31.88	32.25	33.29	34.43	34.83	35.94
2	1.03	1.03	1.03	23.67	24.11	24.75	26.21	26.65	27.28
3	1.01	1.08	1.09	26.05	26.59	26.95	28.79	29.36	29.88
4	0.83	0.84	0.85	21.53	22.12	23.05	24.14	24.73	25.66

Table 7: Time measurements for Test G

6.8 Test H: Coupled "Bending Tower"

The developed program is tested in a fluid-structure interaction. In the simulation the program gets coupled with a fluid solver developed with OpenFOAM, an open-source software for computational fluid dynamics [Fou]. As example problem a simple channel flow around a tower-like obstacle is simulated. The geometry configuration can be seen in Figure 23. The fluid solver simulates the flow coming from the left side of the channel and moving towards the right side. The tower is placed at the bottom side of the channel and has a simply supported boundary condition at this edge. It is simulated by the thesis' program. When the flow reaches the left side of the tower, it will bend to the right side due to the force steadily increasing by the flow. The coupling interface region in this test is the left, top and right edge of the tower.
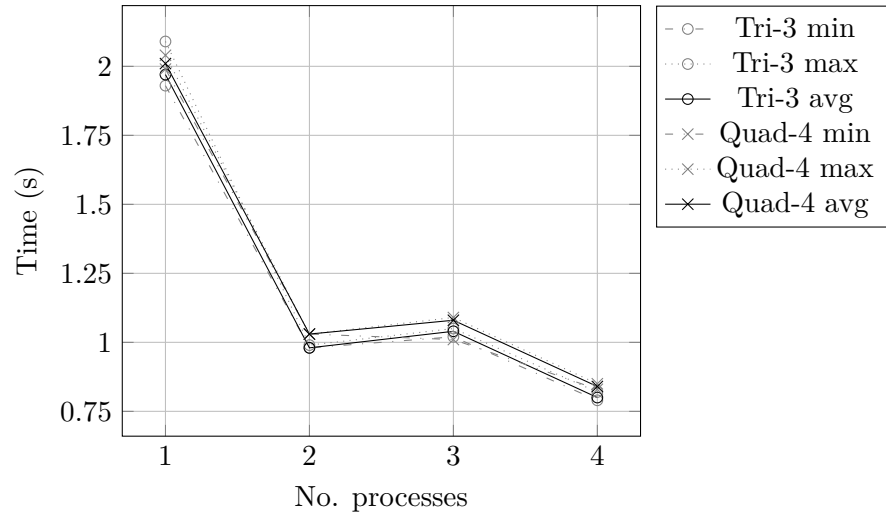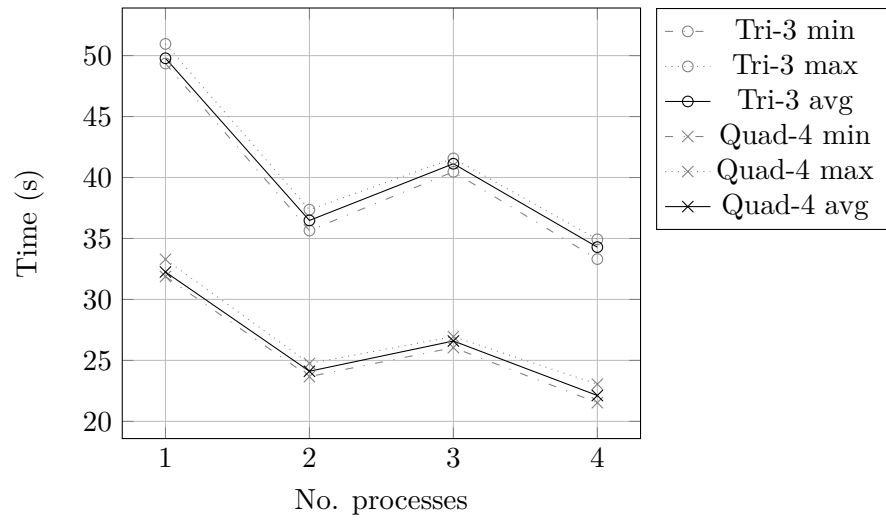
Figure 20: Assembly Times
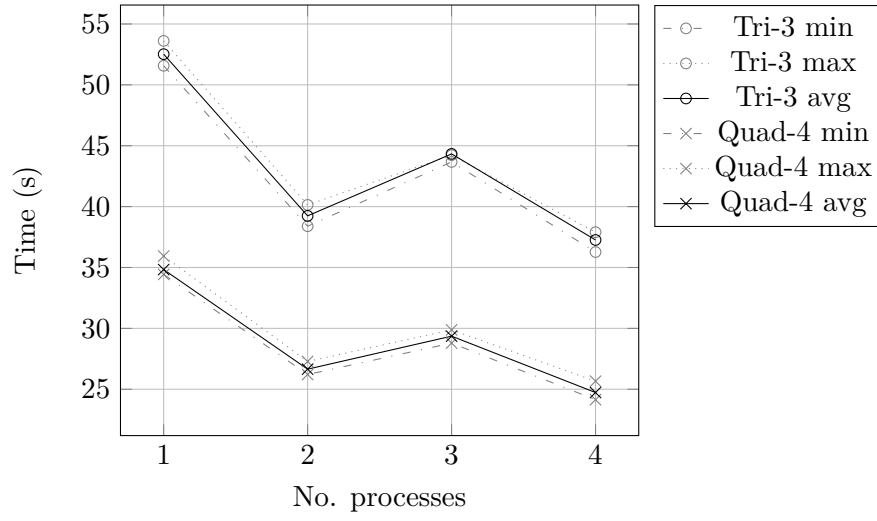

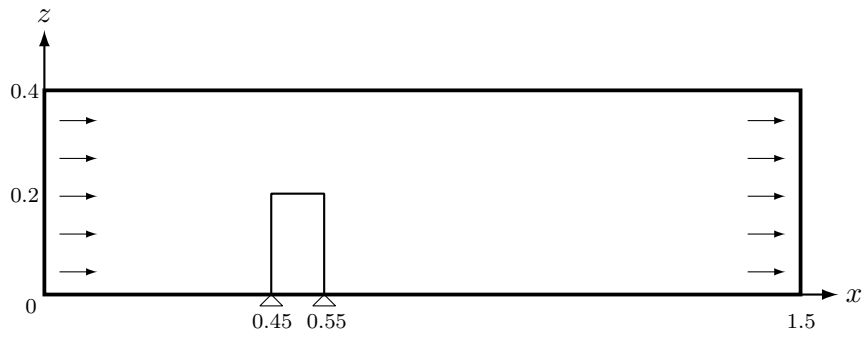Figure 21: Solver Times

Figure 22: Overall Times

Figure 23: Sketch of the "Bending Tower" example. An rectangular obstacle is put into a channel and flow is driven from the left side towords the right side where it leaves the fluid domain. The tower-like obstacle is represented by the structure mesh, the channel by the fluid mesh.

- **Mesh dimensions**
  Tower height $h_t = 0.2$
  Tower length $l_t = 0.1$
  Tower thickness $t = 0.01$
  Channel height $h_c = 0.4$
  Channel length $l_c = 1.5$

- **Material properties**
  Young's Modulus $E = 1.0 \times 10^4$
  Poisson's ratio $\nu = 0.4$

- **Boundary conditions**
  The bottom side of the tower is simply supported.

Displacement	Results from program	Difference (%)

Table 8: Displacements and deviations for Test H

Results:

7 Summary and Conclusions

This chapter summarizes the important results achieved by the thesis and presents a discussion of these results. In the remainder of this chapter an outlook to future development is given.

7.1 Summary

The aim of the thesis was to develop a FEM-code being able to be coupled in a fluid-structure interaction. The program development should be supported by a FEM framework. Comparison aspects had to be created and an evaluation of several FEM libraries was performed. The implemented FEM-code was to be validated with example problems. The coupling part should be managed by the preCICE tool. The overall focus was on creating a well documented and easily readable and maintainable code, ready for further development and extensions. For the FEM framework evaluation different comparison aspects were introduced. Besides organizational requirements like an open-source code, C++ as development programming language and a wide and accurate documentation of the functions and classes, numerical and programmatic aspects were considered. The last aspects included the possibility to parallelize the code via MPI, a large collection of finite element types and built-in iterative linear solvers. During the evaluation process two libraries were practically tested, the first one being MFEM. Because of difficulties in use, another library - libMesh - was tested and finally chosen for the program development. In this thesis flat shell elements were implemented. Such an element is composed of a plane and a plate element part that are superimposed in order to construct the final shell element. Two different types of finite elements were considered: A three-node triangular element, denoted as Tri-3 and a four-node quadrilateral element, denoted as Quad-4. Six models had to be implemented in the scope of this thesis: A plane, plate and shell element for each of the two discretizations. Therefore, existing models like the Discrete Kirchhoff Quadrilateral were selected for implementation. The libMesh framework assisted with developing the FEM-code by providing many components and classes that only needed to be configured and put into the program like blocks. The import of mesh files and the construction of the internal representation of the mesh as nodes, edges and elements is done by libMesh. Also, the creation of a solver and the boundary conditions is simplified by using predefined classes. The solving of the system is done by libMesh or by the external library PETSc which is required as soon as the program is executed in parallel. The major task for the developer was to create the assembly function that construct the overall system matrix and right-hand side. Here, every element was transformed from global space to a local coordinate system in order to assemble its local stiffness matrix. The local stiffness matrices were then re-transformed into global space in order to be added to the global system matrix. The adding step as well as the constraining of elements at the boundary is on the other hand performed by libMesh. Due to the fact that MPI is integrated in libMesh throughout the library, the parallelization of the program only required minor modifications to the code. The coupling tool preCICE was used to create a second version of the program that is ca-

pable of being coupled with other solvers in a multi-physics simulation. preCICE serves as connector between the single solvers, managing data mapping and communication, for instance. One goal of preCICE is to require only minimal changes to the solver's code in order to integrate its API and make it coupling-ready. This was confirmed, as only a few lines of code had to be changed besides additional data structures necessary for inter-solver communication. Every implemented element model were validated with respect to accuracy. In addition to that the convergence was tested, i.e. the behavior of accuracy with an increasing level of mesh subdivision. The parallelization was validated with the same example problem solved by an increasing number of processes and measuring the execution time of different code parts like the matrix assembly or the solving step. The coupling was tested with an fluid-structure interaction problem with the developed program and an OpenFOAM fluid solver as participants. In the test a flow was driven through a pipe with an elastic structure fixed at the bottom side of the tube as deformable obstacle for the flow.

7.2 Conclusion

The evaluation of FEM frameworks yield two suitable libraries for the program's development. The first framework that was used was MFEM. A problem in the practical use made it necessary to switch to another library. Two dimensional geometrical elements are processed within the scope of the program. These elements can be positioned arbitrarily in the three dimensional space. When defining 2D elements in a mesh file that is specified to be in 3D space lead to undefined behavior or even the crashing of the program. If 2D elements were specified with 3D coordinate in a mesh file that were specified to be in 2D space the third coordinate component was ignored. Every tested solution to this problem brought new problems and therefore the second suitable library was taken into account. The libMesh framework offered basically the same features as MFEM although it has other specializations like the focus on adaptive mesh refinements. With the help of this framework the program could be developed without any major issues. The possibilities of libMesh qualify for further use in future development of the structure solver.

In order to offer an appropriate structure solver for coupling purposes, all element components used for the implementation of flat shell elements were validated separately. Hence, several example problems were selected that provided either an analytical solution or solutions created by commercial and approved software. The tests of the Tri-3 and Quad-4 plane elements showed very good accuracy compared to the commercial software SAP2000 with a difference of less than 0.03%. The first example problem also illustrated that the arrangement of elements in the mesh has an important part in accuracy. The test of the Tri-3 plate element indicated a first impression of the involvement of mesh subdivisions: While the accuracy was only around 9% for a 4×4 mesh refinement, the accuracy increased to only around 1% difference to the exact value. Another involved factor got visible in the Quad-4 plate element test: The type of loading has different influence to the accuracy of the example problem. A uniform load where the forces are distributed throughout the whole plate leads to good accuracy even with coarse mesh

refinement. For a concentrated loading applied on only one node at the center of the plate, the mesh refinement must be increased to gain the same level of accuracy. This behavior can only be observed with plate elements and is due to the chosen finite element models. A eight-node quadrilateral element would approximate edge twists better and therefore lead to more accurate results. The combinations of plane and plate elements produce a shell elements. These were tested with an example problem where an I-beam was fixed at the one end and nodal forces were applied at the other end facing inwards to the middle of the beam. This results in deformations in all three spatial axes. The accuracy of the Tri-3 element was better compared to the Quad-4 element although the difference between the x-, y- and z-displacement is larger when compared to the quadrilateral element. In a separate series of tests the element's convergence with respect to accuracy was validated. The level of mesh subdivisions was increased from test to test. While all tests showed that the higher the mesh refinement is the better the analytical solution is approximated by the solver, the test with uniform loading and clamped boundary conditions sticks out: The accuracy stagnates around 1.7% difference to the exact value independently from the number of elements forming the plate. This behavior cannot be explained and might only be specific to this scenario. The other tests confirmed previous observations: With a uniform loading the accuracy is high even for coarse meshes while concentrated loadings need finer meshes. With clamped boundary conditions the solutions show large difference to the analytical result if the mesh is not very fine subdivided.

The parallelization of the program was validated in an additional test. For this, three different times were measured while executing the program: The time needed to assembly the system matrix and right-hand side, the time to solve the system and the overall time from the beginning of the program to the last line of code. This validation must be split in halves since it contains code that is outside of the scope of this thesis: The solving is done by PETSc in parallel. The benefits with respect to time saving can only be observed not explained. The other half is the assembly of the system. Here, every element of the mesh must be processed. This task can be efficiently partitioned and lend to the single processes. Therefore the time for assembling the matrix is nearly halved from the sequential to the execution with two processes. This would lead to a speedup of around 100% for this part of the code and without further tests with more processes. Since every element can predominantly be processed independently from the others this speedup value is possible, though. The bottleneck in this case is the adding of the local matrices to the overall system matrix that is done by libMesh. The solver behaves less efficient compared to the assembly function. Here, the benefit of multiple processes is also visible, but not as dominant as before. The tests were performed on an Intel dual-core processor that uses hyper-threading. This technology lead to even shorter times when both, physical and logical cores are active. Though, the two additional logical cores cannot replace a real quad-core processor. This is especially noticeable when using three processes, i.e. the two physical cores and one logical core: The computation times even rise compared to only two processes but still stay below the single-threaded run. When using three processes it is imaginable that either the operating system or the processor controller have difficulties to handle the extra logical core besides the two

physical cores that are used. This could lead to extra managements like communication, memory transfer or context switches that slows down the execution time. All in all one can state that the solver is capable of being executed in parallel and offers good performance with multiple processes. The solver takes the biggest part of the runtime at least for big meshes. For very small meshes the time to import the mesh, initialize the system and write the outputs - which would otherwise cost minimal time - dominates the execution time.

The last validation test was a fluid-structure interaction where the developed program was coupled with a fluid solver written with the OpenFOAM framework and connected via the preCICE tool.

As a resumé the developed structure solver can be seen as qualified for multi-physics simulation under the assumption of a fine level of mesh subdivision for good accuracy. The good performance in parallel scaling makes it possible to provide such a fine subdivision level without risking to spend too much time for processing of the increased number of elements.

7.3 Future Development

The developed FEM-code successfully implemented flat shell elements and is able to work in coupled multi-physics simulations. The validation showed good accuracy for fine mesh subdivisions. The program was designed to be able to easily introduce new models for plane and plate elements, for example quadratic or cubic quadrilateral elements with 8 or 9 nodes. This would enhance the accuracy further. The requirements of the developed program included two dimensional elements. One could expand this limit also to beam and truss elements of one dimension or even to three dimensional solid elements. The libMesh library and the coupling tool preCICE would support such features.

All scenarios considered in this work so far had constant conditions. The finite element idealization could be extended to situations that are time dependent and simultaneously add dynamic behavior to the elastic structures. When displacements of an elastic body vary with time two additional forces must be introduced, namely the inertia or acceleration and resistance opposing the motion. Therefore, a new type of solver must be introduced as well.

References

[BAA⁺15] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015.

[BBH80] Jean-Louis Batoz, Klaus-JÜRgen Bathe, and Lee-Wing Ho. A study of three-node triangular plate bending elements. *International Journal for Numerical Methods in Engineering*, 15(12):1771–1812, 1980.

[BF85] PG Bergan and CA Felippa. A triangular membrane element with rotational degrees of freedom. *Computer Methods in Applied Mechanics and Engineering*, 50(1):25–69, 1985.

[BLSS] Martin Bogdan, Kai Ludwig, Elena Sapozhnikova, and Bernd Speiser. Echem++ - a problem solving environment for electrochemistry. `http://www.echem.uni-tuebingen.de/echem/software/EChem++/echem++.shtml`.

[Bra07] Dietrich Braess. *Finite elements: Theory, fast solvers, and applications in solid mechanics.* Cambridge University Press, 2007.

[BT82] Jean-Louis Batoz and Mabrouk Ben Tahar. Evaluation of a new quadrilateral thin plate bending element. *International Journal for Numerical Methods in Engineering*, 18(11):1655–1677, 1982.

[bun] preCICE - a fully parallel library for multi-physics surface coupling.

[CMPW02] Robert D. Cook, David S. Malkus, Michael E. Plesha, and Robert J. Witt. *Concepts and Applications of Finite Element Analysis.* J. Wiley, 2002.

[Con] Feel++ Consortium. Feel++ - finite element embedded library in c++. `http://www.feelpp.org`.

[eig] Eigen – a c++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. `http://eigen.tuxfamily.org`.

[Fou] OpenFOAM Foundation. OpenFOAM® | open source CFD | the openFOAM foundation. `http://www.openfoam.org`.

[Gat15] Bernhard Gatzhammer. *Efficient and flexible partitioned simulation of fluid-structure interactions.* Verlag Dr. Hut, 2015.

[gms] Gmsh manual. `http://geuz.org/gmsh/doc/texinfo/gmsh.html`.

[Jar] Alexander H. Jarosch. icetools: a numerical ice flow model download. `http://sourceforge.net/projects/icetools/`.

[Jin94]   Lanheng Jin. *Analysis and evaluation of a shell finite element with drilling degree of freedom.* PhD thesis, 1994.

[Kan04]   Kaushalkumar Kansara. *Development of Membrane, Plate and Flat Shell Elements in Java.* PhD thesis, Virginia Tech, 2004.

[Kir07]   Benjamin Shelton Kirk. *Adaptive Finite Element Simulation of Flow and Transport Applications on Parallel Computers.* PhD thesis, Austin, TX, USA, 2007.

[KK98]   George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[Kle13]   Bernd Klein. *FEM: Grundlagen und Anwendungen der Finite-Elemente-Methode.* Springer-Verlag, 2013.

[KPS13]   B. S. Kirk, J. W. Peterson, and R. H. Stogner. The libMesh finite element library: A case for object-oriented high-performance computing. In *PRACE Summer School 2013 - Frameworks for Scientific Computing on Supercomputers; Ostrava, Czech Republic*, June 17–21, 2013. `http://ntrs.nasa.gov/search.jsp?R=20130013759`.

[KPSC06]   B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3–4):237–254, 2006. `http://dx.doi.org/10.1007/s00366-006-0049-3`.

[las]   LASpack download page. `http://www.netlib.org/linalg/`.

[LCY⁺06]   Rainald Löhner, Juan R Cebral, Chi Yang, Joseph D Baum, Eric L Mestreau, and Orlando Soto. Extending the range and applicability of the loose coupling approach for fsi simulations. In *Fluid-structure interaction*, pages 82–100. Springer, 2006.

[lib]   libmesh publications listing. `http://libmesh.github.io/publications.html`.

[LKBD09]   Gerhard Link, M Kaltenbacher, Michael Breuer, and M Döllinger. A 2d finite-element scheme for fluid–solid–acoustic interactions and its application to human phonation. *Computer Methods in Applied Mechanics and Engineering*, 198(41):3321–3334, 2009.

[mfea]   MFEM - finite element discretization library. `http://mfem.org/publications/`.

[mfeb]   MFEM: Modular finite element methods. `http://www.mfem.org`.

[MH85]   Richard H Macneal and Robert L Harder. A proposed standard set of problems to test finite element accuracy. *Finite elements in Analysis and Design*, 1(1):3–20, 1985.

[Pat]   Bořek Patzák. publications [oofem wiki]. `http://www.oofem.org/wiki/doku.php?id=publications`.

[Pat09]   Bořek Patzák. Oofem project home page, 2009. `http://www.oofem.org`.

[PB01]   B. Patzák and Z. Bittnar. Design of object oriented finite element code. *Advances in Engineering Software*, 32(10):759–767, 2001.

[PCD⁺12]   Christophe Prud'Homme, Vincent Chabannes, Vincent Doyeux, Mourad Ismail, Abdoulaye Samaké, and Gonçalo Pena. Feel++: A computational framework for galerkin methods and advanced numerical methods. In *ESAIM: Proceedings*, volume 38, pages 429–455. EDP Sciences, 2012.

[pet]   PETSc web page. `http://www.mcs.anl.gov/petsc`.

[PR]   Julien Pommier and Yves Renard. Getfem++ - an open source library based on collaborative development. `http://download.gna.org/getfem/html/homepage/index.html`.

[sap00]   *SAP2000 - Integrated Finite Element Analysis and Design of Structures*, 2000.

[SMU⁺10]   Frank Schäfer, Stefan Müller, Thomas Uffinger, Stefan Becker, Jens Grabinger, and Manfred Kaltenbacher. Fluid-structure-acoustic interaction of the flow past a thin flexible structure. *AIAA journal*, 48(4):738–748, 2010.

[Spe88]   Bernhard Specht. Modified shape functions for the three-node plate bending element passing the patch test. *International Journal for Numerical Methods in Engineering*, 26(3):705–715, 1988.

[SPR]   Luis Saavedra, Julien Pommier, and Yves Renard. Mpi parallelization of getfem++ - getfem++. `http://download.gna.org/getfem/html/homepage/userdoc/parallel.html`.

[Ste15]   Peter Steinke. *Finite-Elemente-Methode: Rechnergestützte Einführung.* Springer-Verlag, 2015.

[Thi]   Axel Thielscher. free software package for the simulation of non-invasive brain stimulation. `http://simnibs.de/`.

[Toc63]   James Lionel Tocher. *Analysis of plate bending using triangular elements.* PhD thesis, University of California, Berkeley, 1963.

[Wer95]   Horst Werkle. *Finite Elemente in der Baustatik.* Springer, 1995.

[Wil96]   Edward L Wilson. *Three-dimensional static and dynamic analysis of structures.* Computers and Structures Inc., Berkeley, California, 1996.

[ZT00]   Olgierd Cecil Zienkiewicz and Robert Leroy Taylor. *The finite element method: Solid mechanics*, volume 2. Butterworth-heinemann, 2000.

All URLs were lastly checked at December 8, 2015.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift