Dirichlet Boundary Value Problems on Deformed Domains

Bachelor Thesis

Christian Baumann

Monday 10th July, 2017

Advisor: Prof. Dr. R. Hiptmair
Seminar for Applied Mathematics, ETH Zürich

Abstract

In this thesis we solve a Dirichlet boundary value problem (BVP) on a deformed domain, as the title suggests. Since this deformed domain is only given by its boundary, we first need to solve another Dirichlet BVP which yields a mapping of the unit disk to the given deformed domain.

Then we can apply a Galerkin finite element discretization and solve the problem numerically. Since we need to calculate the mapping from the unit disk to the deformed domain, we only need a mesh of the unit disk to solve the problem on the deformed mesh by using transformation techniques. For the discretization a hybrid mesh is used that contains elements of different orders, large elements of high order in the center and small elements of low order closer to the boundary.

Finally a convergence study is conducted and convergence rates are determined empirically. This all is implemented in C++ using the linear algebra library Eigen [3].

Contents

Chapter 1

Introduction

1.1 Motivation

Assume we have some domain Ω specified by a possibly large parameter vector $\mathbf{y}$. Our problem consists of not knowing these parameters exactly, i.e. $\mathbf{y}$ follows some distribution with known or estimated uncertainty. Therefore the geometry of Ω is *not* known exactly. Our goal is now to solve a PDE over this uncertain domain and retrieve some information about the uncertainty of the solution of this PDE. This problem is known as Shape Uncertainty Quantification, an application of which can be found in [7]. One possibility to solve such a problem uses a Monte-Carlo scheme where the PDE is solved on many domains determined by samples from some distribution and then from these solutions the statistics is computed. If we implement this directly, we would need to generate meshes for every sampled domain and this might be computationally expensive. In order to avoid this re-meshing, this thesis describes a way of solving the PDE with the finite element method using only one mesh, a mesh of the unit disk.

1.2 Notation

Here are some explanations about the notation used and some conventions used to simplify the writing process. For any variable holds that if it is in bold, it is vector. E.g. $\mathbf{z}$ is a vector, whereas z is just a scalar. The problem discussed in this papper is in 2D and $\mathbf{x} = (x, y), \tilde{\mathbf{x}} = (\tilde{x}, \tilde{y})$ and $\hat{\mathbf{x}} = (\hat{x}, \hat{y})$ are possible coordinates in some 2-dimensional domains. In general, the letters $u, v, g, f, \mathbf{w}$ denote functions, if they are in bold they are vector valued and scalar valued otherwise. Most of them take some coordinates as an argument which will often be omitted to make the equations shorter, so $g = g(\mathbf{x}) = g(x, y)$, what exactly is meant as argument should be clear from the context. Further, ∇u denotes the gradient of u, whereas $\nabla \cdot \mathbf{w}$ denotes the divergence of $\mathbf{w}$. If Φ is a mapping from some space to another, then $D\Phi$ denotes its jacobian. A short overview of the things discussed and some more information can be found below.

Symbol	Meaning
$x, y, z, c, s, \phi, \alpha, \mu$	scalar variable or constant
i, j, k, N, M	integer
A, M	matrix
$\mathbf{x}, \mathbf{y}, \mathbf{z}, \boldsymbol{\alpha}, \boldsymbol{\mu}$	vector
u, v, f, g, r	scalar function taking a vector or a real number as argument
Ω, K, B_1	2D domain
$\partial\Omega, \partial B_1$	boundary of 2D domain
$\mathbf{w}$	vector-valued function
Φ, φ	mapping from a 2D domain to another
$\nabla u, \nabla v$	gradient of scalar function
$D\Phi$	jacobian of vector-valued function
$\nabla \cdot \mathbf{w}$	divergence of vector-valued function

Subscripts usually denote elements of a vector, i.e. if $\mathbf{c}$ is a vector, then c_i is its ith element. If some symbol carries a superscript which is an integer, it usually is a discrete approximation of some quantity, e.g. u^N denotes an N-dimensional approximation of u.

1.3 General Remarks

The code contributing to this thesis can be found at Gitlab at the following url: https://gitlab.com/chbauman/Bachelor_Thesis_Christian_Baumann. A description of how to execute the code is provided. All the plots in this thesis were created with python's Matplotlib [5], the corresponding plotting files can also be found at Gitlab. There are even some Mathematica [6] notebooks, but they are not needed to run the code and reproduce the results presented in this thesis.

Chapter 2

Problem Description

2.1 Strong formulation

Given a 2D star-shaped bounded domain $\Omega \subset \mathbb{R}^2$ with boundary $\partial\Omega$, the following boundary value problem needs to be solved. Find a function $u \in H^1(\Omega)$, such that

$$\begin{aligned} -\Delta u &= 0 \text{ on } \Omega \\ u &= g \text{ on } \partial\Omega \end{aligned} \tag{2.1}$$

for a given function $g \in H^1(\Omega)$. This seems simple, but the problem is that the domain Ω is only given by its boundary

$$\partial\Omega := \{r(\phi)(\sin(\phi), \cos(\phi))^T \subset \mathbb{R}^2, 0 \leq \phi \leq 2\pi\} \tag{2.2}$$

where $r : [0, 2\pi] \mapsto \mathbb{R}$ is a continuous function which may be given through a real Fourier series

$$r(\phi) := 1 + \sum_{j=1}^{N}(c_j y_j \cos(j\phi) + s_j z_j \sin(j\phi)) \tag{2.3}$$

with parameters $-1 \leq y_j, z_j \leq 1$, which are collected in the parameter vector $\mathbf{y}$. We assume

$$\sum_{j=1}^{\infty}(|c_j| + |s_j|) \leq \frac{1}{2} \tag{2.4}$$

In order to solve the original problem, 2.1, we apply a *mapping approach*, transforming it to the unit disc $B_1 \subset \mathbb{R}^2$ by means of the diffeomorphism $\mathbf{\Phi} : B_1 \mapsto \Omega(\mathbf{y})$, defined as

$$\mathbf{\Phi}(\hat{\mathbf{x}}) := \hat{\mathbf{x}} + \mathbf{w}(\hat{\mathbf{x}}) \tag{2.5}$$

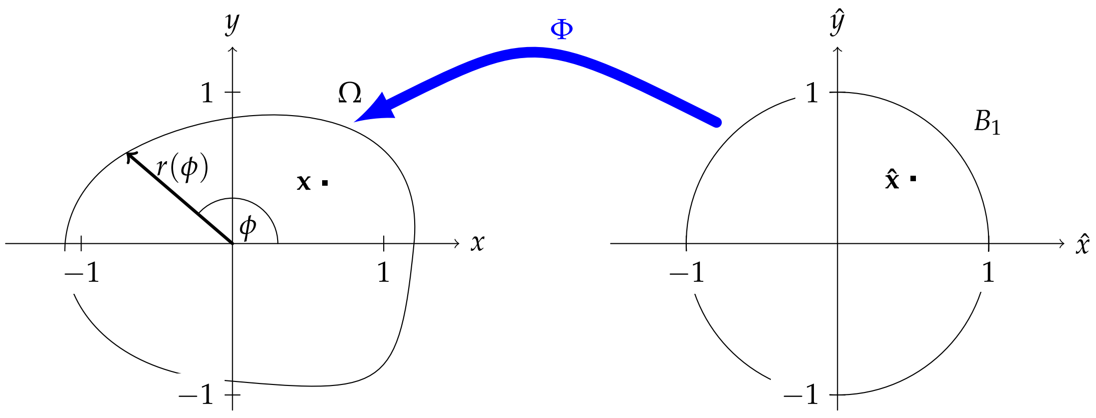


Figure 2.1: Sketch of the situation.

and $\mathbf{w}$ can be found by solving the following problem

$$\begin{aligned} -\Delta \mathbf{w} &= 0 \text{ on } B_1 \\ \mathbf{w} &= \begin{pmatrix} (r(\phi)-1)\cos\phi \\ (r(\phi)-1)\sin\phi \end{pmatrix} \text{ on } \partial B_1 \end{aligned} \tag{2.6}$$

where ϕ is the polar angle coordinate on ∂B_1. Figure 2.1 illustrates the meaning of $r(\phi)$ for an example of such a domain Ω. This special Ω was constructed *only for visualization* and will also be used later in this paper, the corresponding coefficients c_j, s_j, z_j and y_j are unknown and may not fulfill inequality 2.4.

2.2 Variational formulation

To be able to apply a Galerkin discretization, we first need the weak variational formulation of the problem 2.1. To get to that, we multiply the PDE with a test function $v \in H_0^1(\Omega)$ that vanishes on the boundary of Ω and use integration by parts in 2D to get the following modified problem. Find $u \in H^1(\Omega)$ s.t.

$$\begin{aligned} \int_\Omega \nabla u \cdot \nabla v dx = 0 \ \forall v \in H_0^1(\Omega) \\ u = g \text{ on } \partial\Omega \end{aligned} \tag{2.7}$$

Using transformation techniques from [4], we can transform this problem back to the unit disk. If we do this, we get the problem. Find $\hat{u} \in H^1(B_1)$ s.t.

$$\begin{aligned} \int_{B_1} D\Phi^{-T}\nabla\hat{u} \cdot D\Phi^{-T}\nabla\hat{v} |\det D\Phi| d\hat{x} = 0 \ \forall \hat{v} \in H_0^1(B_1) \\ \hat{u}(\hat{\mathbf{x}}) = g(\Phi(\hat{\mathbf{x}})) \text{ on } \partial B_1 \end{aligned} \tag{2.8}$$

where Φ is the transformation defined in 2.5 and it holds that $\hat{u}(\hat{\mathbf{x}}) = u(\Phi(\hat{\mathbf{x}}))$, the same relation is true for $\hat{v}$ and v. To find Φ, we need to solve another Dirichlet BVP. We apply exactly the same techniques to find the variational formulation of the auxiliary problem 2.6 and get the following. Find $\mathbf{w} = (w_1, w_2)^T \in (H^1(B_1))^2$ s.t.

$$\begin{aligned} \int\limits_{B_1} \nabla w_1 \cdot \nabla \hat{v} d\hat{x} = 0 \ \forall \hat{v} \in H_0^1(B_1) \\ \int\limits_{B_1} \nabla w_2 \cdot \nabla \hat{v} d\hat{x} = 0 \ \forall \hat{v} \in H_0^1(B_1) \\ \mathbf{w} = \begin{pmatrix} (r(\phi) - 1)\cos\phi \\ (r(\phi) - 1)\sin\phi \end{pmatrix} \text{ on } \partial B_1 \end{aligned} \tag{2.9}$$

This means that for the transformation Φ we need to solve two decoupled Dirichlet BVPs, one for each component of $\mathbf{w}$. Since these problems are posed directly on B_1, no transformation to another domain is needed.

Chapter 3

Discretization

To solve the problem defined in the previous section numerically, we use a Galerkin discretization with Lagrangian finite elements.

3.1 Meshes

3.1.1 Notation and Conventions

Let N denote the number of interpolation nodes and M the number of elements in the mesh. N_∂ denotes the number of interpolation nodes on the boundary and N_0 denotes the number of interpolation nodes that do not lie on the boundary. So it follows that $N = N_\partial + N_0$. The ith interpolation node is denoted as $d_i, i = 1, ..., N$. As a convention and to simplify the programming part, we constrain the interpolation nodes that lie on the boundary of our domain to have the *lowest* indices, i.e. d_i lies on the boundary if $i \leq N_\partial$.

3.1.2 Mesh Used for Convergence Study

For the whole problem a hybrid mesh containing small linear elements at the boundary and large high order elements in the center was used. Figure 3.1 shows the four coarsest meshes of this kind that will be used for convergence study. The black dots denote the interpolation nodes. Note that on the boundary of the domain only linear elements are used, this means that only a piecewise linear boundary approximation is used. Note also that the corresponding FE spaces are not strictly nested because of this boundary approximation.

Another set of meshes is shown in figure 3.2. These meshes contain quadratic triangular elements only. The boundary approximation is also piecewise linear as for the previous meshes. Since the elements at the boundary are not linear but quadratic, a better boundary approximation would have been possible, but the main task of this set of meshes is to validate the code, i.e. we

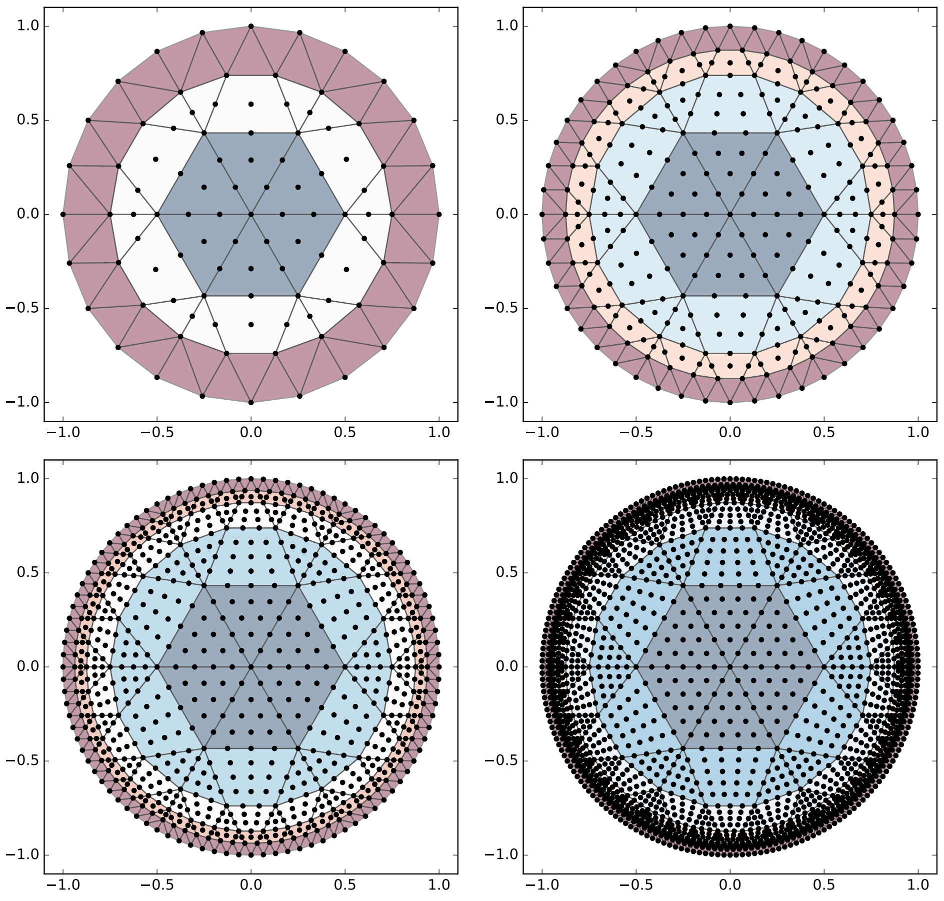


Figure 3.1: Coarsest Hybrid Meshes

want to see if the empirical convergence rates for this set of meshes coincide with the ones from theory.

3.2 Auxiliary problem

Let's first discretize the auxiliary problem 2.9. We use a mesh containing Lagrangian finite elements of varying orders. Since we need to solve two problems that are the same except for the Dirichlet boundary conditions, we will just discuss how to find an approximation of $w_1(\hat{\mathbf{x}})$, then w_2 can be found in the same way using the corresponding boundary conditions. As basis functions we use piecewise polynomial functions, denoted by b_i, that are different from zero at only one interpolation node, more precisely they satisfy the condition:

$$b_i(\hat{\mathbf{d}}_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{3.1}$$

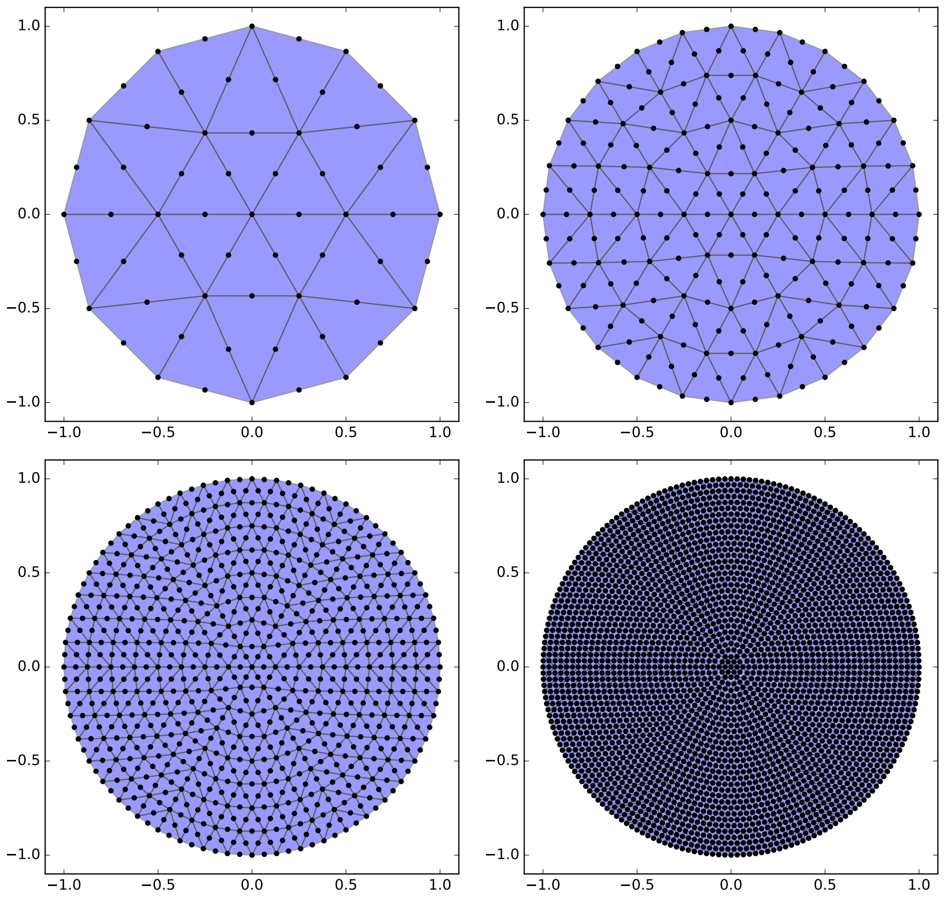


Figure 3.2: Coarsest Quadratic Triangular Meshes

where $\hat{\mathbf{d}}_j$ denotes the jth interpolation node of the discretized domain B_1. If we restrict such a basis function to an element, say K, then we can describe it as a polynomial, i.e. $b_i|_K \in \mathcal{P}_n$ for some $n \in \mathbb{N}$ depending on the order of the basis functions used. Now we can approximate the function space H^1 as $H^{1,N} := span\{\hat{b}_i | i = 1, ..., N\}$. The other space H_0^1 is approximated similarly as $H_0^{1,N} := span\{\hat{b}_i | \hat{\mathbf{d}}_i$ does not lie on the boundary$\}$. Now we need to find an approximation of w_1, denoted as w_1^N in $H^{1,N}$ s.t. the equations of 2.9 hold for all $v^N \in H_0^{1,N}$. For this purpose, we write w_1^N as a linear combination of our basis functions $\hat{b}_i(\hat{\mathbf{x}}), i = 1, ..., N$, i.e.

$$w_1^N(\hat{\mathbf{x}}) := \sum_{i=1}^{N} \alpha_{1,i} \hat{b}_i(\hat{\mathbf{x}}) \tag{3.2}$$

with unknown coefficients $\alpha_{1,i}, i = 1, ..., N$. Since the problem is linear in the test functions v and its gradients ∇v, we *only* need to test the equations of 2.9 for the basis functions. If we plug 3.2 into 2.9 we get the discrete problem:

$$\sum_{i=1}^{N} \alpha_{1,i} \int_{B_1} \nabla \hat{b}_i \cdot \nabla \hat{b}_j d\hat{x} = 0 \ \forall \hat{b}_j \in H_0^{1,N}(B_1)$$
$$\alpha_{1,i} = (r(\varphi) - 1)\cos\varphi \text{ if interpolation node } i \text{ lies on } \partial B_1 \tag{3.3}$$

The boundary condition is equivalent to $\alpha_{1,i} = (r(\arg \hat{\mathbf{d}}_i) - 1)\cos(\arg \hat{\mathbf{d}}_i)$ for $i \leq N_\partial$ according to the convention in 3.1.1 and φ is just the argument of interpolation node d_i. Similarly the first equation of 3.3 has to be true for all $\hat{b}_j$ with $N_\partial < j \leq N$ since the first N_∂ interpolation nodes are those that lie on the boundary. We can rewrite 3.3 as

$$\sum_{i=N_\partial+1}^{N} \alpha_{1,i} \int_{B_1} \nabla \hat{b}_i \cdot \nabla \hat{b}_j d\hat{x} = -\sum_{i=1}^{N_\partial} \alpha_{1,i} \int_{B_1} \nabla \hat{b}_i \cdot \nabla \hat{b}_j d\hat{x} \text{ for } j = N_\partial + 1, ..., N \tag{3.4}$$

and we get a linear system of equations with $N - N_\partial = N_0$ unknowns $\alpha_{1,N_\partial+1}, ..., \alpha_{1,N}$ and N_0 equations. Next we define $A^{w_1} \in \mathbb{R}^{N,N}$,

$$(A^{w_1})_{i,j} = \int_{B_1} \nabla \hat{b}_i \cdot \nabla \hat{b}_j d\hat{x} \ i,j = 1, ..., N \tag{3.5}$$

and partition it similarly as in [4, Chapter 3, Section 3.6.6] in the following way, $A^{w_1} = \begin{pmatrix} A^{w_1}_{\partial,\partial} & A^{w_1}_{\partial,0} \\ A^{w_1}_{0,\partial} & A^{w_1}_{0,0} \end{pmatrix}$ with $A^{w_1}_{\partial,\partial} \in \mathbb{R}^{N_\partial,N_\partial}$ and $A^{w_1}_{0,0} \in \mathbb{R}^{N_0,N_0}$. We also put the coefficients $\alpha_{1,i}$ into a vector $\boldsymbol{\alpha}_1$, $(\boldsymbol{\alpha}_1)_i = \alpha_{1,i}$ for $i = 1, ..., N$, and partition it similarly, i.e. $\boldsymbol{\alpha}_1 = \begin{pmatrix} \boldsymbol{\alpha}_{1,\partial} \\ \boldsymbol{\alpha}_{1,0} \end{pmatrix}$. In this case $\boldsymbol{\alpha}_{1,\partial} \in \mathbb{R}^{N_\partial}$ is known from the boundary conditions, and $\boldsymbol{\alpha}_{1,0} \in \mathbb{R}^{N_0}$ are the unknown coefficients we want to find. With these definitions, we can rewrite 3.4 in linear algebra notation:

$$A^{w_1}_{0,0}\boldsymbol{\alpha}_{1,0} = -A^{w_1}_{0,\partial}\boldsymbol{\alpha}_{1,\partial} \tag{3.6}$$

This can be solved easily, the only problem remaining is the evaluation of the entries of A^{w_1}, as described in equation 3.5. This will be discussed in chapter 4, which is about implementation. Similarly as described above, we can find w_2^N and by equation 2.5 we can find an approximation of Φ:

$$\Phi^N(\hat{\mathbf{x}}) := \hat{\mathbf{x}} + \mathbf{w}^N(\hat{\mathbf{x}}) \tag{3.7}$$

where $\mathbf{w}^N(\hat{\mathbf{x}}) = (w_1^N(\hat{\mathbf{x}}), w_2^N(\hat{\mathbf{x}}))^T$.

3.3 Real Problem

The problem addressed in the previous section was only needed because we didn't have a mesh of the domain Ω, so we need to apply the mapping approach described in 2.1. If we discretize the problem 2.8, which was mapped to the unit disk, in the same way we did in the previous section, we get

$$\hat{u}^N(\hat{\mathbf{x}}) := \sum_{i=1}^{N} \mu_i \hat{b}_i(\hat{\mathbf{x}}) \tag{3.8}$$

$$A^{\hat{u}} = \begin{pmatrix} A^{\hat{u}}_{\partial,\partial} & A^{\hat{u}}_{\partial,0} \\ A^{\hat{u}}_{0,\partial} & A^{\hat{u}}_{0,0} \end{pmatrix}, \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_\partial \\ \boldsymbol{\mu}_0 \end{pmatrix} \tag{3.9}$$

$$A^{\hat{u}}_{0,0}\boldsymbol{\mu}_0 = -A^{\hat{u}}_{0,\partial}\boldsymbol{\mu}_\partial \tag{3.10}$$

where this time the unknown coefficients are $\mu_i, i > N_\partial$. The sizes of the matrices are the same as in the previous case, just the entries are calculated differently:

$$(A^{\hat{u}})_{i,j} = \int_{B_1} (D\Phi^N)^{-T} \nabla \hat{b}_i \cdot (D\Phi^N)^{-T} \nabla \hat{b}_j | \det D\Phi^N | d\hat{x} \; i, j = 1, ..., N \tag{3.11}$$

Note that we used the discrete transformation Φ^N here which we computed in the previous section.

Chapter 4

Implementation

4.1 Mesh Data Structure

To represent a mesh in the code, a class named `mesh` is used, it is defined as follows.

Listing 4.1: Mesh Class

```
typedef unsigned int index_t;
typedef std::vector<element> element_vec_t;
struct mesh {
    element_vec_t elements;
    const index_t numEls;
    const index_t num_dofs;
    const index_t num_boundary_dofs;
    // and some constructor
};
```

As the names suggest, `numEls` $= M$ denotes the number of elements contained in the mesh, `num_dofs` $= N$ denotes the total number of interpolation nodes in the mesh and `num_boundary_dofs` $= N_\partial$ denotes the number of interpolation nodes that lie on the boundary. `elements` finally is a `std::vector` of instances of class `element`. These elements contain information about the connectivity of the mesh and look as follows.

Listing 4.2: Element Class

```
typedef unsigned short el_type_t;
typedef unsigned short el_order_t;
typedef Eigen::Matrix<index_t, -1, 1> index_vec_t;
struct element {
    const el_type_t el_type;
    const el_order_t order;
    const el_order_t num_missing_dofs;
    index_vec_t nodes;
    element(...) {};
    index_t corner(index_t i) const {...};
};
```

In this case, `el_type` determines what type of element it is, i.e. `el_type` is either 3 for triangles or 4 if the element is a quadrilateral. `order` denotes the order of the element, i.e. 1 for linear elements, 2 for quadratic elements etc. `num_missing_dofs` $=0$ for normal elements and `num_missing_dofs` $=1$ for elements that are missing an interpolation node on one edge of the element. The latter are needed because the mesh contains elements of different orders, so the mesh needs to contain some elements that have two different orders. This will be discussed in more detail later. `nodes` finally describes the element by containing the indices of the interpolation nodes that are contained in the element, it is just an index vector and does not contain the coordinates of the interpolation nodes. These interpolation nodes are stored independently of the mesh in an `Eigen::Matrix` as shown below.

Listing 4.3: Dofs

```
typedef double numeric_t;
typedef Eigen::Matrix<numeric_t, 2, -1> nodes_t;
nodes_t nodes;
```

4.2 Mesh Construction

This section describes an algorithm that produces a mesh of the unit disk B_1. The function returning the mesh has the following signature:

Listing 4.4: Signature

```
template<class Vector, class Layers, class Func>
mesh construct_mesh(const Layers & layer_vec,
        const Vector & orders,
        const index_t n_tria_in_center,
        nodes_t & nodes, const Func & r,
        const bool center_higher_order = false)
{...};
```

It takes some parameters which are described later and returns an object of class `mesh`. The `mesh` essentially is a collection of `elements` which are just indices referring to columns in the matrix `nodes`. This 2 by N matrix contains the coordinates of the interpolation nodes of the mesh columnwise.

The parameter layer_vec is a std::vector of Layers that will be descibed later.

4.2.1 Center of the Mesh

The codes takes an argument n_tria_in_center which specifies the number of triangular elements in the center. These will be the center of the mesh, further elements will be added later. This may look as follows in the case of n_tria_in_center = 7:

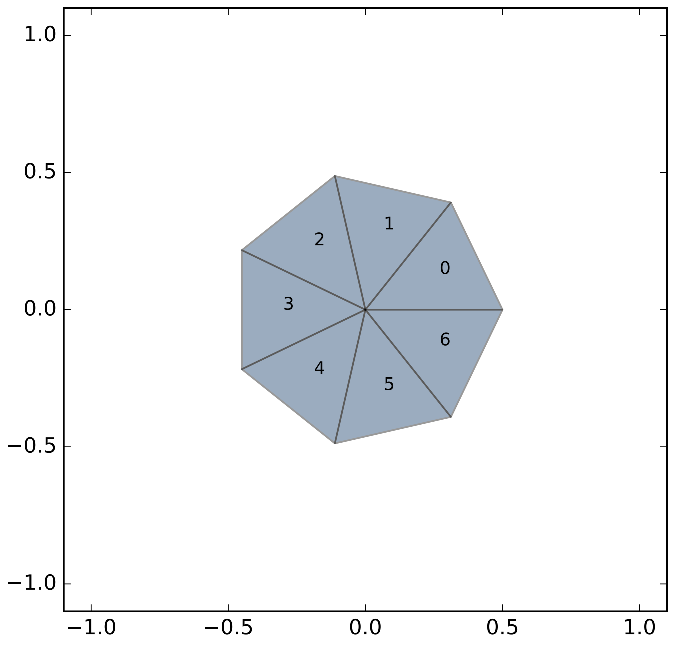


Figure 4.1: The center of the mesh.

4.2.2 Adding layers

Then the code adds layers around the center. These are specified by a vecor of Layers, which are structs containing an index vector ids_vec with indices in $\{1, 2, 3\}$.

Listing 4.5: Signature

```
struct Layer {
    const double offset;
    index_vec_t ids_vec;
    index_t num_inner_els;
    index_t num_outer_els;
    Layer(...){...};
};
```

These indices have the following meaning:

1. Quadrilateral element.
2. Triangular element pointing towards the center.
3. Triangular element pointing away from the center.

The algorithm then puts the elements in the specified order around the previous layer and repeats that until the layer is closed. So we can e.g. define our layer with the index vector: $\{2, 1, 2\}$ which means that the layer will consist of a triangle pointing inwards, a quadrilateral and again a triangle pointing inwards, and then this pattern will be repeated again until no more elements can be added. Fiugre 4.2 shows the mesh after the first part of the layer and after the whole layer is added.

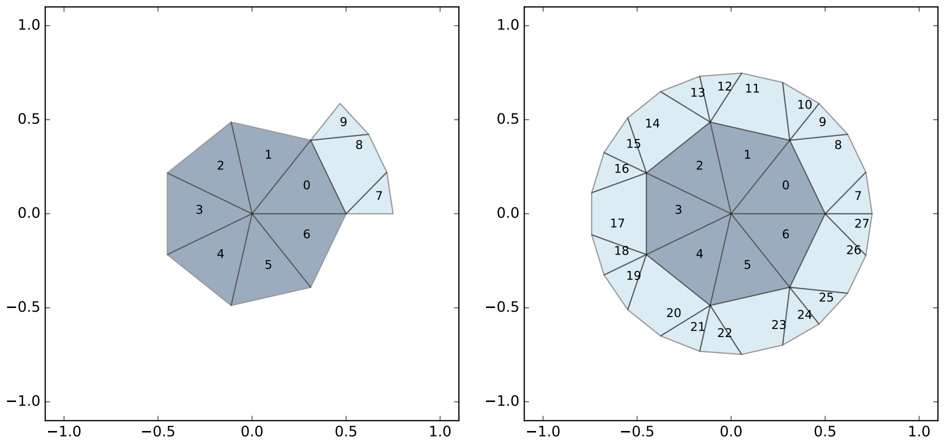


Figure 4.2: Construction of first layer

Concerning the other fields of the Layer class, `num_inner_els` denotes the number of elements that have a common edge with the previous layer, similarly `num_outer_els` denotes the number of elements that have a common edge with the next layer. These integers can be computed from the index vector `ids_vec`, this is done in the constructor of the `struct Layer`. In the above case `num_inner_els` $= 1$ and `num_outer_els` $= 3$.

Note that not any layer can be added to the mesh, they need to be compatible, i.e. if in the previous layer k elements share an edge with the current layer, then `num_inner_els` of the current layer needs to divide k. There can also be an `offset` specified, this means that the new layer will be shifted by `offset` elements in angular direction. In the case of `offset` $= 0.5$ this would look like in figure 4.3. This offset might also be negative.

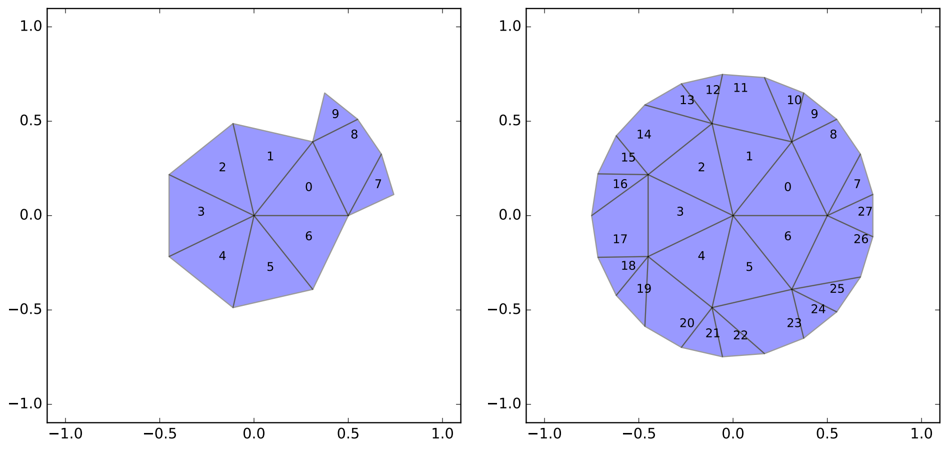


Figure 4.3: First Layer with Offset

One has to be careful when choosing the layers and the function r, the elements should not have too big angles. This might have an impact on the convergence behavior discussed later.

4.2.3 Thickness of layers

The thicknesses of the layer is indirectly specified by a real valued function $r(x) : [0,1] \mapsto [0,1]$. Given there are n layers without including the center, then the center has radius $r(\frac{1}{n+1})$ and then the ith layer, $i \in \{1, ..., n\}$, will occupy the part of the unit circle between $r(\frac{i}{n+1})$ and $r(\frac{i+1}{n+1})$, so it will have thickness $r(\frac{i+1}{n+1}) - r(\frac{i}{n+1})$. If $r(x) = x$ then the layers will all have the same thickness $\frac{i}{n+1}$. To get a mesh that is not overlapping in some parts this function r must be strictly increasing. Figure 4.4 show a mesh with $r(x) = x$, i.e. uniform thicknesses of layers, and figure 4.5 a mesh with $r(x) = \sin(\frac{\pi x}{2})$, i.e. the thickness is smaller for layers closer to the boundary.

4.2.4 Order of Elements in Layer

The order of the elements in a layer can be specified by a vector `orders`, the length of this vector has to be the same as the number of layers in the mesh. Then the i-th component of this vector specifies the polynomial degree of the i-th layer. If `center_higher_order` is `false` then the elements in the center will just have the same order as the elements in the layer next to it, else their order will be one order higher. For simplicity the order of the elements in the ith layer can either be the same as the order of the elements

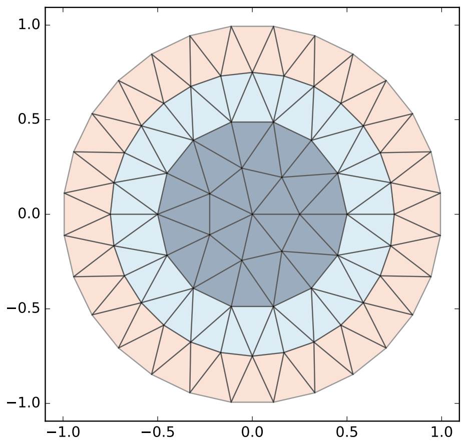


Figure 4.4: Uniform layer thickness.

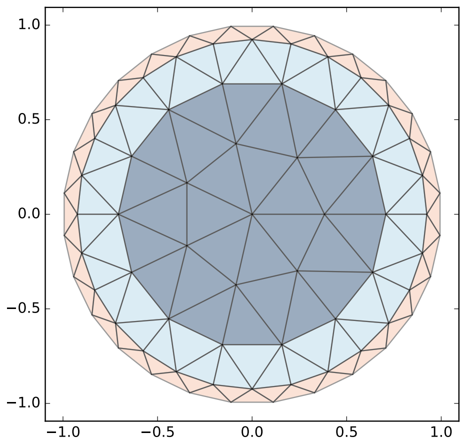


Figure 4.5: Thickness decreasing.

of the previous layer or it can have decreased by one, i.e. the order can only decrease by at most one in each layer when going from the center of the mesh to the boundary. This convention ensures that only one special kind of shape functions have to be implemented, namely ones that have *one* degree of freedom less on only *one* side. Figure 4.6 shows a mesh that was constructed with `orders` $= \{3, 2, 1\}$ and `center_higher_order` = `true`:

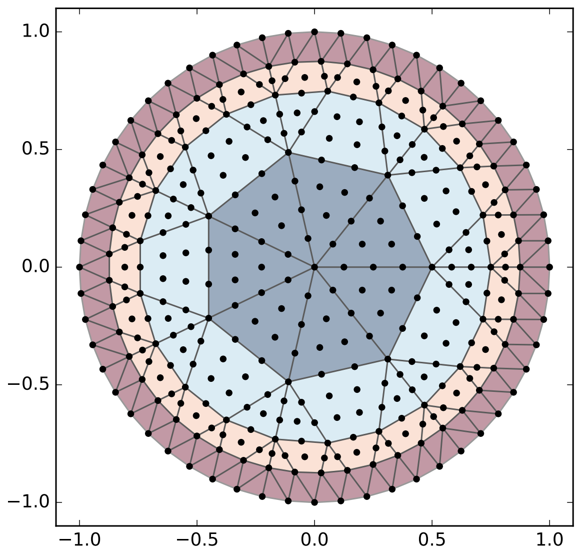


Figure 4.6: Example mesh.

4.2.5 Some example meshes

Figure 4.7 shows a radial mesh containing only a few triangles and many quadrilaterals with constant order 1. Figure 4.8 shows a mesh containing triangles only but with elements of higher order in the center. The maximum order of this particular mesh is 4.

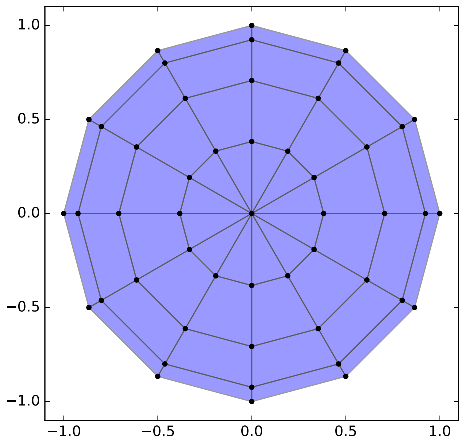


Figure 4.7: Radial mesh, constant order 1

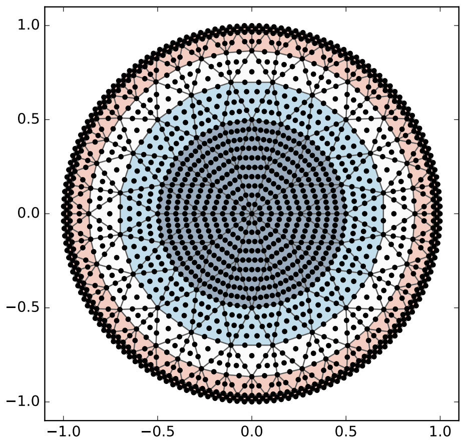


Figure 4.8: Triangular mesh, max. order 4

4.3 Shape Functions

Since we do not have the basis functions we need on any degree element everywhere in $\mathbb{R}^2$, we need to map the integrals needed for the computation of the entries of our matrices back to some reference element, denoted by $\hat{K}$. All the basis functions used in this thesis satisfy the nodal condition already seen in 3.2. But this time we look at the basis functions on the reference element, this means that we now have

$$\bar{b}_i(\bar{\mathbf{d}}_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{4.1}$$

where $\bar{\mathbf{d}}_j$ denotes the j-th local interpolation node and $\bar{b}_i$ the i-th shape function, both on the reference element.

4.3.1 Quadrilateral elements

For a quadrilateral element, the reference element is the unit square $\hat{K}_Q := [0,1]^2$ with subscript Q indicating that it is the reference element of the quadrialteral elements. Let's consider the general case of order p elements.

Since we consider a quadratic element, we have $N_{LD} := (p+1)^2$ interpolation nodes and the same number of shape functions. Figure 4.9 shows how the interpolation nodes are distributed on $\hat{K}_Q$.

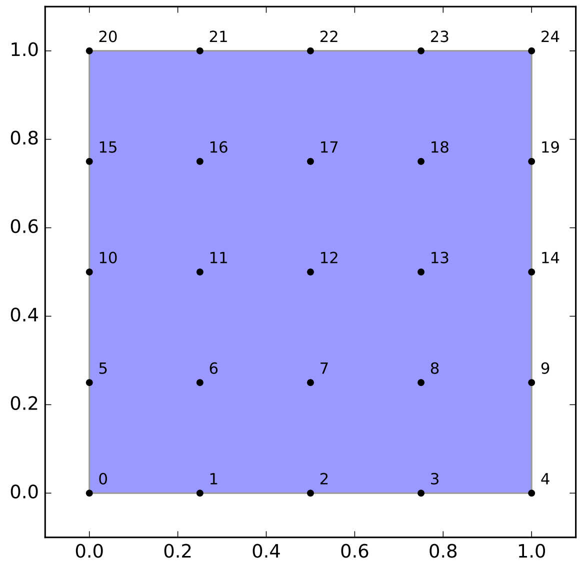


Figure 4.9: Dofs of a quadrilateral element of order 4.

Further it shows how the local ordering of the interpolation nodes was chosen for this thesis: Let $\bar{\mathbf{d}}_j = (x_{d_j}, y_{d_j})^T$, then the index j is smallest for y_{d_j} the smallest, and among those with the same y_{d_j}, the ones with the smallest x_{d_j} have the smallest index. This also allows us to find a nice expression for the interpolation nodes in terms of j.

$$\bar{\mathbf{d}}_j = \frac{1}{p}(j \mod p+1, \left\lfloor \frac{j}{p+1} \right\rfloor)^T \tag{4.2}$$

Now there are two different ways to construct the actual shape functions. Since 4.1 has to hold for every one of the N_{LD} shape functions, we have N_{LD} conditions on every shape function. If we make the ansatz $\bar{\mathbf{b}}_j \in \mathcal{Q}_p(\mathbb{R}^2)$, where $\mathcal{Q}_p(\mathbb{R}^2)$ is the space of tensor product polynomials described in [4, Chapter 3, Section 3.4.2], then we also have the same amount, namely N_{LD}, of coefficients that we can now determine uniquely to get our shape functions by solving a linear system of equations. This has been done using Mathematica [6], for $p = 1, ..., 6$. Listing 4.6 shows the signature of these functions which are implemented in the file `shape_functions_quad.hpp`.

Listing 4.6: Low Order Shape Functions and Gradients for Quadrilateral Element

```cpp
typedef Eigen::Matrix<numeric_t, 2, 1> coords_t;
Eigen::Matrix<numeric_t, -1, 1>
                        b1_quad(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 2>
                        gradb1_quad(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 1>
                        b2_quad(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 2>
                        gradb2_quad(const coords_t & x);
/// ...
Eigen::Matrix<numeric_t, -1, 1>
                        b6_quad(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 2>
                        gradb6_quad(const coords_t & x);
```

`bp_quad(x), p = 1, ..., 6` returns a length $N_{LD} = (p+1)^2$ vector whose entries are all the shape functions of the quadrilateral reference element with corresponding order p evaluated at point `x`. `grad_bp_quad(x)` returns, as the name suggests, the gradients transposed of the corresponding shape functions, all evaluated at point `x` in a N_{LD} by 2 matrix. For $p > 6$ the following method, which works for any p, was used. The shape functions can almost directly be written down as a product of linear functions.

$$\bar{b}_i(x,y) = a_i b_i^x(x) b_i^y(y) = a_i \prod_{\substack{k=0 \\ k \neq i \mod p+1}}^{p} (x - \frac{k}{p}) \prod_{\substack{k=0 \\ k \neq \left\lfloor \frac{i}{p+1} \right\rfloor}}^{p} (y - \frac{k}{p}) \tag{4.3}$$

where a_i is a normalization constant that needs to be chosen such that $\bar{b}_i(\bar{\mathbf{d}}_i) = 1$, i.e.

$$\frac{1}{a_i} = \prod_{\substack{k=0 \\ k \neq i \mod p+1}}^{p} (x_{d_i} - \frac{k}{p}) \prod_{\substack{k=0 \\ k \neq \left\lfloor \frac{i}{p+1} \right\rfloor}}^{p} (y_{d_i} - \frac{k}{p}) \tag{4.4}$$

where again $\bar{\mathbf{d}}_i = (x_{d_i}, y_{d_i})^T$. It can be shown that $\bar{b}_i(x,y)$ chosen according to 4.9 satisfies the nodal condition 4.1 and that it lies in the space $\mathcal{Q}_p(\mathbb{R}^2)$. Therefore it will be the exact same function as we would have computed using the first method. Using 4.9 and 4.4 we can write an algorithm that computes shape functions of arbitrary order. But this is not all, yet, we also need the gradients of these shape functions, so we apply the product rule and get

$$\nabla\bar{b}_i(x,y) = a_i \begin{pmatrix} (b_i^x)'(x) b_i^y(y) \\ b_i^x(x)(b_i^y)'(y) \end{pmatrix}$$

$$= a_i \begin{pmatrix} \sum\limits_{\substack{j=0 \\ j\neq i \mod p+1}}^{p} \prod\limits_{\substack{k=0 \\ k\neq j \\ k\neq i \mod p+1}}^{p} (x - \frac{k}{p}) \prod\limits_{\substack{k=0 \\ k\neq \left\lfloor \frac{i}{p+1} \right\rfloor}}^{p} (y - \frac{k}{p}) \\ \prod\limits_{\substack{k=0 \\ k\neq i \mod p+1}}^{p} (x - \frac{k}{p}) \sum\limits_{\substack{j=0 \\ j\neq \left\lfloor \frac{i}{p+1} \right\rfloor}}^{p} \prod\limits_{\substack{k=0 \\ k\neq j \\ k\neq \left\lfloor \frac{i}{p+1} \right\rfloor}}^{p} (y - \frac{k}{p}) \end{pmatrix} \tag{4.5}$$

where the a_i can be calculated as above. These shape functions were implemented in the file `very_high_order_regular_shapefunctions.hpp`. Listing 4.7 shows the signature of these function.

Listing 4.7: Arbitrary Order Shape Functions and Gradients for Quadrilateral Element

```
typedef Eigen::Matrix<numeric_t, 2, 1> coords_t;
template<index_t order>
Eigen::Matrix<numeric_t, -1, 1> bi_quad
                          (const coords_t & x);
template<index_t order>
Eigen::Matrix<numeric_t, -1, 2> grad_bi_quad
                          (const coords_t & x);
```

The first one, `bi_quad<order>(x)` returns a length $N_{LD} = (order + 1)^2$ vector whose entries are all the shape functions of the quadrilateral reference element with corresponding `order` evaluated at point `x`. `grad_bi_quad<order>(x)` returns, as the name suggests, the gradients transposed of the shape functions, all evaluated at point `x` in a N_{LD} by 2 matrix.

4.3.2 Triangular elements

We again have the same two possibilities, solving a LSE or constructing the shape functions explicitly. The first case is very similar to what was done for the quadrilateral element, the only difference is that we have another number of interpolation nodes $N_{LD} := \frac{(p+1)(p+2)}{2}$. So we choose $\bar{\mathbf{b}}_j \in \mathcal{P}_p(\mathbb{R}^2)$, where $\mathcal{P}_p(\mathbb{R}^2)$ is the space of multivariate polynomials, also found in [4, Chapter 3, Section 3.4.2]. Then again the nodal property 4.1 leads to a linear system of equations which can be solved easily. And of course this was done and the signatures of the corresponding functions can be found in listing 4.8.

Listing 4.8: Low Order Shape Functions and Gradients for Triangular Element

```
typedef Eigen::Matrix<numeric_t, 2, 1> coords_t;
Eigen::Matrix<numeric_t, -1, 1>
                        b1_tria(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 2>
                        gradb1_tria(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 1>
                        b2_tria(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 2>
                        gradb2_tria(const coords_t & x);
/// ...
Eigen::Matrix<numeric_t, -1, 1>
                        b6_tria(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 2>
                        gradb6_tria(const coords_t & x);
```

These functions are also found in the file `shape_functions_tria.hpp`. But as in the case of the quadrilateral, we still need shape functions of arbitrary high order. On the triangular reference element, they are even more complicated to compute in a general way than the ones on the quadrilateral reference element, but basically the derivations are similar, one can just write down the shape functions in terms of linear factors.

$$\begin{aligned}\bar{b}_{i,j}(x,y) &= a_{i,j} b_i^x(x) b_j^y(y) b_{i,j}^{xy}(x,y) \\ &= a_{i,j} \prod_{k=0}^{i-1}(x-\frac{k}{p}) \prod_{k=0}^{j-1}(y-\frac{k}{p}) \prod_{k=i+j-1}^{p}(x+y-\frac{k}{p})\end{aligned} \tag{4.6}$$

To be able to write this down relatively easily, we used shape functions $\bar{b}_{i,j}(x,y)$ with two subscripts, i and j. This means that $\bar{b}_{i,j}(x,y)$ is the shape function corresponding to interpolation node

$$\bar{\mathbf{d}}_{i,j} = (x_{d_{i,j}}, y_{d_{i,j}})^T = (\frac{i}{p}, \frac{j}{p})^T \text{ for } 0 \leq i,j \leq p, i+j \leq p \tag{4.7}$$

An example of the ordering of the interpolation nodes used in the implementation can be seen in figure 4.10, which shows the interpolation nodes of a triangular element of order 5. It is similar to the ordering of the interpolation nodes of the quadrilateral element.

The normalization constant $a_{i,j}$ can be computed as

$$\begin{aligned}\frac{1}{a_{i,j}} &= \bar{\mathbf{b}}_{i,j}(x_{d_{i,j}}, y_{d_{i,j}}) \\ &= \prod_{k=0}^{i-1}(x_{d_{i,j}}-\frac{k}{p}) \prod_{k=0}^{j-1}(y_{d_{i,j}}-\frac{k}{p}) \prod_{k=i+j-1}^{p}(x_{d_{i,j}}+y_{d_{i,j}}-\frac{k}{p})\end{aligned} \tag{4.8}$$

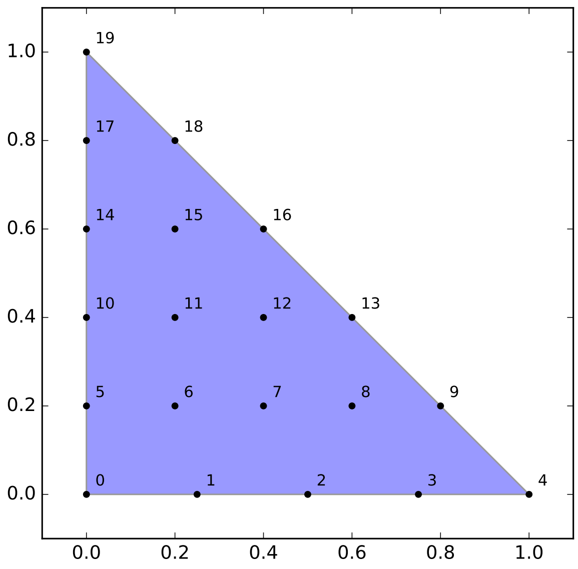


Figure 4.10: Interpolation Node Ordering of a Triangular Element of order 5.

The gradient of the shape functions can directly be derived using the product rule

$$
\begin{aligned}
\nabla \bar{b}_{i,j}(x,y) &= a_{i,j} \begin{pmatrix} b^y_j(y)[(b^x_i)'(x) b^{xy}_{i,j}(x,y) + b^x_i(x) \frac{\partial}{\partial x} b^{xy}_{i,j}(x,y)] \\ b^x_i(x)[(b^y_i)'(y) b^{xy}_{i,j}(x,y) + b^y_i(y) \frac{\partial}{\partial y} b^{xy}_{i,j}(x,y)] \end{pmatrix} \\
&= a_{i,j} \begin{pmatrix} \prod_{k=0}^{j-1}(y - \frac{k}{p})[\prod_{k=i+j-1}^{p}(x+y-\frac{k}{p}) \sum_{l=0}^{i-1} \prod_{\substack{k=0 \\ k\neq l}}^{i-1}(x-\frac{k}{p}) \\ \quad + \prod_{k=0}^{i-1}(x-\frac{k}{p}) \sum_{l=i+j-1}^{p} \prod_{\substack{k=0 \\ k\neq l}}^{i-1}(x+y-\frac{k}{p})] \\ \prod_{k=0}^{i-1}(x - \frac{k}{p})[\prod_{k=i+j-1}^{p}(x+y-\frac{k}{p}) \sum_{l=0}^{j-1} \prod_{\substack{k=0 \\ k\neq l}}^{j-1}(y-\frac{k}{p}) \\ \quad + \prod_{k=0}^{j-1}(y-\frac{k}{p}) \sum_{l=i+j-1}^{p} \prod_{\substack{k=0 \\ k\neq l}}^{i-1}(x+y-\frac{k}{p})] \end{pmatrix}
\end{aligned}
\tag{4.9}
$$

These shape functions and their gradients are implemented in the same file and are named similarly as the corresponding functions for the quadrilateral elements which shows listing 4.9.

Listing 4.9: Arbitrary Order Shape Functions and Gradients for Triangular Element

```
typedef Eigen::Matrix<numeric_t, 2, 1> coords_t;
```

```
template<index_t order>
Eigen::Matrix<numeric_t, -1, 1> bi_tria
                                (const coords_t & x);
template<index_t order>
Eigen::Matrix<numeric_t, -1, 2> grad_bi_tria
                                (const coords_t & x);
```

4.3.3 Transition Elements

It was already mentioned that our mesh contains elements of different orders, this means that we have elements with edges that do not all have the same number of interpolation nodes on them. For simplicity, we only allow elements of this type that have exactly *one* edge that has exactly *one* interpolation node less than the others. Then we define our reference element to have this missing interpolation node on the first edge which lies at $y = 0$. Let's assume our element has order p, this means that on all edges but on the first there lie $p + 1$ interpolation nodes. For the case of an element with $p = 5$ it looks as in figure 4.11.

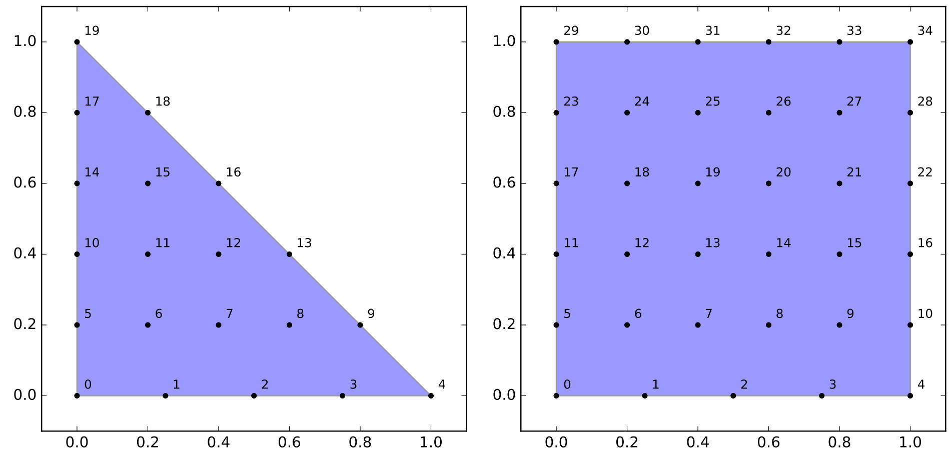


Figure 4.11: Interpolation Node Ordering of a Triangular and a Quadrilateral Transition Element of order 5 .

Let $N_{LD}^{p,1} = N_{LD}^{p} - 1$ be the number of local interpolation nodes of such a transition element of order p, $N_{LD}^{p,1} = (p+1)^2 - 1$ for quadrilateral and $N_{LD}^{p,1} = \frac{(p+1)(p+2)}{2} - 1$ for triangular transition elements. Let $\bar{b}_i^p$ be the i-th shape function of a regular order p element and $\bar{b}_i^{p,1}$ be the shape functions of a transition element of order p, with the 1 indicating the missing interpolation node. Since the regular order p shape function which do *not* lie on the first edge are identical to zero on the whole mentioned edge, we can choose these also for the transition element, i.e. $\bar{b}_i^{p,1} = \bar{b}_{i+1}^{p}$ for $i = p, ..., N_{LD}^{p,1}$. The remaining shape functions are constructed as a linear combination of the shape functions used just before and the shape functions of a regular order $p-1$ element wich lie on the $y=0$ edge. If we choose for $i = 0, ..., p-1$

$$\bar{b}_i^{p,1} = \bar{b}_i^{p-1} - \sum_{k=p}^{N_{LD}^{p,1}} \bar{b}_i^{p-1}(d_i^{p,1})\bar{b}_i^{p,1} \tag{4.10}$$

then it can be shown that these $\bar{b}_i^{p,1}$ fulfill the nodal property 4.1 and that these polynomials are of order $p-1$ if we restrict them to $y=0$ which ensures continuity. This procedure can be used to construct shape functions for the quadrilateral as well as for the triangular transition element from the regular shape functions. An algorithm doing exactly this is also implemented in the file `very_high_order_regular_shapefunctions.hpp` and the functions take the following forms

Listing 4.10: Arbitrary Order Shape Functions and Gradients for Transition Elements

```
typedef Eigen::Matrix<numeric_t, 2, 1> coords_t;
template<index_t order>
Eigen::Matrix<numeric_t, -1, 1> bi_1_tria
                                (const coords_t & x);
template<index_t order>
Eigen::Matrix<numeric_t, -1, 1> grad_bi_1_tria
                                (const coords_t & x);
template<index_t order>
Eigen::Matrix<numeric_t, -1, 2> bi_1_quad
                                (const coords_t & x);
template<index_t order>
Eigen::Matrix<numeric_t, -1, 2> grad_bi_1_quad
                                (const coords_t & x);
```

where the 1 indicates the missing interpolation node and `quad` is used for quadrilateral and `tria` for triangular elements. The name and the purpose of these functions is analogous to the previous cases. It is also possible to compute the shape functions directly from the nodal property by solving a LSE as discussed for the regular shape functions, this was done for transition elements with low order, precisely for $p = 2, ..., 6$ as seen in listing 4.11.

Listing 4.11: Low Order Shape Functions and Gradients for Transition Elements

```
typedef Eigen::Matrix<numeric_t, 2, 1> coords_t;
Eigen::Matrix<numeric_t, -1, 1>
```

```
                    b2_1_tria(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 2>
                    gradb2_1_tria(const coords_t & x);
/// ...
Eigen::Matrix<numeric_t, -1, 1>
                    b6_1_tria(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 2>
                    gradb6_1_tria(const coords_t & x);
/// And the same for quadrilaterals
Eigen::Matrix<numeric_t, -1, 1>
                    b1_1_quad(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 2>
                    gradb1_1_quad(const coords_t & x);
/// ...
Eigen::Matrix<numeric_t, -1, 1>
                    b6_1_quad(const coords_t & x);
Eigen::Matrix<numeric_t, -1, 2>
                    gradb6_1_quad(const coords_t & x);
```

4.4 Local transformations

Since the basis functions are only available on the reference element and the same is true for the quadrature points, we need to transform the integral over an arbitrary element to the corresponding reference element. To do so a suitable transformation from the reference element to an arbitrary element in the mesh is needed. This transformation will be denoted as φ_i, it transforms the reference element to the element $\hat{K}_i$ in the mesh of the unit circle.

4.4.1 Affine and bilinear transformations

The mesh of the unit disk constructed as described in 4.2 only contains elements with straight edges, this means that we do not need any complicated transformations, we can just use the affine transformation for triangular elements and the bilinear transformation for quadrilateral ones. The affine transformation of the reference element to a specific triangular element in the mesh with corners $\hat{\mathbf{d}}_j, j = 1, 2, 3$ is defined as follows

$$\varphi_i(\bar{\mathbf{x}}) := \hat{\mathbf{d}}_0 + \begin{pmatrix} \hat{\mathbf{d}}_1 - \hat{\mathbf{d}}_0 & \hat{\mathbf{d}}_2 - \hat{\mathbf{d}}_0 \end{pmatrix} \bar{\mathbf{x}} \tag{4.11}$$

It is easy to see that the corners of the reference element are mapped to the corners of the element in the mesh, $\hat{\mathbf{d}}_j$. The Jacobian of this transformation is

$$D\varphi_i(\bar{\mathbf{x}}) := \begin{pmatrix} \hat{\mathbf{d}}_1 - \hat{\mathbf{d}}_0 & \hat{\mathbf{d}}_2 - \hat{\mathbf{d}}_0 \end{pmatrix} \tag{4.12}$$

The bilinear transformation of the reference element $\hat{K}_Q$ to a quadrilateral element in the mesh with corners $\hat{\mathbf{d}}_j, j = 1,2,3,4$ is given as

$$\varphi_i(\bar{\mathbf{x}}) := \hat{\mathbf{d}}_0 + \begin{pmatrix}\hat{\mathbf{d}}_1 - \hat{\mathbf{d}}_0 & \hat{\mathbf{d}}_3 - \hat{\mathbf{d}}_0\end{pmatrix} \bar{\mathbf{x}} + \begin{pmatrix}\hat{\mathbf{d}}_2 - \hat{\mathbf{d}}_0 - \hat{\mathbf{d}}_1 - \hat{\mathbf{d}}_3\end{pmatrix} \bar{x}\bar{y} \tag{4.13}$$

Its Jacobian is given as

$$D\varphi_i(\bar{\mathbf{x}}) := \begin{pmatrix}\hat{\mathbf{d}}_1 - \hat{\mathbf{d}}_0 & \hat{\mathbf{d}}_2 - \hat{\mathbf{d}}_0\end{pmatrix} + \begin{pmatrix}\bar{y} \\ \bar{x}\end{pmatrix} \begin{pmatrix}\hat{\mathbf{d}}_2 - \hat{\mathbf{d}}_0 - \hat{\mathbf{d}}_1 - \hat{\mathbf{d}}_3\end{pmatrix}^T \tag{4.14}$$

4.4.2 Higher order transformations

One possibility is to parametrize the transformation needed by the shape functions of the reference elements in the following way.

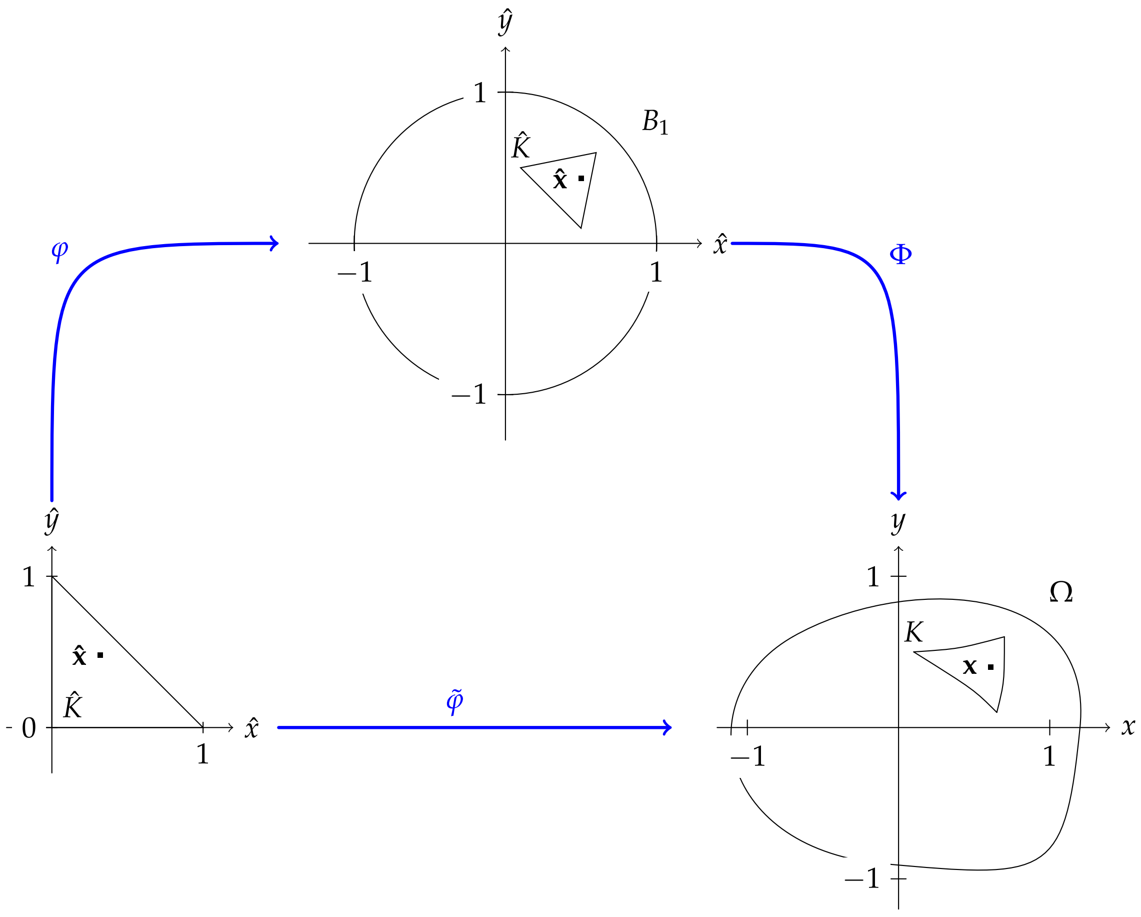
$$\varphi_i(\bar{\mathbf{x}}) := \sum_{j=1}^{N_{LD}} \hat{\mathbf{d}}_j \bar{b}_i(\bar{\mathbf{x}}) \tag{4.15}$$

Here N_{LD} denotes the number of local interpolation nodes in the element and $\hat{\mathbf{d}}_j$ is the j-th interpolation node (coordinates) of the element in the mesh. It is easy to see that this transformation maps the interpolation nodes of the reference element $\bar{\mathbf{d}}_j$ to the interpolation nodes of the element in the mesh $\hat{\mathbf{d}}_j$. The Jacobian of this transformation can be easily computed from the gradients of the shape functions.

$$D\varphi_i(\bar{\mathbf{x}}) := \sum_{j=1}^{N_{LD}} \hat{\mathbf{d}}_j \nabla \bar{b}_i(\bar{\mathbf{x}})^T \tag{4.16}$$

4.5 FE Solution

As mentioned in section 4.3, we need to map the integral for every element to an appropriate reference element to be able to compute it. The following figure shows the setting for a triangular element.



4.5.1 Auxiliary problem

To solve the auxiliary problem 3.6 numerically, we again use transformation techniques described e.g. in [4, Chapter 3, Section 3.7.3]. So we get

$$
\begin{aligned}
(A)_{i,j} &= \int\limits_{B_1} \nabla \hat{b}_i \cdot \nabla \hat{b}_j d\hat{x} \\
&= \sum_{k=1}^{M} \int\limits_{\hat{K}_k} \nabla \hat{b}_i \cdot \nabla \hat{b}_j d\hat{x} \\
&= \sum_{k=1}^{M} \int\limits_{\bar{K}} D\varphi_k^{-T} \nabla \bar{b}_i \cdot D\varphi_k^{-T} \nabla \bar{b}_j |\det D\varphi_k| d\bar{x}
\end{aligned}
\tag{4.17}
$$

where φ_k maps the reference element $\bar{K}$ to $\hat{K}_k$ which denotes the k-th element of the mesh of the unit disk. As seen in section 4.4, we can express φ_k in terms of the shape functions on the reference element. This means that we can compute the above expression for the entries of the system matrix using quadrature.

4.5.2 Original problem

For the original problem this is not as easy. One difficulty is that we already transformed the integral using Φ, but from this Φ only an approximation, denoted by Φ^N, is available as a numerical solution to the auxiliary problem. Therefore it can't be easily evaluated at any point in B^1. We want to compute

$$\begin{aligned}(A^{\hat{u}})_{i,j} &= \int\limits_{\Omega} \nabla b_i \cdot \nabla b_j dx \\ &= \int\limits_{B_1} (D\Phi^N)^{-T} \nabla \hat{b}_i \cdot (D\Phi^N)^{-T} \nabla \hat{b}_j |\det D\Phi^N| d\hat{x} \; i,j = 1,...,N\end{aligned} \tag{4.18}$$

But again, we have to transform this to a sum of integrals over reference elements only. We will look at two different, but mathematically equivalent ways to do this.

Direct Method

If we use the first part of the above expression 4.18, we get:

$$\begin{aligned}(A^{\hat{u}})_{i,j} &= \int\limits_{\Omega} \nabla b_i \cdot \nabla b_j dx \\ &= \sum_{k=1}^{M} \int\limits_{K_k} \nabla b_i \cdot \nabla b_j dx \\ &= \sum_{k=1}^{M} \int\limits_{\bar{K}} D\tilde{\varphi}_k^{-T} \nabla \bar{b}_i \cdot D\tilde{\varphi}_k^{-T} \nabla \bar{b}_j |\det D\tilde{\varphi}_k| d\bar{x}\end{aligned} \tag{4.19}$$

where K_k denotes the k-th element of the deformed mesh and $\tilde{\varphi}_k = \Phi^N(\varphi_k(\bar{\mathbf{x}}))$ with φ_k defined in the previous subsection. This means that $\tilde{\varphi}_k$ maps the reference element directly to the element of the deformed mesh. So to be able to compute the entries of $A^{\hat{u}}$, we only need $D\tilde{\varphi}_k$.

Recall that the problem is Φ^N, which is partly only given through a FE solution, i.e. $\Phi^N(\hat{\mathbf{x}}) = \hat{\mathbf{x}} + \mathbf{w}^N(\hat{\mathbf{x}})$ with $\mathbf{w}^N(\hat{\mathbf{x}}) := \sum_{i=1}^{N} \begin{pmatrix} \alpha_{1,i} \\ \alpha_{2,i} \end{pmatrix} \hat{b}_i(\hat{\mathbf{x}})$. We have:

$$\begin{aligned} \tilde{\varphi}_k(\bar{\mathbf{x}}) &= \Phi^N(\varphi_k(\bar{\mathbf{x}})) = \varphi_k(\bar{\mathbf{x}}) + \mathbf{w}^N(\varphi_k(\bar{\mathbf{x}})) \\ &= \varphi_k(\bar{\mathbf{x}}) + \sum_{i=1}^{N} \begin{pmatrix} \alpha_{1,i} \\ \alpha_{2,i} \end{pmatrix} \hat{b}_i(\varphi_k(\bar{\mathbf{x}})) \\ &= \varphi_k(\bar{\mathbf{x}}) + \sum_{i=1}^{N} \begin{pmatrix} \alpha_{1,i} \\ \alpha_{2,i} \end{pmatrix} \bar{b}_i(\bar{\mathbf{x}}) \end{aligned} \tag{4.20}$$

And from this it follows immediately.

$$D\tilde{\varphi}_k = D\varphi_k + \sum_{i=1}^{N} \begin{pmatrix} \alpha_{1,i} \\ \alpha_{2,i} \end{pmatrix} \nabla \bar{b}_i^T \tag{4.21}$$

This expression can be computed since it only contains the shape functions on the reference element and the transformation which can be expressed in terms of them. If we plug this into 4.19, we can compute the entries of $A^{\hat{u}}$ using quadrature.

Mesh Deformation

This method is essentially the same as the previous one. The idea is that we take the interpolation nodes of the mesh of the unit circle and thansform them using Φ^N, in this way we just get a mesh of Ω. Let $\mathbf{d}_i := \Phi^N(\hat{\mathbf{d}}_i) = \hat{\mathbf{d}}_i + \begin{pmatrix} \alpha_{1,i} \\ \alpha_{2,i} \end{pmatrix}$ for $i = 1, ..., N$. Since the $\hat{\mathbf{d}}_i$ denote the interpolation nodes of the mesh of the unit circle, the $\mathbf{d}_i$ are the interpolation nodes of a mesh of the original domain Ω with the same connectivity. This means that we can directly compute $\tilde{\varphi}_k$ and $D\tilde{\varphi}_k$ using the modified interpolation nodes . Thus we can compute the entries of our system matrix according to 4.19 in a similar way as we did for the auxiliary problem.

4.6 Deformed Elements and Interpolation Node Ordering

4.6.1 Interpolation Node Ordering

In section 4.3 the ordering of the interpolation nodes of all element types is shown for some example elements. Additionally, all the elements in the mesh of the unit disk are constrained to have the same orientation, i.e. there are no elements that are flipped. This is handled by the mesh constructor, all the elements of the constructed mesh will have the orientation of the elements shown in the section mentioned above. Problems might arise if we apply the mesh deformation technique described in section 4.5, the computed deformation might switch the orientation of some elements. Therefore we need to detect such deformation since this will prevent convergence.

4.6.2 Detecting Bad Deformations

The orientation constraint defined above makes this task quite easy. Since all the elements have the same orientation and it is also the same as the one of the reference elements, we only have to check if the determinant of the transformation from the reference element to the element in the deformed mesh is positive, i.e. if

$$\det D\tilde{\varphi}_k(\bar{\mathbf{x}}) > 0 \tag{4.22}$$

for all $\bar{\mathbf{x}}$ within the corresponding reference element. If this is the case, then there's no problem, else our computations will fail. Since the mesh of the unit disk does not contain such badly deformed elements, it is enough to test if

$$\det D\Phi^N(\hat{\mathbf{x}}) > 0 \tag{4.23}$$

for all $\hat{\mathbf{x}}$ within the corresponding element in the mesh of the unit disk. In the code inequality 4.22 is tested at all quadrature points since there we need to evaluate $\det D\tilde{\varphi}_k$ anyways for the assembly. In this way this won't slow down the computations a lot.

4.7 Quadrature

With all these transformations our problem boils down to computing integrals over the reference elements. To do so we use numerical quadrature. The `class` containing all necessary information about a specific quadrature rule looks as follows

Listing 4.12: Quadrature Rule Class

```
typedef unsigned short el_type_t;
typedef unsigned short n_quad_p_t;
typedef Eigen::Matrix<numeric_t, -1, 1> quad_weights_t;
typedef Eigen::Matrix<numeric_t, -1, 2> quad_points_t;
struct quadRule {
    const el_type_t el_type;
    const n_quad_p_t n;
    const quad_weights_t weights;
    const quad_points_t points;
    quadRule(const el_type_t el_type,
                n_quad_p_t n,
                quad_weights_t weights,
                const quad_points_t points){...}
};
```

where n denotes the number of quadrature points, points is a n by 2 matrix storing the quadrature points and weights a length n vector storing the corresponding weights. el_type determines whether the quadrature rule corresponds to the triangular reference element, in this case el_type $= 3$, or to the quadrilateral one, i.e. el_type $= 4$.

4.7.1 Quadrature Rules for Quadrilateral Reference Element

The specific quadrature rules for the quadrilateral reference element are implemented as seen in listing 4.13.

Listing 4.13: Quadrature Rules for Quadrilaterals Elements

```
typedef unsigned short quad_ord_t;
quadRule quad_qr(const quad_ord_t k){
    if(k == 1){
        quad_weights_t weights(1);
        quad_points_t points(1,2);
        weights <<  1.;
        points <<   0.5, 0.5;
        return quadRule(4,1,weights,points);
    } else if(k == 2){
        quad_weights_t weights(4);
        quad_points_t points(4,2);
        weights <<  0.25,
                    0.25,
                    0.25,
                    0.25;
        points <<   0.2113248654051871, 0.2113248654051871,
                    0.2113248654051871, 0.7886751345948129,
                    0.7886751345948129, 0.2113248654051871,
                    0.7886751345948129, 0.7886751345948129;
        return quadRule(4,4,weights,points);
    }
    /// And so forth
}
```

quad_qr takes an integer k as an input and returns a quadrature rule which is exact for polynomials in $\mathcal{Q}_{2k-1}(\mathbb{R}^2)$. This code returning quadrature rules was created from 1D Gaussian quadrature rules implemented in Mathematica [6] in the function GaussianQuadratureWeights from the package NumericalDifferentialEquationAnalysis.

4.7.2 Quadrature Rules for Triangular Reference Element

For the triangular reference element it looks quite the same as can be seen in listing 4.14.

Listing 4.14: Quadrature Rules for Quadrilaterals Elements

```
typedef unsigned short quad_ord_t;
quadRule tria_qr(const quad_ord_t k){
    if(k == 1){
        quad_weights_t weights(1);
        quad_points_t points(1,2);
        weights <<  1;
        points <<  0.3333333333333298,0.3333333333333298;
        return quadRule(3, 1, weights, points);
    } else if(k == 2){ // # of points: 3
        quad_weights_t weights(3);
        quad_points_t points(3,2);
        weights <<  0.33333333333333298,
                    0.33333333333333298,
                    0.33333333333333298;
        points <<  0.66666666666666696,0.16666666666666699,
                    0.16666666666666699,0.16666666666666699,
                    0.16666666666666699,0.66666666666666696;
        return quadRule(3, 3, weights, points);
    }
    /// And so forth
}
```

The main difference is that the function `tria_qr` with input `k` returns a quadrature rule which is exact for polynomials in $\mathcal{P}_k(\mathbb{R}^2)$. The quadrature rules implemented here are derived in [2] and the implementation found here [1] was used to construct an implementation in Eigen [3].

4.7.3 Choosing Quadrature Order

Let's consider an element of order p. In this case, if we compute the integral over the corresponding reference element, we use a quadrature rule of order $2p$, i.e. one that it is exact for polynomials of order up to $2p-1$. Note that these polynomials need to be chosen from a suitable space according to whether the element is triangular or quadrilateral. Since there is a difference in the implementation of the quadrature rules for the triangular and the quadrilateral element, we have to choose `k` in the code in the following way. `k` $= 2p-1$ for triangular elements and `k` $= p$ for quadrilateral elements, each of order p.

4.8 Error Computation

To measure the convergence behavior of our problem, we need to compute the error between our numerical solution and an exact solution. Let u be the exact solution that can be evaluated anywhere on our domain and $u^N = \sum\limits_i \mu_i b_i$ the FE approximation.

4.8.1 L^2-Error

The L^2-error is defined as follows.

$$||u - u^N||^2_{L^2(\Omega)} = \int\limits_\Omega (u - u^N)^2 dx \tag{4.24}$$

The problem here is that we cannot evaluate the discrete solution u^N at any point in Ω and we also do not have a quadrature rule for our domain Ω. We proceed in a similar way as we did when we computed the entries of the system matrix, we split the domain into finite elements.

$$\begin{aligned} ||u - u^N||^2_{L^2(\Omega)} &= \int\limits_\Omega (u(x) - u^N(x))^2 dx \\ &\approx \sum_k \int\limits_{K_k} (u(x) - u^N(x))^2 dx \\ &= \sum_k \int\limits_{\bar{K}} (u(\tilde{\varphi}_k(\bar{x})) - u^N(\tilde{\varphi}_k(\bar{x})))^2 |det D\tilde{\varphi}_k(\bar{x})| d\bar{x} \end{aligned} \tag{4.25}$$

We plug in the discrete solution as $u^N = \sum\limits_i \mu_i b_i(x)$ and get the following.

$$\begin{aligned} ||u - u^N||^2_{L^2(\Omega)} &= \sum_k \int\limits_{\bar{K}} (u(\varphi_k(\bar{x})) - \sum_i \mu_i b_i(\varphi_k(\bar{x})))^2 |det D\tilde{\varphi}_k(\bar{x})| d\bar{x} \\ &= \sum_k \int\limits_{\bar{K}} (u(\varphi_k(\bar{x})) - \sum_i \mu_i \bar{b}_i(\bar{x}))^2 |det D\tilde{\varphi}_k(\bar{x})| d\bar{x} \end{aligned} \tag{4.26}$$

In this way the error can actually be computed using an appropriate quadrature rule since the transformation $\tilde{\varphi}_k$ is the same as we saw in section 4.5 when the assembly of the system matrix was discussed. Note that in the inner sum indices i only needs to be considered if $\mathbf{d}_i$ is contained in element k, this means the error computation can be done element-wise.

4.8.2 H^1-Error

Equivalently we derive for the H^1-error the following.

$$\begin{aligned}
||u-u^N||^2_{H^1(\Omega)} &= \int\limits_{\Omega} \nabla(u-u^N)\cdot\nabla(u-u^N)dx \\
&= \int\limits_{\Omega} ||\nabla(u-u^N)||_2^2 dx \\
&= \int\limits_{\Omega} ||\nabla u-\nabla u^N||_2^2 dx \\
&\approx \sum_k \int\limits_{K_k} ||\nabla u-\nabla u^N||_2^2 dx \\
&= \sum_k \int\limits_{\bar{K}} ||\nabla u(\tilde{\varphi}_k(\bar{x})) - \sum_i \mu_i D\tilde{\varphi}_k^{-T}\nabla\bar{b}_i(\bar{x})||_2^2 |detD\tilde{\varphi}_k(\bar{x})| d\bar{x}
\end{aligned} \tag{4.27}$$

Of course we need the exact gradient ∇u to compute this error.

4.8.3 Boundary Approximation

The error arising from the approximation of the boundary is *not* taken into account in this thesis. One can see that in the previous two error computations there was an approximation even without using quadrature, we just assumed that the elements cover the domain Ω exactly, the error on the part of Ω which is not covered by elements is just ignored to make the implementation easier.

Chapter 5

Results

5.1 Convergence on Unit Disk

To validate the code, we conduct a convergence study on the unit disk. Thereby we solve the following problem

$$\begin{aligned} -\Delta u &= 0 \text{ on } B_1 \\ u &= g \text{ on } \partial B_1 \end{aligned} \tag{5.1}$$

To measure the convergence, we choose the Dirichlet boundary condition as a harmonic function, in that way the sought after function u is identical to the boundary condition g.

5.1.1 Constant order element mesh

To verify that the code works for high order elements, we use a mesh with elements of constant order p and an analytic harmonic function:

$$u(x,y) = g(x,y) = e^y \sin(x) \tag{5.2}$$

That allows us to get an asymptotical error depending only on the order of the elements used. In [4, Chapter 5, Section 5.3.5] and in [4, Chapter 5, Section 5.6.3] the following can be found

$$\begin{aligned} ||u - u_N||_{H^1} &\leq C_1 N^{-\frac{p}{2}} \\ ||u - u_N||_{L^2} &\leq C_2 N^{-\frac{p+1}{2}} \end{aligned} \tag{5.3}$$

for some constants C_1, C_2 and with the number of interpolation nodes N large enough. The mesh used for this first convergence study is described in section 3.1.2, the one containing only triangular elements of constant order. The boundary of the unit circle is approximated piecewise linearly for any order elements, the error from the approximation is neglected for simplicity.

Figure 5.1 shows that the code achieves this exact asymptotic error behavior, in the first case the mesh consisted of linear finite elements and in the second of cubic.

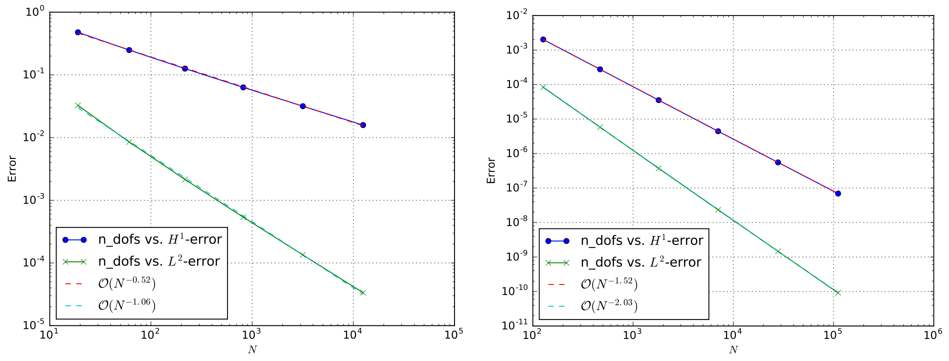


Figure 5.1: Convergence on unit disk using triangular mesh with constant order elements.

This study suggests that the implementation is correct since we get the expected order of convergence.

5.1.2 Hybrid Mesh with Elements of Different Orders

Since we'll use a hybrid mesh containing elements of different orders for the original problem, we also test this mesh with the problem of the previous section. In this way we gain some insights about the convergence we can expect if we treat the original problem. Figure 5.2 shows the convergence behavior of the problem using the hybrid mesh described in section 3.1.2.

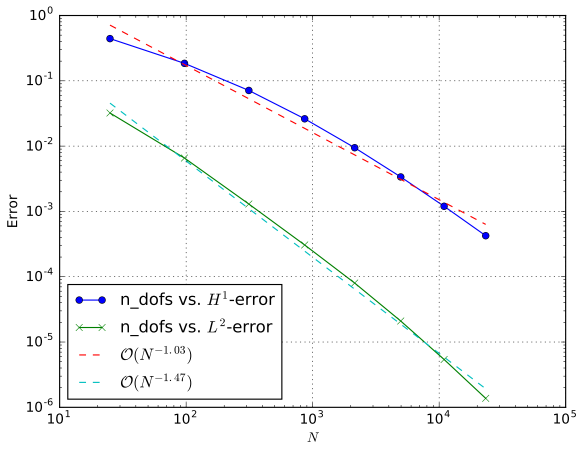


Figure 5.2: Convergence using hybrid mesh.

5.2 Convergence Study

In this section we will look at the original problem posed on our deformed domain Ω using the techniques discussed in 4.5. The problem is the following, as already seen in section 2.1

$$\begin{aligned} -\Delta u &= 0 \text{ on } \Omega \\ u &= g \text{ on } \partial\Omega \end{aligned} \tag{5.4}$$

As in the previous section, we choose the Dirichlet boundary conditions as the smooth harmonic function $g(x,y) = e^y \sin(x)$ for the convergence studies such that the solution will be the same, i.e. $u(x,y) = e^y \sin(x)$ and it will be easy to measure the convergence.

5.2.1 Small Deformation

The first deformation discussed is only small, it is described by the coefficients

$$\mathbf{s} = \begin{pmatrix} 0.1 \\ 0.05 \end{pmatrix}, \mathbf{c} = \begin{pmatrix} 0.06 \\ 0.12 \end{pmatrix}, \mathbf{y} = \begin{pmatrix} 0.6 \\ -0.8 \end{pmatrix} \text{ and } \mathbf{z} = \begin{pmatrix} -0.4 \\ 0.9 \end{pmatrix} \tag{5.5}$$

If the mesh deformation technique from section 4.5 is applied, we find that the mesh deformed according to the coefficients above looks approximately as in figure 5.3

Note that the mesh in figure 5.3. does not represent the true deformation in the interior since the mesh contains elements of up to order 7 and they were actually deformed such that their edges would not be straight anymore, but this was too complicated to visualize. This should just give an impression of what the deformation looks like. Figure 5.4 shows the convergence of the discrete solution for this specific case.

One can see that the convergence behavior for this particular deformation is very similar to the case of no deformation.

5.2.2 Star Shaped Deformation

A second deformation that was tested is determined by the following coefficients.

$$\begin{aligned} s_j &= 0, j = 1, ..., N \\ c_j &= \begin{cases} \frac{3(-1)^{\frac{j}{4}}}{(\frac{j}{4})^2 \pi^2} & \text{if 4 divides } j \\ 0 & \text{else} \end{cases} \\ y_j &= z_j = 1, j = 1, ..., N \end{aligned} \tag{5.6}$$

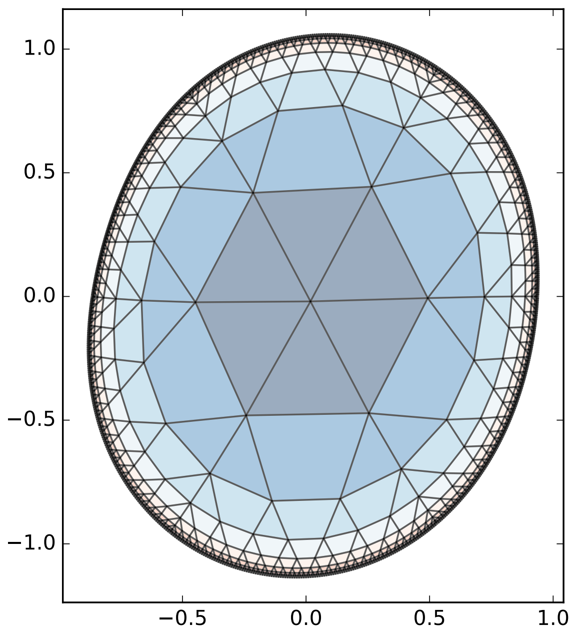


Figure 5.3: Slightly Deformed Mesh

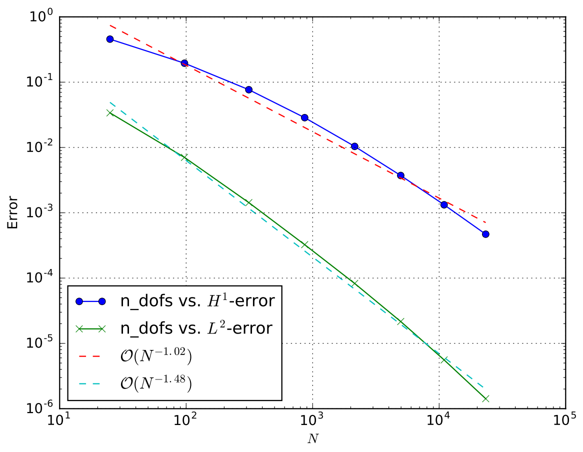


Figure 5.4: Convergence on Slightly Deformed Mesh

If the mesh deformation method is used, the resulting deformed mesh looks as in figure 5.5. In this case $N = 100$ was used.

Despite the rather sharp corners, the algorithm converges with a slightly smaller rate as in the previous case as figure 5.6 shows.

If we solve the same problem using a mesh with elements of constant order we observe the convergence behavior seen in figure 5.11. One can observe

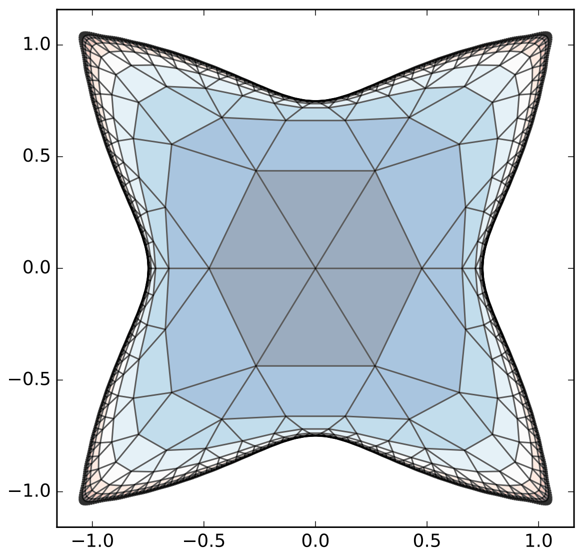


Figure 5.5: Star Shaped Mesh

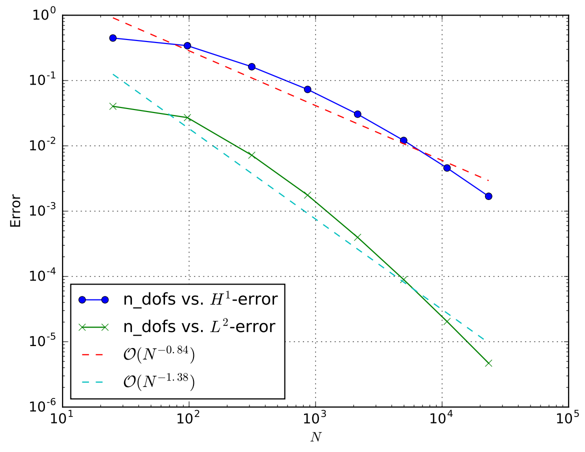


Figure 5.6: Convergence on Star Shaped Mesh

that the asymptotic error behavior is also in this case slightli worse that the optimal behavior observed with the non-deformed meshes.

5.2.3 Deformations Causing Problems

Not all deformed meshes that were tested showed the nice convergence behavior as seen in the previous subsection. The condition 2.4 in section 2.1 does not suffice for the mesh transformation Φ to be bijective, there are cases for which parts of the deformed mesh overlap with themselves. This happens e.g. when using the following coefficients which satisfy the condition

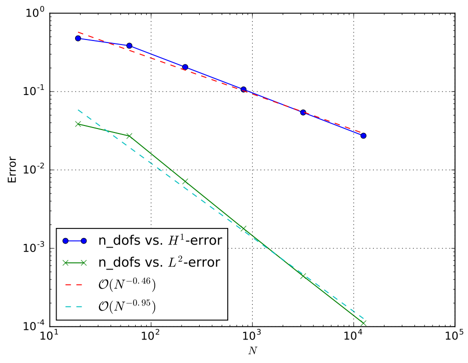


Figure 5.7: Order 1

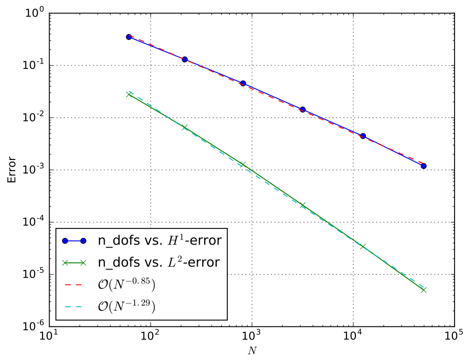


Figure 5.8: Order 2

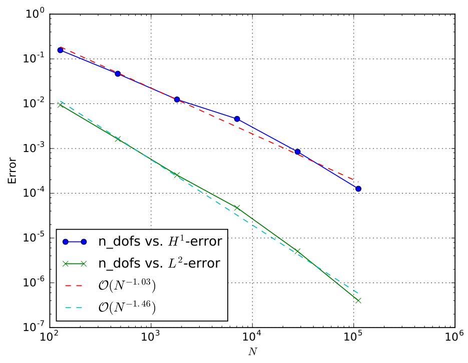


Figure 5.9: Order 3

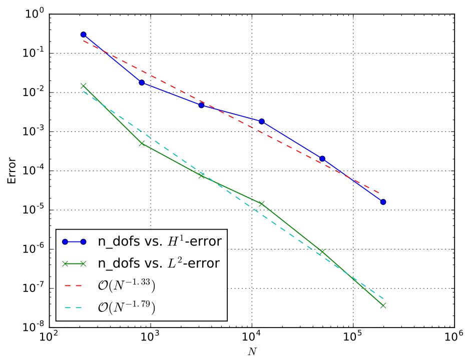


Figure 5.10: Order 4

Figure 5.11: Convergence on star-shaped mesh containing only triangular elements of constant order.

mentioned above.

$$\begin{aligned} s_j &= 0, j = 1, ..., N \\ c_j &= -2 * \frac{1-\cos(dj\pi)}{d(2-d)j^2\pi^2}, d = 0.15, j = 1, ..., N \\ y_j &= z_j = 1, j = 1, ..., N \end{aligned} \tag{5.7}$$

The deformation that these coefficients describe can be seen in figure 5.12.

As one can see the elements of the mesh overlap at the crack on the right. This causes problems, i.e. the solution will not converge as figure 5.13 shows.

We see that if the mesh is fine enough, the error won't decrease any further. Note that this will not be different when using another mesh since overlapping is caused by the definition of the transformation Φ, seen in section 2.1,

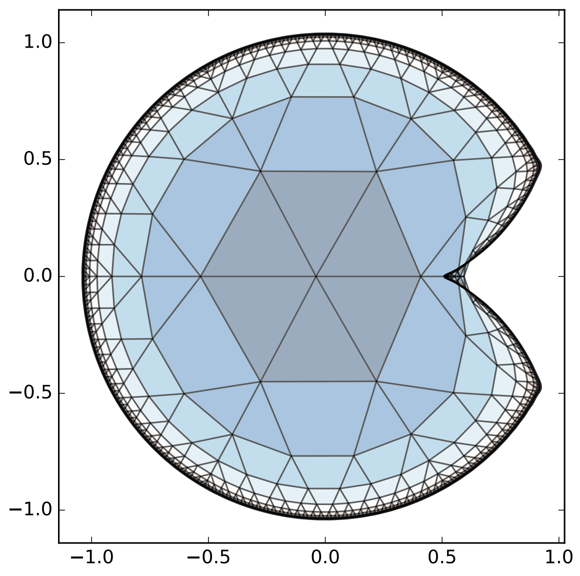


Figure 5.12: Deformed Mesh Causing Problems

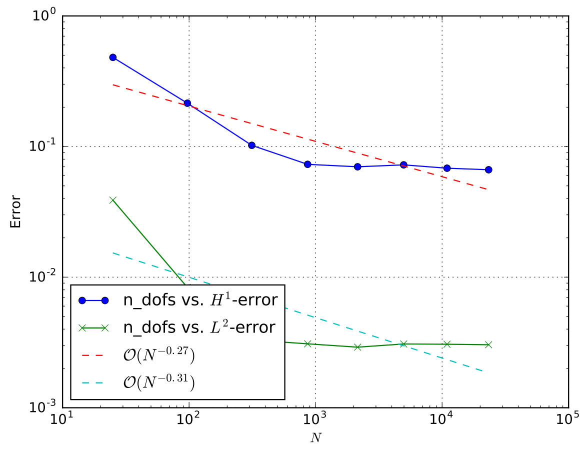


Figure 5.13: Convergence on Badly Deformed Mesh

even if we knew the true solution, this would still lead to the overlapping of the mesh. If we wanted to avoid this problem, we would have to find another method to find a better transformation from B_1 to Ω. This problem can also occur if the re-entrant corner is not as sharp.

Figure 5.14 shows such a case. One could conjecture that if the deformed mesh is convex, then there won't be any problems with the discussed method since this is what was observed in the examples that were tried out for this thesis. But then it might still be hard to find a condition on the parameters describing the deformation that ensure that the resulting domain will be con-

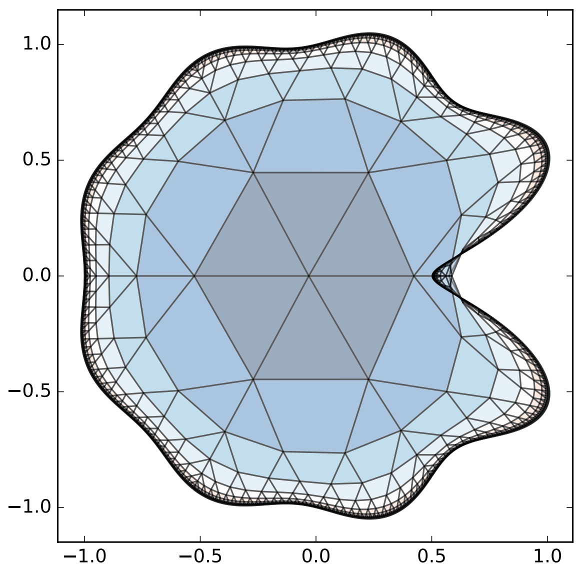


Figure 5.14: Another Deformed Mesh Causing Problems

vex.. Another aspect is that there are non-convex deformations where this method works as was observed for the star shaped mesh from the previous subsection.

5.3 Timing and Efficiency

The code developed for this thesis is based only on Eigen [3], there were no FEM-specific libraries used, these algorithms were all written from scratch. This also means that it is certainly not the most efficient code, since the focus of this thesis was primarily on the convergence analysis and not on the efficiency of the code. Figure 5.15 shows the time needed for the assembly of the matrix for the auxiliary problem, the time needed for the direct assembly as described in 4.5.2 and the time needed for solving the resulting LSE with Eigen's `SimplicialLDLT` direct solver.

One can see that much more time is needed to assemble the matrices than to solve the LSE. It is a bit different if a mesh with only triangular elements of which all have the same order is used. Then the timing looks as in figure 5.16. In this case the mesh contained only elements of order 3.

In this case the assembly is quite a lot faster, though it is still slower than the solving of the LSE.

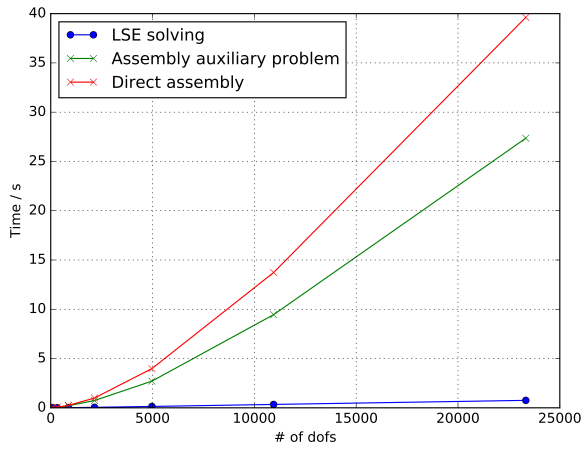


Figure 5.15: Timing using hybrid mesh

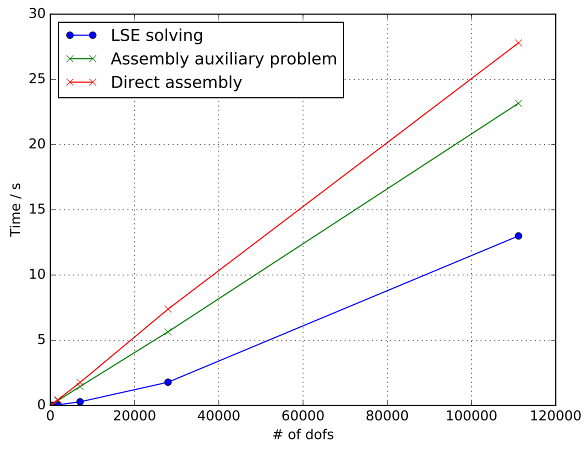


Figure 5.16: Timing using triangular order 3 mesh

5.4 Conclusions

We have discussed a method that allows to find a solution of a Poisson equation on some deformed domain without having to mesh the actual domain by mapping the PDE to the unit disk. It could also be shown empirically that the convergence on the deformed domain can be almost as fast as the one on the unit disk given the deformation is not too severe, in this case, this method can fail, or if the deformation is not too bad, the convergence might just be a bit slower.

5.5 Extensions

Possible extension of this work include the following. The code could easily be modified to also be able to solve Neumann boundary value problem on these deformed domains. Also one could extend the code in such a way that it is possible to solve any elliptic PDE on the deformed domains. The code is already able to solve a Dirichlet BVP on any domain given a suitable mesh, though the meshes would have to be converted to be compatible with this code. What is still unclear is when does the deformation cause problems? One could try random deformations using this code and maybe find out when the deformation is too severe and impose a stronger condition on the coefficients describing the deformation. Perhaps a better way would be to find a way to compute a better mapping from the unit disk to the deformed domain where these deformations do not lead to the overlapping of elements.

There is certainly much more that can be done, but this is enough for this thesis.

Bibliography

[1] John Burkardt. Quadrature rules for the triangle. https://people.sc.fsu.edu/~jburkardt/cpp_src/triangle_dunavant_rule/triangle_dunavant_rule.html, 2006.

[2] D. A. Dunavant. High degree efficient symmetrical gaussian quadrature rules for the triangle, 1985.

[3] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[4] Prof. Dr. R. Hiptmair. Numerical methods for partial differential equations. http://www.sam.math.ethz.ch/~hiptmair/tmp/NPDE/NPDE16.pdf. Last accessed 14 June 2017.

[5] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[6] Wolfram Research, Inc. Mathematica, Version 10.4. Champaign, IL, 2016.

[7] L. Scarabosio. *Shape uncertainty quantification for scattering transmission problems*. Eth dissertation no. 23574, ETH Zürich, 2016.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Dirichlet Boundary Value Problems on Deformed Domains

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Baumann	Christian

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
10.07.2017, Schaffhausen	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.