Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical
Engineering

Polyhedral mesh optimization for better accuracy of numerical computations

Optimalizace polyhedrálních sítí pro zpřesnění numerických výpočtů

Bachelor's Degree Project

Author: **Petr Král**

Supervisor: **doc. Ing. Tomáš Oberhuber, PhD.**

Academic year: 2022/2023

České vysoké učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská

Katedra: matematiky

Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	Petr Král
Studijní program:	Matematické inženýrství
Studijní specializace:	Matematická informatika
Název práce (česky):	Optimalizace polyhedrálních sítí pro zpřesnění numerických výpočtů
Název práce (anglicky):	Polyhedral mesh optimization for better accuracy of numerical computations

Pokyny pro vypracování:

1) Seznamte se s implementací polyhedrálních síti v knihovně TNL.
2) Prostudujte numerickou aproximaci gradientu funkce pomocí metody konečných objemů.
3) Odvoďte optimalizační metodu pro modifikaci sítě za účelem přesnější aproximace gradientu funkcí.
4) Proveďte výpočetní studii a porovnání s jinými podobnými algoritmy.

Doporučená literatura:

1) A. Quarteroni, , R. Sacco, F. Saleri, Numerical mathematics. Springer, 2010.

2) D. P., Bertsekas, Convex optimization algorithms. Athena Scientific, 2015.

3) J. Hahn, K. Mikula, P. Frolkovič, B. Basara, Finite volume method with the soner boundary condition for computing the signed distance function on polyhedral meshes. International Journal for Numerical Methods in Engineering, 123(4), 1057–1077, 2021. https://doi.org/10.1002/nme.6888

4) J. Hahn, K. Mikula, P. Frolkovič et al. Inflow-Based Gradient Finite Volume Method for a Propagation in a Normal Direction in a Polyhedron Mesh. J Sci Comput 72, 442–465, 2017.

Jméno a pracoviště vedoucího bakalářské práce:

doc. Ing. Tomáš Oberhuber, Ph.D.
Katedra matematiky, Fakulta jaderná a fyzikálně inženýrská, ČVUT v Praze, Trojanova 13, 120 00 Praha 2

Jméno a pracoviště konzultanta:

Datum zadání bakalářské práce: 31.10.2022

Datum odevzdání bakalářské práce: 2.8.2023

Doba platnosti zadání je dva roky od data zadání.

V Praze dne 31.10.2022

.....
garant oboru

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská
-3-

.....
vedoucí katedry

.....
děkan

Acknowledgment:

I would like to thank doc. Ing. Tomáš Oberhuber, PhD. for his patient, expert guidance.

Author's declaration:

I declare that this Bachelor's Degree Project is entirely my own work and I have listed all the used sources in the bibliography.

Prague, August 2, 2023 Petr Král

Název práce:

Optimalizace polyhedrálních sítí pro zpřesnění numerických výpočtů

Autor: Petr Král

Studijní program: Matematické inženýrství

Specializace: Matematická informatika

Druh práce: Bakalářská práce

Vedoucí práce: doc. Ing. Tomáš Oberhuber, PhD., Katedra matematiky, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze

Abstrakt: Výsledkem této práce je algoritmus pro optimalizaci sítě za účelem zpřesnění aproximace gradientu pomocí metody konečných objemů za použití knihovny TNL. K dosažení tohoto cíle používáme gradientní sestup a detailněji se věnujeme různým způsobům výpočtu gradientu: analytickému derivování, konečným diferencím a automatickému derivování. Výsledný algoritmus používá konečné diference. Na závěr uvádíme několik výstupů tohoto algoritmu.

Klíčová slova: gradientní sestup, metoda konečných objemů, polyhedrální síť

Title:

Polyhedral mesh optimization for better accuracy of numerical computations

Author: Petr Král

Abstract: In this Bachelor project, we present an algorithm for optimizing triangular meshes to improve finite volume gradient approximation using the TNL library. We employ gradient descent to achieve our goal and explore different methods of evaluating the gradient: analytical differentiation, finite differences, and automatic differentiation. The optimization algorithm we developed utilizes finite differences. We present several outputs of the algorithm.

Key words: finite volume method, gradient descent, polyhedral mesh

Contents

Introduction

In numerical mathematics, it is often necessary to approximate a gradient of a scalar function f. One possible way to achieve this is to discretize the domain Γ of the problem by the means of a mesh and then compute a discrete approximation of ∇f, denoted $\nabla_h f$, based on this mesh.

Various types of meshes exist, some more suitable than others for certain tasks. Our main concern will be triangular and polyhedral meshes. Triangular meshes are easy to manipulate, but computations on them may be slow due to the high amount of cells required to cover certain domains. Polyhedral meshes are more flexible and are often used to discretize more complex domains. This Bachelor project focuses exclusively on two-dimensional triangular meshes because of their inherent simplicity which can be exploited in calculations.

The main objective of this Bachelor project is to optimize polyhedral meshes with respect to the finite volume method gradient approximation. Our goal is to alter the geometry of meshes in such manner that the finite volume approximation differs from the analytical gradient as little as possible. We do not want to change the topology of the meshes at all. We often forbid movement of border vertices. This constraint maintains consistency not only in topology, but also the meshed domain remains the same.

The means to achieving our goal will be mainly the C++ library `TNL`, [4], and specifically its tools for manipulating meshes, [6], [7].

In the first chapter, some necessary pieces of theory are introduced and the objective of the Bachelor project is formulated in mathematical language as an optimization problem with the objective function L, generally referred to as a loss function.

The second chapter introduces gradient descent, both as a method for solving general optimization problems, and as a method tailored specifically for our mesh optimization.

The third chapter then elaborates on different ways of evaluating the gradient ∇L, a crucial element for effective gradient descent.

Continuing with the fourth chapter, we explore implementation of meshes in `TNL`. We also present the code where we implement the solution of the problem as discussed in prior chapters.

Finally, the fifth chapter shows sample computations performed by the programs developed in chapter four. For this, we utilize a mesh generating software, [3], as well as some visualization tools, [10], [11].

Two appendices are also provided. They further enhance the text, but are not essential for understanding its gist.

Chapter 1

Optimization problem

As was stated in the introduction, the goal of this Bachelor project is to optimize a mesh for the finite volume gradient approximation of a function $f : \mathbb{R}^n \to \mathbb{R}$, so that the finite volume approximation is as close to the real (sometimes referred to as "analytical") gradient as possible, under the constraint that topology of the mesh is not altered. Another logical request would be not to alter the meshed domain. This is easily achievable by the means of the library `TNL`.

In other words we minimize the difference of the analytical gradient ∇f and its numerical approximation $\nabla_h f$. Precisely, we calculate

$$\operatorname*{argmin}_{\mathbf{y}\in 2^{\Gamma}} \mathrm{L} \tag{1.1}$$

where the *loss function* L is defined as $\mathrm{L}(\mathbf{y}) \equiv \|\nabla_h f(\mathbf{y}) - \nabla f(\mathbf{y})\|^2$. $\mathbf{y}$ represents all possible meshes on the domain Γ which satisfy other arbitrary constraints. This $\operatorname{argmin} \mathrm{L} = \mathbf{y}^*$ is an abstract representation of an optimal mesh with respect to the finite volume gradient approximation $\nabla_h f$, considering all the constraints employed.

We intend to find this optimal configuration using a gradient descent method which converges to $\mathbf{y}^*$. In order to achieve this, some necessary theory will be presented in the first part of this Chapter. In the rest of it, we will examine the optimization problem closer.

The following definitions and theorems are formulated to suit the scope of this Bachelor project, i.e. not necessarily as generally as possible.

1.1 Unstructured Meshes

Meshes are used in numerical mathematics as means of discretization of a domain. They consist of *cells* which are bordered by *faces*. Faces are manifolds bordered by the *points* (or *vertices*, *nodes*) of the mesh, just like in Figure 1.1. The following definitions are generally based on [6] and [7].

Definition 1 (Mesh)**.** A *mesh* is a collection of geometric objects arranged in such manner, that they cover a certain subset of $\mathbb{R}^n$.

The aforementioned objects are generally referred to as *mesh entities*.

If for two mesh entities E_1, E_2, it holds that $E_2 \subset E_1$, we refer to E_2 as to a *subentity* of E_1. Similarly, under these conditions E_1 is a *superentity* of E_2.

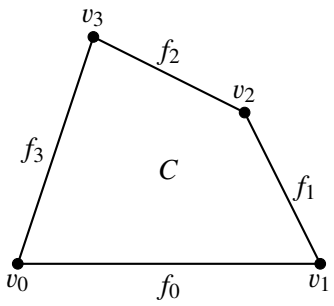


Figure 1.1: Example of mesh entities, C denotes a cell, f_i stands for faces and v_j for vertices.

Dimension of the mesh is a number $\mathbb{N}_0 \ni n = \max\{d \mid d \text{ is the dimension of a mesh entity}\}$.

The $n-$dimensional entities of an $n-$dimensional mesh are referred to as *cells*, the $(n-1)-$dimensional ones are faces and the $0-$dimensional entities are vertices.

Remark. To put the terms sub- and superentity into perspective, let us consider Figure 1.1 one more time. All the other entities are subentities of C. C is a superentity of all the other entities. The vertices v_1 and v_2 are subentities of the face f_1. f_2 is a superentity of v_2 and v_3 etc. This concept is refered to as *incidence* of mesh entities.

Definition 2 (Mesh entities' incidence)**.** Mesh entities E_1, E_2 are incident if

- E_1 is a subentity of E_2, or
- E_1 is a superentity of E_2.

One way of systemizing meshes is discerning structured and unstructured ones.

- Structured meshes are such, whose entities can be numbered in a way, that each entity's location is exactly given by this number.
- Unstructured meshes do not have the structured property described above. Vertices of unstructured meshes are subentities of a variable number of superentities.

Another possible systemization is by the shape of the cells. Let us consider a 2D mesh, which can be triangular, quadrangular or polygonal. The first two options are self-explanatory, a polygonal mesh is such, whose cells are general polygons in 2D (some of them may be triangles or quadrangles). For examples, see Figure 1.2.

This Bachelor project is dealing with unstructured triangular meshes.

1.2 Generalized Stokes' Theorem

Next, let us introduce a well known theorem, which is, however, central to the idea of finite volume method, and allows us to derive an elegant gradient approximation.

Theorem 1 (Generalized Stokes' Theorem, scalar field gradient case)**.** Let $\Omega \subset \mathbb{R}^n$ be an open connected space, $f : \Omega \to \mathbb{R}$ be a scalar field. Then

$$\int_\Omega \nabla f = \int_{\partial\Omega} \mathbf{n} f,$$

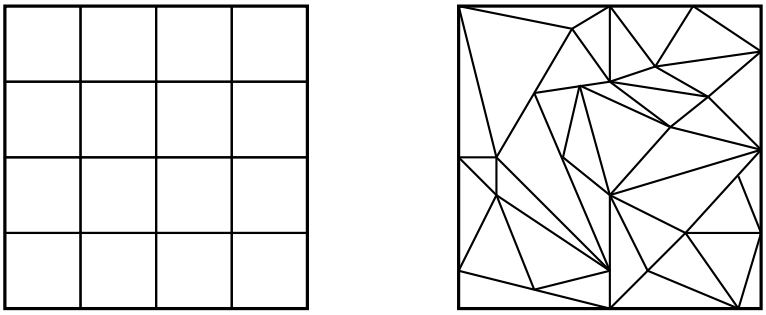


Figure 1.2: A *structured quadrangular* mesh on the left and an *unstructured triangular* mesh on the right hand side. The cells of the triangular meshes have different sizes and rotations and it is not possible to determine their locations by numbering them.

where $\mathbf{n}$ is the unit outward normal vector to $\partial\Omega$.

Remark. The following formulation suits the application in this Bachelor project better. Let V be an n-dimensional volume and $S \equiv \partial V$ be the $(n-1)$-dimensional surface of V such that $S \equiv \bigcup_{j=1}^{m} \sigma^j$, then

$$\int_V \nabla f \mathrm{d}V = \int_S \mathbf{n} f \mathrm{d}S = \sum_{\sigma \subset S} \int_\sigma \mathbf{n} f \mathrm{d}S = \sum_{j=1}^{m} \int_{\sigma_j} \mathbf{n} f \mathrm{d}S.$$

1.3 Approximation of ∇f

Let us approximate both sides of the equation in Theorem 1. The process relies on the following theorem.

Theorem 2 (First integral mean value theorem)**.** Let $f : [a, b] \to \mathbb{R}$ be a continuous function. Then there exists $c \in (a, b)$ such that

$$\int_a^b f(x)\mathrm{d}x = f(c) \cdot (b - a) \tag{1.2}$$

Remark. The Theorem 2 can be generalized to $\mathbb{R}^n$. Instead of the interval $[a, b]$, consider an open, convex subset $\Gamma \subset \mathbb{R}^n$. Then, there exists $c \in \Gamma$ such that $\int_\Gamma f = f(c) \cdot m^{(n)}(\Gamma)$, where $m^{(n)}$ denotes the $n-$dimensional Lebsegue measure.

Remark. Theorem 2 assures that one can approximate an integral of a continuous function by that function's value somewhere in the integration domain. In practice, this point $\mathbf{x}^*$ is not computed exactly, but rather chosen arbitrarily. The choice is not necessarily accurate and this is what creates an approximation error in this process.

It is, however, possible to decrease this error by choosing a denser mesh with smaller cells. Given that our arbitrary choice $\mathbf{x^c}$ lies within the same cell as $\mathbf{x}^*$, these two are getting closer as the cell gets smaller.

For the left hand side, we obtain

$$\int_V \nabla f \mathrm{d}V \approx \nabla f(\mathbf{x}^*) m^{(n)}(V),$$

where $\mathbf{x}^* \in V$ is an n-dimensional analogue of the point $c \in (a, b)$ from Theorem 2 and $m^{(n)}$ denotes the n-dimensional Lebesgue measure (for mesh applications, this is mostly volume or surface).

For the right hand side, the approximation goes like

$$\int_{\partial V} \mathbf{n} f \mathrm{d}S \approx \sum_{\sigma \subset S} m^{(n-1)}(\sigma) f(\mathbf{x}^\sigma) \mathbf{n}^\sigma \tag{1.3}$$

where $m^{(n-1)}$ denotes $n-1$-dimensional Lebesgue measure and $\mathbf{x}^\sigma$ is an arbitrarily chosen point in the surface σ, again based on Theorem 2.

Since measures of mesh entities are easy to obtain via TNL's template function `getEntityMeasure` and normals $\mathbf{n}^\sigma$ and function values $f(\mathbf{x}^\sigma)$ can be computed using plain C++ (or TNL) arithmetic functions, the only thing we do not know is $\nabla f(\mathbf{x}^*)$, which can thus be expressed simply by comparing the equations 1.2 and 1.3, resulting in

$$\nabla f(\mathbf{x}^*) \approx \frac{1}{m^{(n)}} \sum_{\sigma \in S} m^{(n-1)} f(\mathbf{x}^\sigma) \mathbf{n} \tag{1.4}$$

where $\mathbf{n}$ is the unit outward normal vector to the face σ. An implementation of this approximation for a triangular mesh is listed in section 4.2.

1.4 A closer look at the Optimization problem

Now that we have derived $\nabla_h f$, an approximation of the gradient ∇f, let us explore ways to optimize the mesh for it. For this reason, we will now reformulate the function L from L=L($\mathbf{y}$), where $\mathbf{y}$ represents a mesh to L=L($\mathbf{x^1}, \ldots, \mathbf{x^N}$), where $\mathbf{x^i}$ is a vertex of the mesh $\mathbf{y}$.

Let $\|\cdot\| \equiv \|\cdot\|_2$, N be the number of vertices of a 2-dimensional triangular mesh. As per [6], each cell C_i is bordered by faces $\sigma^{i0}, \sigma^{i1}, \sigma^{i2}$ with corresponding vectors $\boldsymbol{\sigma}^{\mathbf{i0}} = \mathbf{x^{i2}} - \mathbf{x^{i1}}$, $\boldsymbol{\sigma}^{\mathbf{i1}} = \mathbf{x^{i0}} - \mathbf{x^{i2}}$ and $\boldsymbol{\sigma}^{\mathbf{i2}} = \mathbf{x^{i1}} - \mathbf{x^{i0}}$ respectively, i.e. face vectors point counter clockwise and the face σ^{ij} is opposite to the point $\mathbf{x^{ij}}$ in each respective cell, as shown in Figure 1.3 below.

Let f be a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, superscript denotes index of a vector, subscript denotes a vector component.

In the following text, we minimize the loss function

$$\mathrm{L}(\mathbf{x^1}, ..., \mathbf{x^N}) = \sum_{i=1}^{\mathrm{N}} \|\nabla_h f(\mathbf{x^i}) - \nabla f(\mathbf{x^i})\|^2,$$

where

$$\nabla_h f\left(\mathbf{x^i}\right) = \frac{1}{m^{(2)}(C_i)} \sum_{\sigma} m^{(1)}(\sigma) f(\mathbf{x}^\sigma) \mathbf{n}^\sigma$$

is the gradient approximation described in section 1.3.

Specifically in case of a triangular mesh

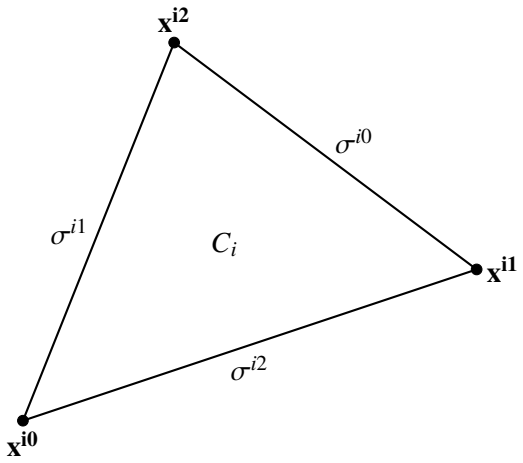


Figure 1.3: A single triangular cell. The numbering of the entities is in line with TNL's implementation, according to [6].

$$\nabla_h f\left(\mathbf{x^i}\right) = \frac{1}{m^{(2)}(C_i)} \sum_{j=0}^{2} m^{(1)}(\sigma^j) f(\mathbf{x}^{\sigma^\mathbf{j}}) \mathbf{n}^{\sigma^\mathbf{j}}.$$

At times, the notation $f(\mathbf{x^i}) \equiv f^\mathbf{i}$ will be used for the sake of clarity and fitting the text in a page.

To perform the optimization, we calculate

$$\nabla \mathrm{L}(\mathbf{x^1}, ..., \mathbf{x^N}),$$

which is a vector of 2N components, i.e. dimension of the problem times number of vectors in the mesh we want to optimize, or 2 components for each $\mathbf{x^i}$, $\mathbf{i} \in \{1, ..., \mathrm{N}\}$.

This requires us to obtain partial derivatives of all the following functions[1]:

- $m^{(2)} : \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}_0^+$,
- $m^{(1)} : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}_0^+$,
- $\mathbf{n} : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$,
- $\mathbf{x}^\sigma : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$
- $f : \mathbb{R}^2 \to \mathbb{R}$.

The derivatives are to be understood in a sense that ∂_{k_1} is the partial derivative by the first component of the k-th vertex of the mesh, as shown in Figure 1.4. This way, a vector $\begin{pmatrix} \partial_{k_1} \mathbf{x^k} \\ \partial_{k_2} \mathbf{x^k} \end{pmatrix}$ represents a translation of a single vertex in direction of $\nabla \mathrm{L}$. Calculation of these derivatives is conducted in the Appendix Appendix A.

[1] Note that we are using the very same formulae which are implemented in TNL to compute these values. There are of course multiple ways to obtain e.g. measure of a 2D triangle. For more insight into implementation, see [5].

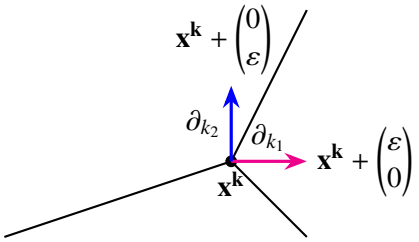


Figure 1.4: A scheme of ∂_{k_1} and ∂_{k_2}.

$m^{(2)}(C_i)$ denotes the 2D Lebesgue measure of a cell. In the triangular case $m^{(2)}(C_i) = \frac{1}{2}|\det \mathbb{A}_i|$, where $\mathbb{A}_i$ is a 2×2 matrix of any two vectors representing the triangle's faces.

$m^{(1)}(\sigma)$ is the length of a triangle's face σ. An intuitive approach is to use the L^2 norm, meaning $m^{(1)}(\sigma^{ij}) = \|\mathbf{x^{i(j+2)\%3}} - \mathbf{x^{i(j+1)\%3}}\| = \left((\mathbf{x^{i(j+2)\%3}}_1 - \mathbf{x^{i(j+1)\%3}}_1)^2 + (\mathbf{x^{i(j+2)\%3}}_2 - \mathbf{x^{i(j+1)\%3}}_2)^2\right)^{\frac{1}{2}}$.

$\mathbf{n}$ denotes a unit outward normal to a face $\boldsymbol{\sigma^{ij}} = \mathbf{x^{i(j+2)\%3}} - \mathbf{x^{i(j+1)\%3}}$. This means that $\mathbf{n}(\boldsymbol{\sigma^{ij}}) = \frac{1}{\|\sigma^{ij}\|} \begin{pmatrix} \boldsymbol{\sigma}^{\mathbf{ij}}_2 \\ -\boldsymbol{\sigma}^{\mathbf{ij}}_1 \end{pmatrix}$. In deed, the dot product $\mathbf{n} \cdot \boldsymbol{\sigma^{ij}} = \boldsymbol{\sigma}^{\mathbf{ij}}_1 \boldsymbol{\sigma}^{\mathbf{ij}}_2 - \boldsymbol{\sigma}^{\mathbf{ij}}_1 \boldsymbol{\sigma}^{\mathbf{ij}}_2 = 0$.

$\mathbf{x}^{\boldsymbol{\sigma^{ij}}}$ is the center of σ^{ij}, that is $\mathbf{x}^{\boldsymbol{\sigma^{ij}}} = \frac{1}{2}(\mathbf{x^{i(j+1)\%3}} + \mathbf{x^{i(j+2)\%3}})$.

Chapter 2

Gradient descent solution

2.1 Gradient descent in general

A robust way to solve the optimization problem 1.1 is to employ a gradient descent method. In general, gradient descent methods can find a function's extrema by exploiting the fact that a gradient vector ∇f always points in the direction of the fastest growth of f. They generate a sequence $\{\mathbf{y^n}\}_{n=1}^{\infty}$ which converges to the function's extreme $\mathbf{y}^*$. To find a minimum, we choose an arbitrary $\mathbf{y^1}$, evaluate $\nabla f(\mathbf{y^1})$ and generate $\mathbf{y^2}$ like $\mathbf{y^2} \equiv \mathbf{y^1} - \nabla f(\mathbf{y^1})$.

An improvement of the basic idea is to introduce a *relaxation parameter*, let us denote it λ, whose function is to control the speed of the convergence of the algorithm. A good choice of λ can result in significantly faster convergence, one such case is illustrated in Figure 2.1. The black ellipses demonstrate the function's contour lines and the minimum is in $\mathbf{y}^*$. The red trajectory demonstrates the process of gradient descent with a poor choice of λ, while the blue trajectory converges faster due to a well-chosen λ, which restricts the step size and ensures faster convergence.

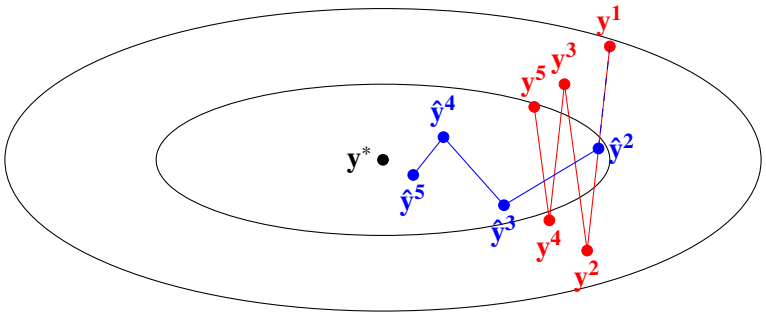


Figure 2.1: Two possible sequences of gradient descent steps with different choices of the λ parameter. Some choices result in faster convergence than others.

All in all, a gradient descent method which converges to a minimum of the function f generates a convergent sequence $\{\mathbf{y^n}\}_{n=1}^{\infty}$ which follows the recurrent formula below.

$$\mathbf{y^{n+1}} \equiv \mathbf{y^n} - \lambda \nabla f(\mathbf{y^n}) \tag{2.1}$$

In a computer implementation, we cannot compute an infinite amount of steps. Instead, we set a *stop condition* and the computation stops when this condition is satisfied. Let us assume the stop condition was satisfied in step k. We then consider $\mathbf{y^k}$ to be the minimum. The exact choice of stop condition determines how accurate this consideration is.

2.2 Gradient descent for our problem

Our aim is now to perform this process on L. For that, we need its gradient, ∇L. ∇L can be computed using three different approaches:

- analytical differentiation of the numerical scheme introduced in section 3.1, the results are summarized in subsection 3.1.1 and the calculations which lead to those results are described in Appendix A in detail,
- approximating derivatives of L based on the so called finite differences, further explained in section 3.2,
- utilizing the library autodiff [8] to obtain the partial derivatives of L using automatic differentiation as described in 3.3.

Since ∇L is a 2N-dimensional vector, for $i \in \hat{\mathrm{N}}$, the vector $\begin{pmatrix} \nabla \mathrm{L}_{2i} \\ \nabla \mathrm{L}_{2i+1} \end{pmatrix}$ gives the direction of the fastest growth of L, given only the position of the vertex $\mathbf{x^i}$ is altered. This is displayed in Figure 2.2. The opposite vector thus denotes the direction in which a gradient descent method moves individual vertices in order to minimize L.

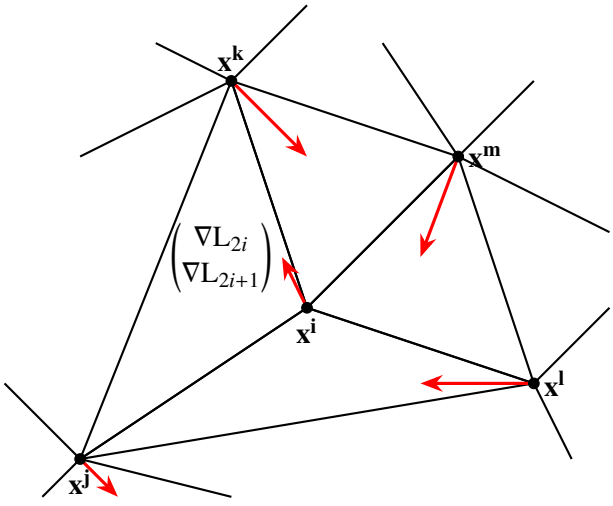


Figure 2.2: Components of $\nabla \mathrm{L} \in \mathbb{R}^{2\mathrm{N}}$ displayed as N vectors $\in \mathbb{R}^2$.

The following flowchart, Figure 2.3, represents a gradient descent method's workflow.

Figure 2.3 is a high level overview of what each of the three presented approaches does. The main difference among them is how they achieve the step "Compute $\nabla \mathrm{L}_{ini}$". This is dealt with in chapter 3.

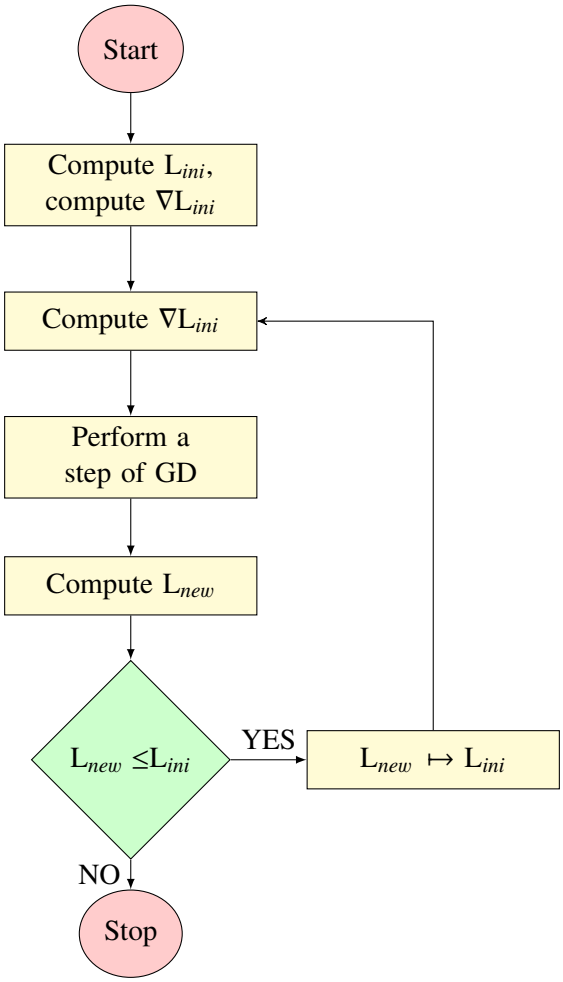


Figure 2.3: Gradient descent method workflow.

Chapter 3

Computation of ∇L

In this chapter, we examine different methods of evaluating ∇L.

3.1 Analytical approach

Perhaps the most intuitive approach to obtain a gradient is to manually calculate formulae for derivatives of the loss function L by all its variables, evaluate them in certain points and arrange the results in a vector.

It is apparent that this approach is very inefficient because for each individual numerical scheme and any choice of the loss function, the derivatives need to be calculated again, which is a tedious and error-prone process. The calculations need to be redone whenever one wants to optimize meshes with a different topology or switch from 2D to 3D.

Having concluded all the calculations, one also has to implement them correctly, which is, again, an error-prone process. The other two approaches to mesh optimization described in Sections 3.2 and 3.3 effectively eliminate these difficulties.

The desired partial derivatives are now presented as formulae. Their detailed calculation can be found in Appendix A.

3.1.1 Formulae

3.1.1.1 $\partial_{\mathbf{k1}}$:

$$\partial_{k_1}\mathbf{x}^{\boldsymbol{\sigma}^{\mathbf{ij}}} = \begin{cases} \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix} & \text{if } k_1 = i((j+1)\%3) \text{ or } k = i((j+2)\%3) \\ \mathbf{0} & \text{else} \end{cases},$$

$$\partial_{k_1} m^{(1)}(\boldsymbol{\sigma}^{\mathbf{ij}}) = \begin{cases} \frac{1}{\|\boldsymbol{\sigma}^{\mathbf{ij}}\|}\boldsymbol{\sigma}^{\mathbf{ij}}{}_1 & \text{if } k_1 = i((j+2)\%3) \\ -\frac{1}{\|\boldsymbol{\sigma}^{\mathbf{ij}}\|}\boldsymbol{\sigma}^{\mathbf{ij}}{}_1 & \text{if } k_1 = i((j+1)\%3) \\ \mathbf{0} & \text{else} \end{cases},$$

$$\partial_{k_1}\mathbf{n^{ij}} = \begin{cases} -\frac{1}{2\|\boldsymbol{\sigma}^{\mathbf{ij}}\|^3} 2\boldsymbol{\sigma}^{\mathbf{ij}}{}_1 \begin{pmatrix} \boldsymbol{\sigma}^{\mathbf{ij}}{}_2 \\ -\boldsymbol{\sigma}^{\mathbf{ij}}_{\mathbf{1}} \end{pmatrix} + \frac{1}{\|\boldsymbol{\sigma}^{\mathbf{ij}}\|} \begin{pmatrix} 0 \\ -1 \end{pmatrix} & \text{if } k_1 = i((j+2)\%3) \\ -\frac{1}{2\|\boldsymbol{\sigma}^{\mathbf{ij}}\|^3} 2\boldsymbol{\sigma}^{\mathbf{ij}}{}_1 \cdot (-1) \cdot \begin{pmatrix} \boldsymbol{\sigma}^{\mathbf{ij}}{}_2 \\ -\boldsymbol{\sigma}^{\mathbf{ij}}_{\mathbf{1}} \end{pmatrix} + \frac{1}{\|\boldsymbol{\sigma}^{\mathbf{ij}}\|} \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \text{if } k_1 = i((j+1)\%3) \\ \mathbf{0} & \text{else} \end{cases},$$

$$\partial_{k_1} m^{(2)}(C_i) = \begin{cases} \frac{1}{2}\mathrm{S}(-\mathbf{x^{i1}}{}_2 + \mathbf{x^{i2}}{}_2) & \text{if } k_1 = i0 \\ \frac{1}{2}\mathrm{S}(-\mathbf{x^{i2}}{}_2 + \mathbf{x^{i0}}{}_2) & \text{if } k_1 = i1 \\ \frac{1}{2}\mathrm{S}(-\mathbf{x^{i0}}{}_2 + \mathbf{x^{i1}}{}_2) & \text{if } k_1 = i2 \\ 0 & \text{else} \end{cases},$$

$$\partial_{k_1} (m^{(2)}(C_i))^{-1} = \begin{cases} \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(-\mathbf{x^{i1}}{}_2 + \mathbf{x^{i2}}{}_2) & \text{if } k_1 = i0 \\ \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(-\mathbf{x^{i2}}{}_2 + \mathbf{x^{i0}}{}_2) & \text{if } k_1 = i1 \\ \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(-\mathbf{x^{i0}}{}_2 + \mathbf{x^{i1}}{}_2) & \text{if } k_1 = i2 \\ 0 & \text{else} \end{cases},$$

3.1.1.2 $\partial_{\mathbf{k2}}$:

$$\partial_{k_2}\mathbf{x}^{\boldsymbol{\sigma}^{ij}} = \begin{cases} \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} & \text{if } k_2 = i((j+2)\%3) \text{ or } k_2 = i((j+1)\%3) \\ \mathbf{0} & \text{else} \end{cases},$$

$$\partial_{k_2} m^{(1)}(\boldsymbol{\sigma}^{\mathbf{ij}}) = \begin{cases} \frac{2}{\|\boldsymbol{\sigma}^{\mathbf{ij}}\|}\boldsymbol{\sigma}^{\mathbf{ij}}{}_2 & \text{if } k_2 = i((j+2)\%3) \\ -\frac{2}{\|\boldsymbol{\sigma}^{\mathbf{ij}}\|}\boldsymbol{\sigma}^{\mathbf{ij}}{}_2 & \text{if } k_2 = i((j+1)\%3) \\ \mathbf{0} & \text{else} \end{cases},$$

$$\partial_{k_2}\mathbf{n^{ij}} = \begin{cases} -\frac{1}{2\|\boldsymbol{\sigma}^{\mathbf{ij}}\|^3} 2\boldsymbol{\sigma}^{\mathbf{ij}}{}_2 \begin{pmatrix} \boldsymbol{\sigma}^{\mathbf{ij}}{}_2 \\ -\boldsymbol{\sigma}^{\mathbf{ij}}_{\mathbf{1}} \end{pmatrix} + \frac{1}{\|\boldsymbol{\sigma}^{\mathbf{ij}}\|} \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \text{if } k_2 = i((j+2)\%3) \\ -\frac{1}{2\|\boldsymbol{\sigma}^{\mathbf{ij}}\|^3} 2\boldsymbol{\sigma}^{\mathbf{ij}}{}_1 \cdot (-1) \cdot \begin{pmatrix} \boldsymbol{\sigma}^{\mathbf{ij}}{}_2 \\ -\boldsymbol{\sigma}^{\mathbf{ij}}_{\mathbf{1}} \end{pmatrix} + \frac{1}{\|\boldsymbol{\sigma}^{\mathbf{ij}}\|} \begin{pmatrix} -1 \\ 0 \end{pmatrix} & \text{if } k_2 = i((j+1)\%3) \\ \mathbf{0} & \text{else} \end{cases},$$

$$\partial_{k_2} m^{(2)}(C_i) = \begin{cases} \frac{1}{2}\mathrm{S}(-\mathbf{x^{i2}}{}_1 + \mathbf{x^{i1}}{}_1) & \text{if } k_2 = i0 \\ \frac{1}{2}\mathrm{S}(\mathbf{x^{i2}}{}_1 - \mathbf{x^{i0}}{}_2) & \text{if } k_2 = i1 \\ \frac{1}{2}\mathrm{S}(-\mathbf{x^{i1}}{}_1 + \mathbf{x^{i0}}{}_1) & \text{if } k_2 = i2 \\ 0 & \text{else} \end{cases},$$

$$\partial_{k_2}(m^{(2)}(C_i))^{-1} = \begin{cases} \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(-\mathbf{x^{i2}}_1 + \mathbf{x^{i1}}_1) & \text{if } k_2 = i0 \\ \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(\mathbf{x^{i2}}_1 - \mathbf{x^{i0}}_2) & \text{if } k_2 = i1 \\ \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(-\mathbf{x^{i1}}_1 + \mathbf{x^{i0}}_1) & \text{if } k_2 = i2 \\ 0 & \text{else} \end{cases}.$$

3.1.1.3 Derivatives of $\mathbf{L(x^1, ..., x^N)}$

If we start differentiating L, we obtain

$$\partial_{k_1} L(\mathbf{x^1}, ..., \mathbf{x^N}) = 2\sum_{i=1}^{\mathrm{N}} \left(\partial_{k_1}(\nabla_h f^{\mathbf{i}})_1 \cdot [\nabla_h f^{\mathbf{i}} - \nabla f^{\mathbf{i}}]_1 + \partial_{k_1}(\nabla_h f^{\mathbf{i}})_2 \cdot [\nabla_h f^{\mathbf{i}} - \nabla f^{\mathbf{i}}]_2 \right).$$

Since $\nabla_h f^{\mathbf{i}}$ is a product of four functions, it has to be differentiated accordingly. For f, g, h, i functions of x, the following holds:

$$\partial_x(fghi) = (\partial_x f)(ghi) + (\partial_x g)(fhi) + (\partial_x h)(fgi) + (\partial_x i)(fgh).$$

That implies

$$\partial_{k_1}\nabla_h f^i = \partial_{k_1}\left(\frac{1}{m^{(2)}(C_i)}\right)\sum_{j=0}^{2} m^{(1)}(\sigma^j) f^{\sigma^j} n^{\sigma^j} + \frac{1}{m^{(2)}(C_i)}\sum_{j=0}^{2} \partial_{k_1}(m^{(1)}(\sigma^j)) f^{\sigma^j} n^{\sigma^j}$$
$$+ \frac{1}{m^{(2)}(C_i)}\sum_{j=0}^{2} m^{(1)}(\sigma^j)\partial_{k_1}(f^{\sigma^j}) n^{\sigma^j} + \frac{1}{m^{(2)}(C_i)}\sum_{j=0}^{2} m^{(1)}(\sigma^j) f^{\sigma^j}\partial_{k_1}(n^{\sigma^j}).$$

Since $f^{\sigma^j} \equiv f(\mathbf{x^{\sigma^{ij}}})$ is a compound function, the definitive formula which will later be used for implementation is

$$\partial_{k_1}\nabla_h f^i = \partial_{k_1}\left(\frac{1}{m^{(2)}(C_i)}\right)\sum_{j=0}^{2} m^{(1)}(\sigma^j) f^{\sigma^j} n^{\sigma^j} + \frac{1}{m^{(2)}(C_i)}\sum_{j=0}^{2} \partial_{k_1}(m^{(1)}(\sigma^j)) f^{\sigma^j} n^{\sigma^j}$$
$$+ \frac{1}{m^{(2)}(C_i)}\sum_{j=0}^{2} m^{(1)}(\sigma^j) \cdot \nabla(f(\mathbf{x^{\sigma^j}})) \cdot \partial_{k_1}\mathbf{x^{\sigma^j}} \cdot n^{\sigma^j} + \frac{1}{m^{(2)}(C_i)}\sum_{j=0}^{2} m^{(1)}(\sigma^j) f^{\sigma^j}\partial_{k_1}(n^{\sigma^j}).$$

Each multiplication is to be interpreted as multiplication of matrices, where scalars are $s \in \mathbb{R}^{1,1}$, $\nabla f \in \mathbb{R}^{1,2}$ and all the other vectors $v \in \mathbb{R}^{2,1}$. The immediate result of this is that $\partial_{k_1}\nabla_h f^i$ is a well defined scalar, because $\nabla f \cdot v \in \mathbb{R}^{1,1}$. Another possible interpretation would be that $\nabla f \cdot v$ is a dot product of two vectors from $\mathbb{R}^2$.

We can calculate the other partial derivative in a similar fashion,

$$\partial_{k_2}\nabla_h f^i = \partial_{k_2}\left(\frac{1}{m^{(2)}(C_i)}\right)\sum_{j=0}^{2} m^{(1)}(\sigma^j) f^{\sigma^j} n^{\sigma^j} + \frac{1}{m^{(2)}(C_i)}\sum_{j=0}^{2} \partial_{k_2}(m^{(1)}(\sigma^j)) f^{\sigma^j} n^{\sigma^j}$$

$$+\frac{1}{m^{(2)}(C_i)}\sum_{j=0}^{2} m^{(1)}(\sigma^j)\cdot\nabla(f(\mathbf{x}^{\sigma^j}))\cdot\partial_{k_2}\mathbf{x}^{\sigma^j}\cdot n^{\sigma^j}+\frac{1}{m^{(2)}(C_i)}\sum_{j=0}^{2} m^{(1)}(\sigma^j)f^{\sigma^j}\partial_{k_2}(n^{\sigma^j}).$$

3.1.2 Verification of the calculations

Since some of the partial derivatives in subsection 3.1.1 are fairly complicated, we decided to check the handcalculated results with results of numerical computations done by the python package MyGrad [9].

The procedure in general consisted of defining the derivatives' formulae, plugging random values in them and comparing them with numerical differentiation done by MyGrad. For clarity, we present a snippet of the code, namely a part used to check if the derivatives of $m^{(1)}$ were calculated correctly. The rest of the code is at disposal at [13] in the file `autodiff.ipynb`.

```
# importing packages
import numpy as np
import mygrad as mg
import random as rn

# defining necessary functions
# Kronecker delta
def delta( first, second ):
    if( first is second ): return 1
    else: return 0

# m^(1)
def m(x00, x01, x10, x11):
    return mg.sqrt((x00-x10)**2+(x01-x11)**2)

# 1st partial derivative
def dm1(x00, x01, x10, x11, j):
    return 1/m(x00, x01, x10, x11) * (x10-x00) * (delta(x10, j) - delta(x00, j))
# 2nd partial derivative
def dm2(x00, x01, x10, x11, j):
    return 1/m(x00, x01, x10, x11) * (x11-x01) * (delta(x11, j) - delta(x01, j))

# checking if the analytical and numerical derivatives
# vary by more than 10e-6, which should imply equality
count = 0
for i in range(1000):
    x00 = mg.tensor(rn.random()*100)
    x01 = mg.tensor(rn.random()*100)
    x10 = mg.tensor(rn.random()*100)
    x11 = mg.tensor(rn.random()*100)
    ms = m(x00, x01, x10, x11)
    ms.backward()
    num = x00.grad
    difference = mg.abs(num - mg.tensor( dm1( x00, x01, x10, x11, x00 ) ) )
    if( difference > 0.000001 ):
        print( difference )
        print( 'for values x00:' + str(x00) + ', x01: ' + str(x01) + ', x10: ' +
            str(x10) + ', x11: ' + str(x11) )
```

```
        count += 1
print( count )
```

If `count == 0` after a thousand random picks of `x00`, `x01`, `x10`, `x11`, we could safely conclude that the calculations described in subsection 3.1.1 were in deed correct.

Interestingly enough, not only was the value of `difference` never above `10e-6`, for which we tested in the first place, but we also never encountered one different from `Tensor(0.)`, which is zero in terms of mygrad's `Tensor` data type.

We took similar approach to check whether all individual derivatives were implemented correctly in C++. For this, we used the library autodiff [8].

3.2 Finite difference approach

This way of gradient approximation, referred to as finite difference, is, among others, discussed in [1], Chapter 10.10, and in [2] in Chapter 8.1.

Let us illustrate the approach on a function $u : \mathbb{R}^2 \to \mathbb{R}$, $u = u(x, y)$. Then, a partial derivative of u can be obtained in the following manner

$$\partial_x u(x, y) = \lim_{\varepsilon \to 0} \frac{u(x + \varepsilon, y) - u(x, y)}{\varepsilon}.$$

The finite difference mimics the limit by computing a similar fraction

$$\partial_x u(x, y) = \delta_x u(x, y) + O(\varepsilon) \equiv \frac{u(x + \varepsilon, y) - u(x, y)}{\varepsilon} + O(\varepsilon),$$

where e.g. $\varepsilon \equiv 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}, \ldots$

3.3 Automatic differentiation approach

This section is heavily based on the information presented in [2], chapter 8.2.

There are multiple ways to perform automatic differentiation. In this appendix, we will briefly describe them and outline their advantages and drawbacks.

In general, this technique exploits the fact, that all elementary functions can be evaluated using a tree of simple binary or unary operations. The binary operations are

- addition,
- subtraction,
- multiplication,
- division,
- power

and the unary operations include e.g.

- trigonometric functions,

- exponential, logarithm,
- ...

A compound function can then be evaluated using a *forward sweep* through a tree of the operations above (which is specific for each individual function), a procedure shown in Figure 3.1. The vector (a, b) where we evaluate the function is called a *seed vector*.

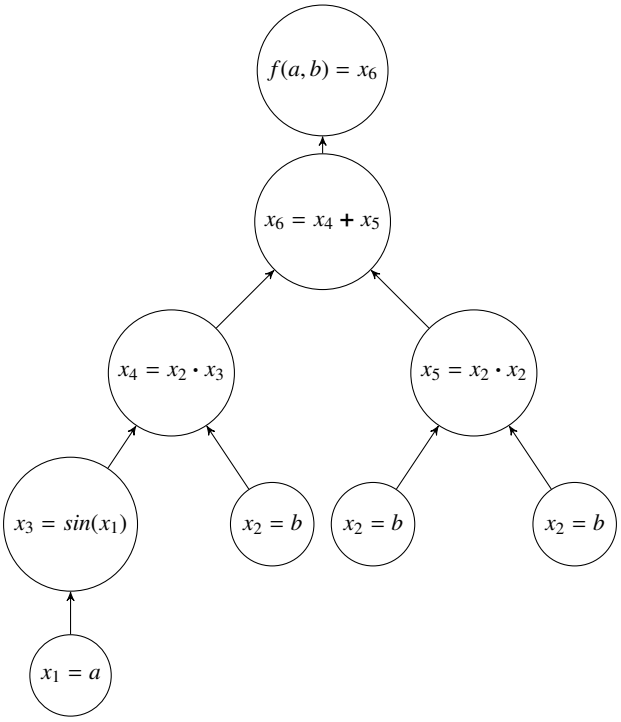


Figure 3.1: Evaluation of a function $f(x_1, x_2) = x_2 \sin(x_1) + x_2^2$ at the point (a, b). This procedure is refered to as a forward sweep. The vector (a, b) is called a seed vector.

This idea can be taken one step further by evaluating a certain derivative at each node as well.

Another important ingredient is, of course, the *chain rule*, which states that for a function $u(v(x))$, where $v(x) \in \mathbb{R}^n$, $u(v(x)) \in \mathbb{R}^m$, the following holds:

$$\frac{\partial (u \circ v)_j}{\partial x_i}(a) = \sum_{k=1}^{m} \frac{\partial u_j}{\partial v_k}\big(v(a)\big) \frac{\partial v_k}{\partial x_j}(a).$$

Finally, we do know how to differentiate each of the unary and binary operations presented above.

3.3.1 Forward mode

Knowing the facts above allows us to efficiently compute partial derivatives with the forward autodiff technique. Let us demonstrate on an example, where we aim to obtain $\frac{\partial f}{\partial x_1}(a, b)$. In a forward sweep, we can evaluate $\frac{\partial}{\partial x_1}$ at every seed node using well known rules for ordinary derivatives of elementary functions and operations allowed in the evaluation true. These rules include:

- $(f+g)' = f' + g'$,
- $(f \cdot g)' = f'g + g'f$,
- $(x^k)' = kx^{k-1}$,
- $(sin(x))' = cos(x)$ and other differential trigonometric identities,
- $(e^x)' = e^x$.

In a forward sweep, these derivatives propagate further into the tree using the chain rule and leading up to (in this case) $\frac{\partial}{\partial x_6}$. For a better visualization, let us show this in Table 3.1 and in a tree in Figure 3.2.

$x_1 = a$	$\dot{x_1} = 1$
$x_2 = b$	$\dot{x_2} = 0$
$x_3 = \sin(a)$	$\dot{x_3} = \cos(a) \cdot \dot{a} = \cos(a) \cdot 1$
$x_4 = x_2 \cdot x_3 = bx_3$	$\dot{x_4} = x_2\dot{x_3} + \dot{x_2}x_3 = b\cos(a) + 0$
$x_5 = x_2 \cdot x_2$	$\dot{x_5} = \dot{x_2}x_2 + x_2\dot{x_2} = 0 + 0$
$x_6 = x_4 + x_5$	$\dot{x_6} = \dot{x_4} + \dot{x_5} = b\cos(a)$

Table 3.1: Evaluating derivatives which lead to $\frac{\partial f}{\partial x_1}(a, b)$ in a forward sweep. $\dot{y} \equiv \frac{\partial y}{\partial x_1}(a)$.

Another way to gain an insight into the forward autodiff process would be to write the derivative like this

$$\begin{aligned}\frac{\partial}{\partial x_1}\Big(x_2\sin(x_1) + x_2^2\Big) &= \frac{\partial}{\partial \sin(x_1)}\Big(x_2\sin(x_1) + x_2^2\Big) \cdot \frac{\partial}{\partial x_1}\Big(\sin(x_1)\Big) \\ &= x_2 \cdot \cos(x_1)\frac{\partial x_1}{\partial x_1} = x_2\cos(x_1),\end{aligned}$$

where the zero derivatives are already omitted.

3.3.2 Reverse mode

In the forward mode, the *denominator* of the derivative $\frac{\partial y}{\partial x_1}$ was fixed and denoted $\dot{y}$. In the reverse mode we denote e.g. $\frac{\partial f}{\partial y} \equiv \overline{y}$, or in other words, we fix the *numerator*. The seed is $\overline{f} = \frac{\partial f}{\partial f} = 1$ in the top node. For the reverse mode to work, a tree of function evalutaion must exist before the computation, so it is necessary to evaluate the function itself by a forward sweep (without evaluating the dotted derivatives).

The evaluation of the tree then goes top to bottom and returns the whole gradient ∇f (which in this case is the vector $\left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right)$) after a single sweep.

The process is demonstrated in Figure 3.3. At the beginning, we set $\overline{x_6} \equiv 1$ and all the other derivatives with a bar to be 0. The top derivatives then propagate to the bottom with the aim to evaluate $\overline{a}$ and $\overline{b}$, the two components of $\nabla f(a, b)$.

3.3.3 Comparision of the modes

For a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, the forward mode only returns one of the n derivatives needed to assemble a gradient vector after each sweep. The reverse mode on the other hand returns the

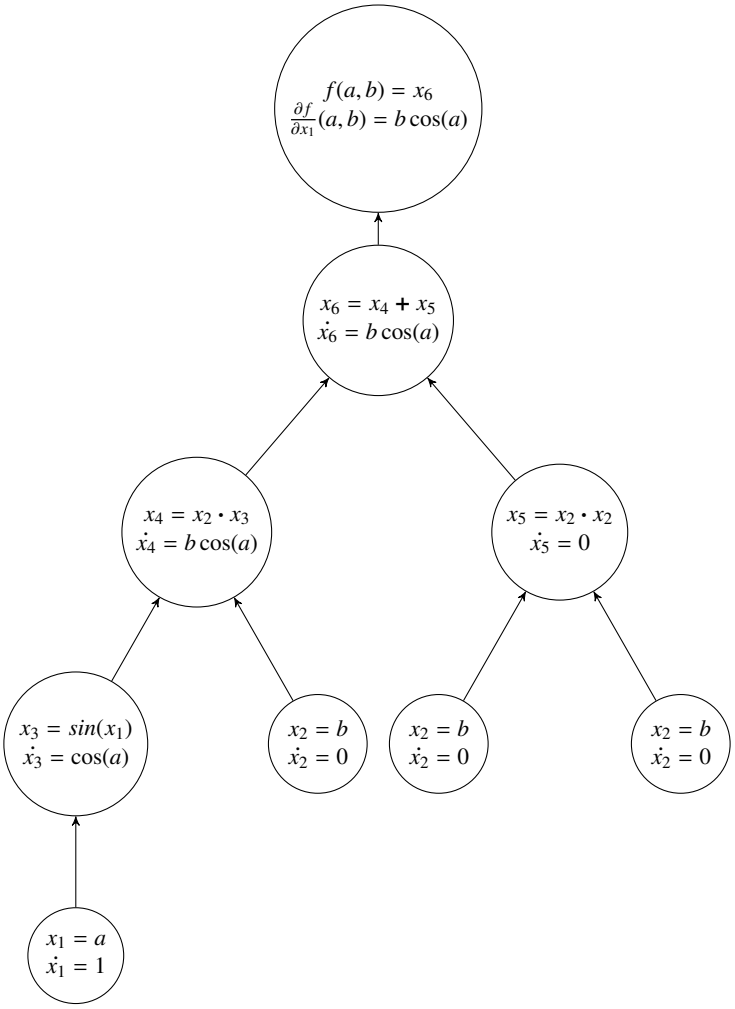


Figure 3.2: Evaluation of $\frac{\partial f}{\partial x_1}(a,b)$ using a forward sweep.

whole gradient after a single reverse sweep. The reverse mode, however, requires that the function evaluation tree is stored in the computer's memory. Depending on the exact value of n, individual functions and implementations of autodiff algorithms, this may or may not be more efficient than executing multiple forward sweeps.

For vector functions (which are by no means in scope of this Bachelor project) $F : \mathbb{R}^n \to \mathbb{R}^m$, the derivative $DF(\mathbf{x})$ is an $m \times n$ matrix. A single forward sweep gives a column of this matrix (note that this works for $m \equiv 1$ as well) while a reverse sweep gives a row. Thus for m significantly larger than n, the forward mode is generally more efficient. This efficiency further depends on the exact implementation.

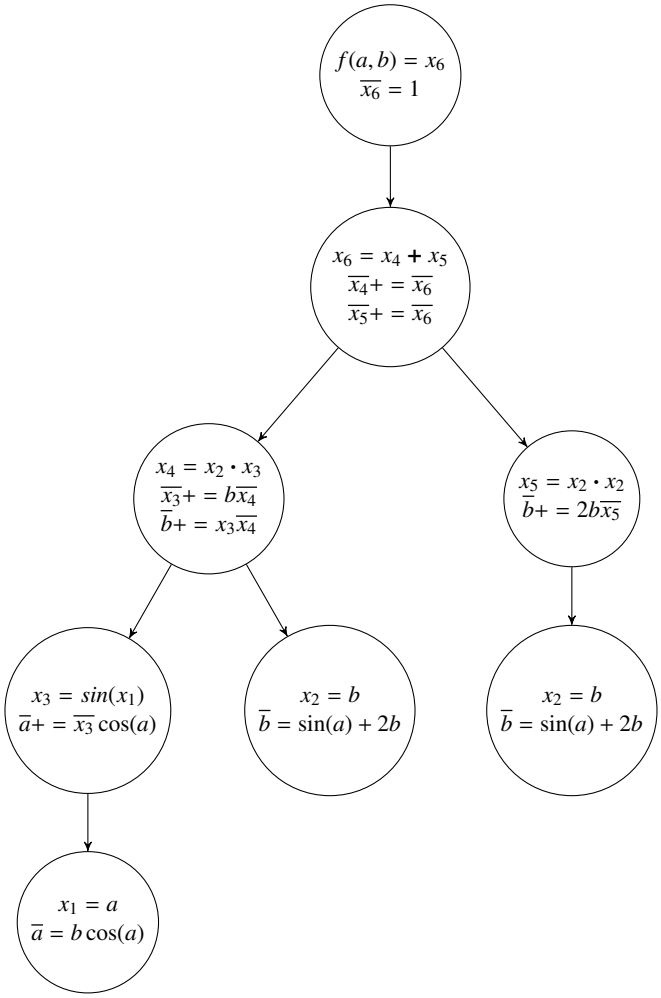


Figure 3.3: Evaluation of $\frac{\partial f}{\partial x_1}(a, b)$ using a reverse sweep.

Chapter 4

Implementation

In this chapter, we present the code in which we implemented an algorithm to optimize meshes for computation of the gradient approximation from section 1.3.

The first section 4.1 is dedicated to describing how meshes are dealt with in the TNL library based on [6] and [7].

In the rest of the chapter, our own code is presented. Since there is a lot of template use, the code contains more whitespaces than usual to allow for faster orientation.

4.1 TNL

4.1.1 Non-mesh data structures

Apart from meshes, which will be described in subsection 4.1.2, TNL implements other data structures which allow for quick and effortless work. In the context of this Bachelor project, they are mostly template classes `TNL::Containers::Array` and `TNL::Containers::Vector`.

`Array` functions similarly to plain C++, as it is a simple container for a certain number of variables of the same type. It serves as a base data structure for TNL meshes and matrices [5]. All infromation about a mesh is stored using `Array` (either plain or as a matrix).

A `Vector` extends `Array` with simple element-wise arithmetics, e.g.

```
TNL::Containers::Vector< double, TNL::Devices::Host > u = { 1.0, 2.0, 3.14 };
std::cout << u << "\n";

TNL::Containers::Vector< double, TNL::Devices::Host > v;
v = 2 * u;
std::cout << v << "\n";
```

The code snippet above prints

```
[ 1, 2, 3.14 ]
[ 2, 4, 6.28 ].
```

4.1.2 Meshes

TNL supports a data structure for unstructured meshes, as well as several vital operations on this structure, which include:

- `getEntitiesCount()` to return the amount of entities of chosen dimension,
- `getSubentityIndex()` to return an index of a subentity with respect to its superentity,
- `getEntityCenter()` to return central point of an entity (cell or face),
- `getEntityMeasure()` to return measure of an entity (cell or face),
- ...

We only mentioned some of the functions which are often used in this Bachelor project. A complete list of mesh functions is best accessible at [5].

If a `mesh` is not constant, it is possible to manipulate its points via `mesh.getPoints()`, which returns a `TNL::Containers::Vector` of all the points, where index of a mesh point matches its index in the `Vector`.

The data structure, whose implementation is described in [7] in detail, contains information about

- coordinates of the mesh's vertices,
- incidence of individual entities,
- neighboring cells,
- boundary tags.

Knowledge of this information is enough to unequivocally represent the stored mesh. The data structure is briefly described in the rest of this section.

4.1.2.1 Vertex coordinates

The vertices are saved in a `TNL::Containers::Array` of $m \cdot n$ elements, where m is the number of vertices and n their dimension. Components of $\mathbf{x}^k$ are stored in elements $k \cdot m, \ldots, k \cdot m + n$, as is illustrated in Figure 4.1.

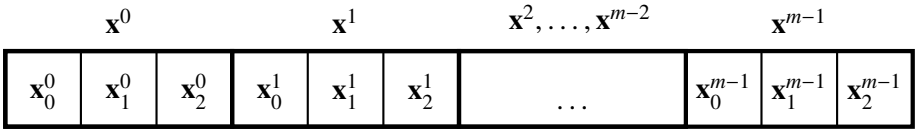


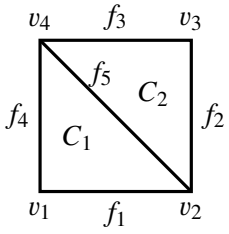
Figure 4.1: Illustration of an array of mesh vertices.

Vertices are the only entities whose exact coordinates need to be saved, since all other entities consist of vertices. The rest of the mesh can be recovered from knowledge of incidence of the remaining entities.

4.1.2.2 Incidence

The information about incidence of entities is stored in a binary *incidence matrix*, let us denote it $\mathbb{I}_{d_1,d_2}$, which has a row for each d_1-dimensional entity (e.g. cell) and a column for each d_2-

dimensional entity (e.g. face). Then $\left(\mathbb{I}_{d_1,d_2}\right)_{j,k} = 1$ if the j-th d_1 dimensional and the k-th d_2 dimensional entity are incident (i.e. if the face f_k is a subentity of the cell C_j) and else, $\left(\mathbb{I}_{d_1,d_2}\right)_{j,k} = 0$. A sample mesh and its incidence matrices are illustrated in Figure 4.2. The illustrated matrices can be transposed to present an inverse relationship between subentities and their superentities, $\mathbb{I}_{d_1,d_2} = \mathbb{I}^T_{d_2,d_1}$ [6].



	C_1	C_2
f_1	1	0
f_2	0	1
f_3	0	1
f_4	1	0
f_5	1	1

	C_1	C_2
v_1	1	0
v_2	1	1
v_3	0	1
v_4	1	1

	f_1	f_2	f_3	f_4	f_5
v_1	1	0	0	1	0
v_2	1	1	0	0	1
v_3	0	1	1	0	0
v_4	0	0	1	1	1

Figure 4.2: A simple mesh and its incidence matrices.

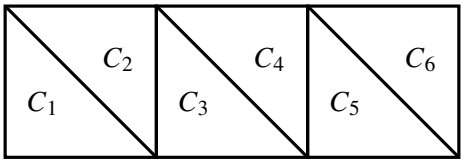
The implementation [7] uses *Ellpack* matrices for incidence of subentities and *Sliced Ellpack* matrices for superentities. These are implemented in `TNL::Matrices` [5].

4.1.2.3 Neighboring cells

A cell C_1 is called a neighbor of cell C_2 (and vice versa) if their intersection $C_1 \cap C_2 \neq \emptyset$ is a mesh entity. As per [7], in TNL, this relationship is represented by an *adjacency matrix* $\mathbb{A}$. $\mathbb{A}$ is a binary square matrix which has a row and a column for each cell of the mesh. It follows $\mathbb{A}_{i,j} = 1$ if the cells C_i and C_j are neighbors and $\mathbb{A}_{i,j} = 0$ else. In Figure 4.3, a simple mesh is presented along with its adjacency matrix.

4.1.2.4 Boundary tag

A cell's boundary tag refers to whether the cell is a border cell or an interior one. The boundary tag of a cell is determined by the faces of that cell. If a face belongs to two or more cells, it is an interior face, else (if it is a subset of only a single cell) the face is a boundary face. If any of the cell's faces is a boundary face, then the cell is a boundary cell [7]. We illustrate this in Figure 4.4, where border faces are red and interior ones are blue. The cell C_1 is written in red (a border cell), because its subentity f_1 is a border face.



$$\mathbb{A}=\begin{pmatrix}1&1&1&0&0&0\\1&1&1&1&0&0\\1&1&1&1&1&0\\0&1&1&1&1&1\\0&0&1&1&1&1\\0&0&0&1&1&1\end{pmatrix}$$

Figure 4.3: An example mesh and its adjacency matrix.

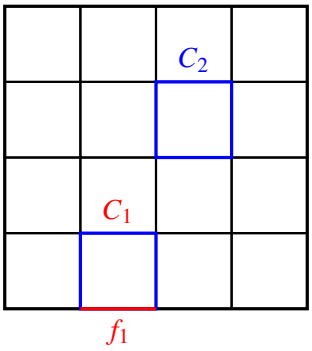


Figure 4.4: The cell C_1 has the face f_1, which is not subentity of any other cell. Therefore, f_1 is a boundary face and C_1 is a boundary cell. The cell C_2, on the other hand, does not have this property, all its faces are also subentities of other cells. That is why C_2 is an interior cell.

4.2 Computation of $\nabla_h f$

In the next listing, we present the code we wrote to compute the finite volume approximation of ∇f which was discussed in section 1.3. The approximated gradient $\nabla_h f$ is computed and stored in a `TNL::Containers::Vector`. Only the most essential part of the code is listed, the whole code is accessible at [13] in the file `numgrad.cpp`.

++ttrmdef@ulttxr.*ansittrmdef@ulttxr.*

```
// the function f whose gradient we approximate
template< typename V >
double f( V v )
{
    double x = v[0];
    double y = v[1];

    return x*x*y*y;
}
```

```
// manually computed gradient of the function above to check approximation
    accuracy
template< typename V >
V angrad( V v )
{
    double x = v[0];
    double y = v[1];

    V grad = { 0, 0 };
    grad[0] = 2*x*y*y;
    grad[1] = 2*x*x*y;
    return grad;
}

template< typename MeshConfig >
bool grad( const Mesh< MeshConfig, Devices::Host >& mesh, const std::string&
    fileName )
{
    ... code intentionally left out ...

    for( int i = 0; i < cellsCount; i++ )
    {
        auto cell = mesh.template getEntity< MeshType::getMeshDimension() >( i );
        PointType sum = { 0, 0 };
        for( int j = 0; j < 3; j++ )
        {
            const auto faceIdx = cell.template getSubentityIndex<
                MeshType::getMeshDimension() - 1 >( j );
            const auto sigma = mesh.template getEntity<
                MeshType::getMeshDimension() - 1 >( faceIdx );

            PointType faceVector = mesh.getPoint( cell.template getSubentityIndex<
                0 > ( (j+2)%3 ))
                            - mesh.getPoint( cell.template getSubentityIndex< 0
                                > ( (j+1)%3 ));

            PointType outwardNormal = normalize< PointType >( { faceVector[1],
                -faceVector[0] } );
            PointType x_sigma = getEntityCenter( mesh, sigma );

            double f_sigma = f< PointType >( x_sigma );
            sum += getEntityMeasure( mesh, sigma ) * f_sigma * outwardNormal;
        }

        PointType cellCenter = getEntityCenter( mesh, cell);
        analytical[ i ] = angrad< PointType >( cellCenter );

        PointType grad = ( 1.0 / getEntityMeasure( mesh, cell ) ) * sum;
        grads[ i ] = grad;
    }

    double error = 0;
```

```
    for(int i = 0; i < cellsCount; i++)
    {
        PointType localError = grads[ i ] - analytical[ i ];
        error += l2Norm( localError );
    }

    return true;
}
```

4.3 Computation of ∇L and Gradient Descent

As we have shown in chapter 3, there are three different approaches to evaluating ∇L. In this section, we present implementations of those computations.

4.3.1 Implementation of the Analytical approach

The formulae and their verification were presented in section 3.1. We, however, did not manage to obtain a convergent gradient descent algorithm. Either we made a mistake in the implementation or the formulae result in too big of a numerical error in C++. The reader is invited to see the file `opti2.cpp` at [13], which summarizes what we managed to implement to the best of our knowledge and belief.

4.3.2 Implementation of the Finite difference approach

As was stated in section 3.2, our aim here is to compute a fraction which approximates the partial derivatives with an error of $O(\varepsilon)$, like this:

$$\partial_x u(x,y) = \delta_x u(x,y) + O(\varepsilon) \equiv \frac{u(x+\varepsilon,y) - u(x,y)}{\varepsilon} + O(\varepsilon),$$

where e.g. $\varepsilon \equiv 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}, \ldots$

For a function $v : \mathbb{R}^n \to \mathbb{R}$, $v = v(x_1, \ldots, x_n)$, the approximation can be generalized in the following manner:

$$\frac{\partial v}{\partial \mathbf{x}_i}(\mathbf{x}) = \partial_i v(\mathbf{x}) \approx \delta_i v(\mathbf{x}) \equiv \frac{v(\mathbf{x} + \varepsilon \mathbf{e}_i) - v(\mathbf{x})}{\varepsilon},$$

where $\mathbf{e}_i$ is a unit vector in the direction $\mathbf{x}_i$ and again $\varepsilon = 10^{-6}, \ldots, 10^{-12}$.

C++ does not offer much higher precision by its nature and the accuracy of approximation of a derivative using a finite difference does not necessarily increase with smaller ε, even though values as small as 10^{-16} can be used in C++. This is the result of the fact, that the finite difference represents a slope of a secant line which, as $\varepsilon \to 0$, approaches the tangent line of a function, whose slope is the derivative. With smaller ε, the secant line is closer to the tangent, but computer's rounding errors increase.

The accuracy of such computation is illustrated in Table 4.1. The values are rounded using `std::setprecision(8)` and written in the same format in which C++ returns them.

The code used for computation of the table is accessible at [13] in the file `diff_testground.cpp`.

f	$x^2 - 2x + 3$				
x	1	2	3	10	100
$\frac{\mathrm{d}}{\mathrm{d}x}$	0	2	4	18	198
δ_x^{-4}	9.9999999e-05	2.0001	4.0001	18.0001	198.0001
δ_x^{-6}	1.0000889e-06	2.000001	4.000001	18.000001	198
δ_x^{-8}	0	2	4	18.000001	197.99991
δ_x^{-10}	0	2.0000002	4.0000003	18.000037	197.997
δ_x^{-12}	0	2.0001778	4.0003556	18.005153	196.45086

Table 4.1: Comparison of ∂_x (or d_x, the procedure works the same) and δ_x for various values of ε. δ_x^k stands for δ_x, where $\varepsilon = 10^{-k}$.

We chose $f(x) = x^2 - 2x + 3$, because the functions we aim to differentiate are fractions of polynomials and do not grow significantly more rapidly than this one.

The optimal value which suffers neither from bad approximation of the tangent (e.g. for $\varepsilon > 10^{-4}$), nor too large rounding errors ($\varepsilon < 10^{-12}$) appears to be "somewhere in the middle", in this case $\varepsilon = 10^{-8}$, but this number may vary for other functions. It also matters whether the derivative is small or large. For derivatives ≈ 0, smaller ε works better, for large derivatives, larger ε is the correct pick.

Overall, we found $\varepsilon \in [10^{-6}, 10^{-10}]$ is a good choice which leads to a stable and convergent gradient descent.

Below, we present the code for a single iteration of the optimization algorithm. Once again, we only list certain essential parts of the code and leave the rest accessible at [13] in the file `nablaDef.cpp`. To automate the process and create an optimization algorithm, it is sufficient to close the whole code in a loop and choose a stop condition.

The code utilizes lambda functions and `TNL`'s parallel `for` cycles, functions `forAll` and `forInterior` (there is also `forBoundary`, but that one is not required here), to iterate over all (or all interior) mesh entities of chosen dimension. If the code is run on a CPU (`TNL::Devices::Host`), this functionality makes the work easier for a programmer who does not have to write a `for` cycle manually. If one uses `TNL::Devices::CUDA` instead, these cycles work in parallel and accelerate the computation.

```
// loss function
template< typename t >
double L( Containers::Vector< t >& nabla_h,
        Containers::Vector< t >& nabla )
{
    double L = 0.0;
    for( int i = 0; i < nabla_h.getSize(); i++ )
    {
        L += l2Norm( nabla_h[ i ] - nabla[ i ] ) * l2Norm( nabla_h[ i ] - nabla[
            i ] );
    }
    return L;
}
```

```
const double EPSILON = 1e-8;

// a single iteration of gradient descent
template< typename MeshConfig >
bool nablaDef( Mesh< MeshConfig, Devices::Host >& mesh, const std::string&
    fileName )
{
    ... left out part of the code ...

    auto get_nabla_h = [ &mesh, &nabla_h, &nabla ] ( GlobalIndexType i ) mutable
    {
        auto cell = mesh.template getEntity< MeshType::getMeshDimension() >( i );
        PointType sum = { 0, 0 };
        for( int j = 0; j < 3; j++ )
        {
            const auto faceIdx = cell.template getSubentityIndex<
                MeshType::getMeshDimension() - 1 >( j );
            const auto sigma = mesh.template getEntity<
                MeshType::getMeshDimension() - 1 >( faceIdx );

            PointType faceVector = mesh.getPoint( cell.template getSubentityIndex<
                0 > ( (j+2) % 3 ) )
                            - mesh.getPoint( cell.template getSubentityIndex< 0
                                > ( (j+1) % 3 ) );

            PointType outwardNormal = normalize< PointType >( { faceVector[ 1 ],
                -faceVector[ 0 ] } );

            PointType x_sigma = getEntityCenter( mesh, sigma );

            double f_sigma = f< PointType >( x_sigma );
            sum += getEntityMeasure( mesh, sigma ) * f_sigma * outwardNormal;
        }

        PointType grad_h = ( 1.0 / getEntityMeasure( mesh, cell ) ) * sum;
        PointType grad = angrad< PointType >( getEntityCenter( mesh, cell ) );

        for( int j = 0; j < 3; j++ )
        {
            int globalPointIdx = cell.template getSubentityIndex< 0 >( j );
            nabla[ globalPointIdx ] += grad;
            nabla_h[ globalPointIdx ] += grad_h;
        }
    };
    mesh.template forAll< MeshType::getMeshDimension() >( get_nabla_h );

    // initializing perturbed vectors
    Containers::Vector< PointType > nabla_h_eps ( verticesCount );
    Containers::Vector< PointType > nabla_eps ( verticesCount );
    nabla_h_eps = 0;
    nabla_eps = 0;
```

```
auto get_nabla_h_eps = [ &mesh, &nabla_h_eps, &nabla_eps ] ( GlobalIndexType
    i ) mutable
{
    auto cell = mesh.template getEntity< MeshType::getMeshDimension() >( i );
    PointType sum = { 0, 0 };
    for( int j = 0; j < 3; j++ )
    {
        const auto faceIdx = cell.template getSubentityIndex<
            MeshType::getMeshDimension() - 1 >( j );
        const auto sigma = mesh.template getEntity<
            MeshType::getMeshDimension() - 1 >( faceIdx );

        PointType faceVector = mesh.getPoint( cell.template getSubentityIndex<
            0 > ( (j+2) % 3 ) )
                        - mesh.getPoint( cell.template getSubentityIndex< 0
                            > ( (j+1) % 3 ) );

        PointType outwardNormal = normalize< PointType >( { faceVector[ 1 ],
            -faceVector[ 0 ] } );

        PointType x_sigma = getEntityCenter( mesh, sigma );

        double f_sigma = f< PointType >( x_sigma );
        sum += getEntityMeasure( mesh, sigma ) * f_sigma * outwardNormal;
    }

    PointType grad_h = ( 1.0 / getEntityMeasure( mesh, cell ) ) * sum;
    PointType grad = angrad< PointType >( getEntityCenter( mesh, cell ) );

    for( int j = 0; j < 3; j++ )
    {
        int globalPointIdx = cell.template getSubentityIndex< 0 >( j );
        nabla_eps[ globalPointIdx ] += grad;
        nabla_h_eps[ globalPointIdx ] += grad_h;
    }
};


// computing L(mesh)
double loss = L< PointType >( nabla_h, nabla );

// giving the points vector value of respective vertices
Containers::Vector< PointType > nabla_mesh( verticesCount );
nabla_mesh = 0;

auto kernel = [ &mesh, &nabla_mesh, &nabla_h, &nabla, &nabla_h_eps,
    &nabla_eps, get_nabla_h_eps ] ( GlobalIndexType i ) mutable
{
    // first partial derivative
    PointType eps0 = { EPSILON, 0 };
    mesh.getPoints()[ i ] += eps0;
    mesh.template forAll< MeshType::getMeshDimension() >( get_nabla_h_eps );
```

```
        nabla_mesh[ i ][ 0 ] = ( L< PointType >( nabla_h_eps, nabla_eps) - L<
            PointType >( nabla_h, nabla ) ) / EPSILON;
        mesh.getPoints()[ i ] -= eps0;
        nabla_h_eps = 0;
        nabla_eps = 0;

        // second partial derivative
        PointType eps1 = { 0, EPSILON };
        mesh.getPoints()[ i ] += eps1;
        mesh.template forAll< MeshType::getMeshDimension() >( get_nabla_h_eps );
        nabla_mesh[ i ][ 1 ] = ( L< PointType >( nabla_h_eps, nabla_eps) - L<
            PointType >( nabla_h, nabla ) ) / EPSILON;
        mesh.getPoints()[ i ] -= eps1;
        nabla_h_eps = 0;
        nabla_eps = 0;
    };
    mesh.template forAll< 0 >( kernel );

    Containers::Array< double > nabla_arr( 3 * verticesCount );
    nabla_arr = 0;
    for( int i = 0; i < 3 * verticesCount; i += 3 )
    {
        nabla_arr[ i ] = nabla_mesh[ i / 3 ][ 0 ];
        nabla_arr[ i + 1 ] = nabla_mesh[ i / 3 ][ 1 ];
    }

    // performing the gradient descent itself
    auto descent = [ &mesh, &nabla_mesh ] ( GlobalIndexType i ) mutable
    {
        mesh.getPoints()[ i ] -= 1e-4 * nabla_mesh[ i ]; // TODO change parameter
    };
    mesh.template forInterior< 0 >( descent );

    ... another part which is not listed ...

    return true;
}
```

4.3.3 Implementation of the Automatic differentiation approach

Using the C++ library autodiff, [8], one can obtain partial derivatives of a multivariable function (e.g. our loss function L) in certain points of its domain (e.g. vertices of a mesh) utilizing the technique of automatic differentiation.

A function `gradient()` for direct gradient computation is also implemented. This function returns an instance of `ArrayXreal` (or `ArrayXdual`), which is a data structure based on `ArrayX` from the library Eigen [12]. This, of course, allows usage of Eigen's methods to manipulate the result. On the other hand, there are compatibility issues among Eigen's and TNL's exclusive data structures and functions which need to be taken care of.

An illustrative example of usage of the autodiff library is presented below:

```
#include <iostream>
#include <autodiff/forward/dual.hpp>
#include <autodiff/forward/dual/eigen.hpp>
using namespace autodiff;

dual h( const ArrayXdual& x )
{
    return sqrt( ( x * x ).sum() );
}

int main()
{
    using Eigen::VectorXd;

    ArrayXdual x(5);
    x << 1, 2, 3, 4, 5;

    dual u;

    VectorXd nabla = gradient( h, wrt(x), at(x), u );

    std::cout << "u = " << u << "\n";
    std::cout << "nabla = \n" << nabla << "\n";

    return 0;
}
```

Having obtained the gradient vector in this fairly simple manner, one can perform a single step of gradient descent and iterate this process until a stop condition is met.

Unfortunately, we did not manage to overcome all the compatibility issues between TNL and autodiff (or TNL and Eigen, which is the foundation of autodiff) to produce consistent results without errors. Our best attempt at doing so, including implementation of TNL's mesh functions using the means of the autodiff and Eigen libraries, can be seen at [13] in the file `nablaAuto.cpp` and we consider it a task to be dealt with in the future.

Chapter 5

Computational study

In this section, we present some illustrative results of the implementations from chapter 4. The parameters for each computation are:

- initial mesh (number of cells, faces, vertices etc.),
- f, the scalar function whose gradient we approximate,
- λ, the relaxation parameter.

Plots of f on the meshed region are also displayed for comparison/formulation of expectations, since the algorithm tends to make the mesh denser in areas where f has relatively larger gradients than elsewhere. The plots were made using the python library matplotlib [10].

Three illustrative results using the finite difference method (4.3.2) are presented.

The stop condition of the algorithm is set as L(before iteration) – L(after iteration) $\overset{?}{<} 10^{-14}$. If the condition is met, the algorithm stops and returns the resulting mesh.

For all computations, $\varepsilon \equiv 10^{-8}$ was used. This seemed like the best fit for the functions we used. It is very much possible that if one chose a different set of functions, other values of ε would result in faster convergence, as discussed in 4.3.2.

5.1 Visualization of computed gradients

Before presenting actual optimized meshes, we show how the ∇L looks. We chose the sine function whose plot is well known and gives a good intuition of where ∇f should have high values and thus differ from $\nabla_h f$.

∇L for the sine function is shown in Figure 5.1 using Paraview [11]. The figure is in line with our intuition that the individual vectors formed from components of ∇L are symmetrical around origin both in norm and in direction.

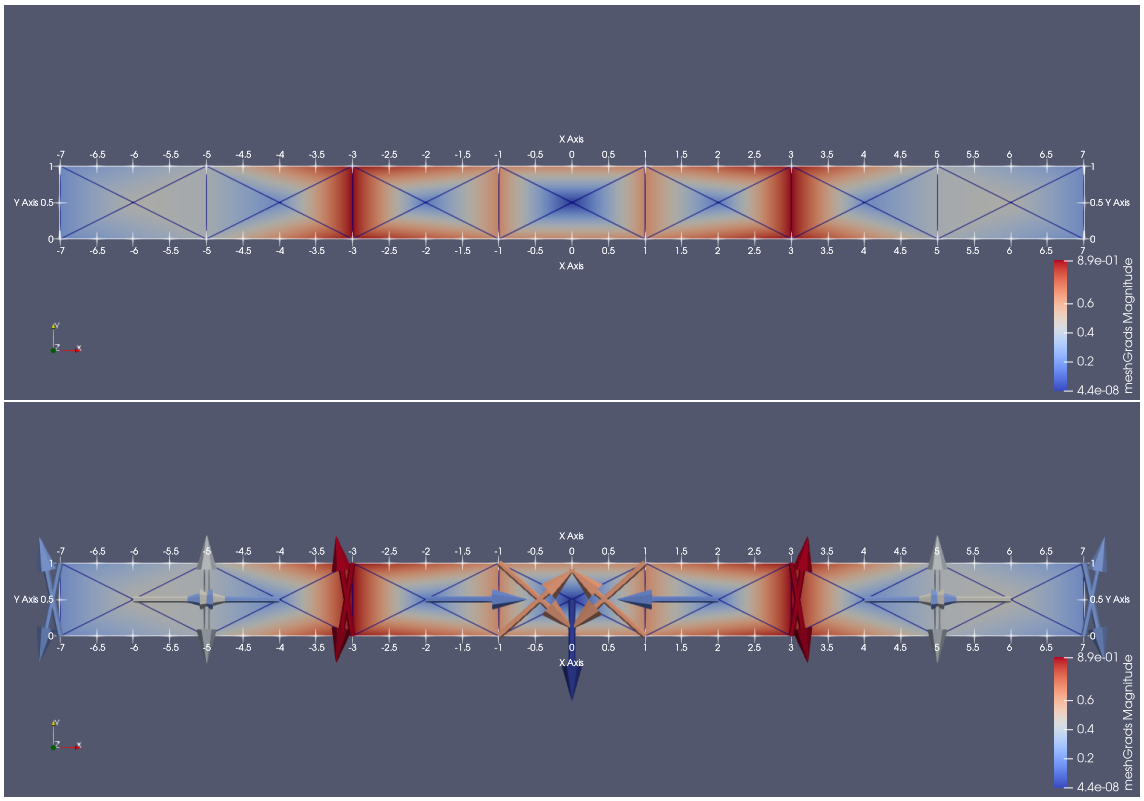


Figure 5.1: ∇L for the sine function, above as colormap of norms, below as actual vectors in (opposite of) whose direction the vertices move in the course of our optimization algorithm.

5.2 Sine function on a stripe

For the first computation, let $f(x, y) = sin(x)$. The mesh is a stripe along the x axis which is shown in Figure 5.2. This mesh is then slightly perturbed by the program described in Appendix B, resulting mesh is in the bottom half of Figure 5.2.

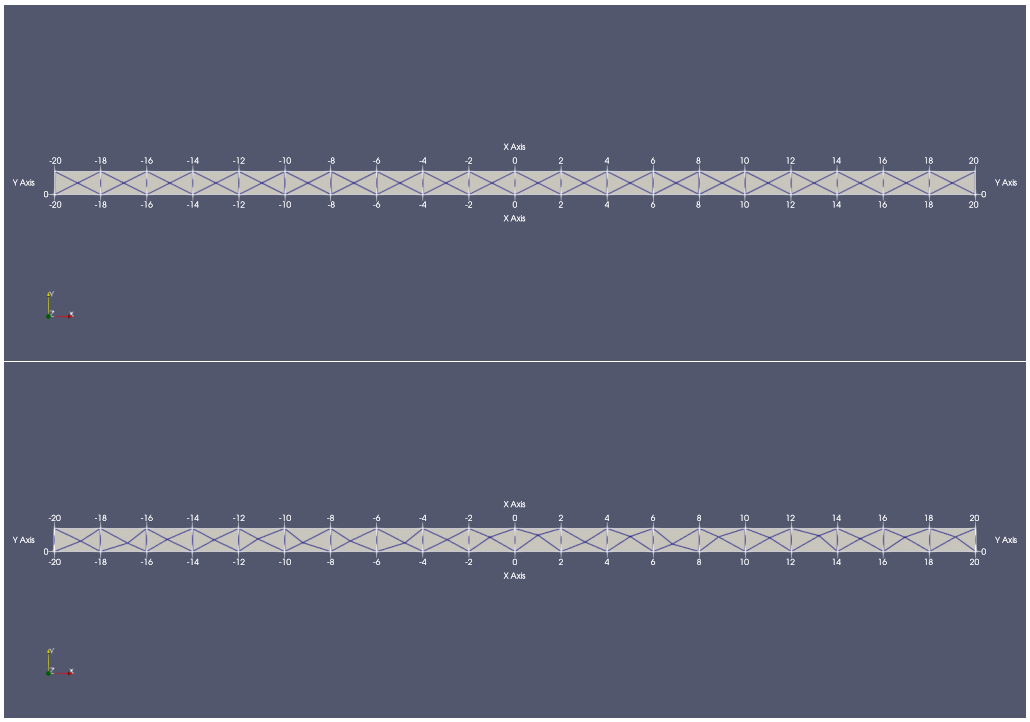


Figure 5.2: The initial stripe mesh and the same mesh with interior vertices randomly perturbed.

As expected, the optimal mesh denses up periodically around the roots of the sine function $x = k\pi$ where the gradient (derivative in this case) is the largest. The optimized meshes are shown in Figure 5.3, where in the first case, all vertices moved, in the second case, only the interior vertices were allowed to move.

The parameters of the computations are presented in the following tables, Table 5.1 and Table 5.2.

λ	10^{-4}
L_{ini}	7.61763
L_{final}	0.641525
iterations to converge	1186

Table 5.1: Parameters of the computation on a stripe, where all the vertices were allowed to move (Figure 5.3, top mesh).

5.3 Gaussian on a rectangle

Next, let us consider a 2D Gaussian, $f(x, y) = e^{-x^2-y^2}$. The mesh is a rectangle $[-1, 1] \times [0, 1]$, as shown in Figure 5.4.

λ	10^{-4}
L_{ini}	7.61763
L_{final}	0.932091
iterations to converge	932

Table 5.2: Parameters of the computation on a stripe, where only the interior vertices were allowed to move (Figure 5.3, bottom mesh).

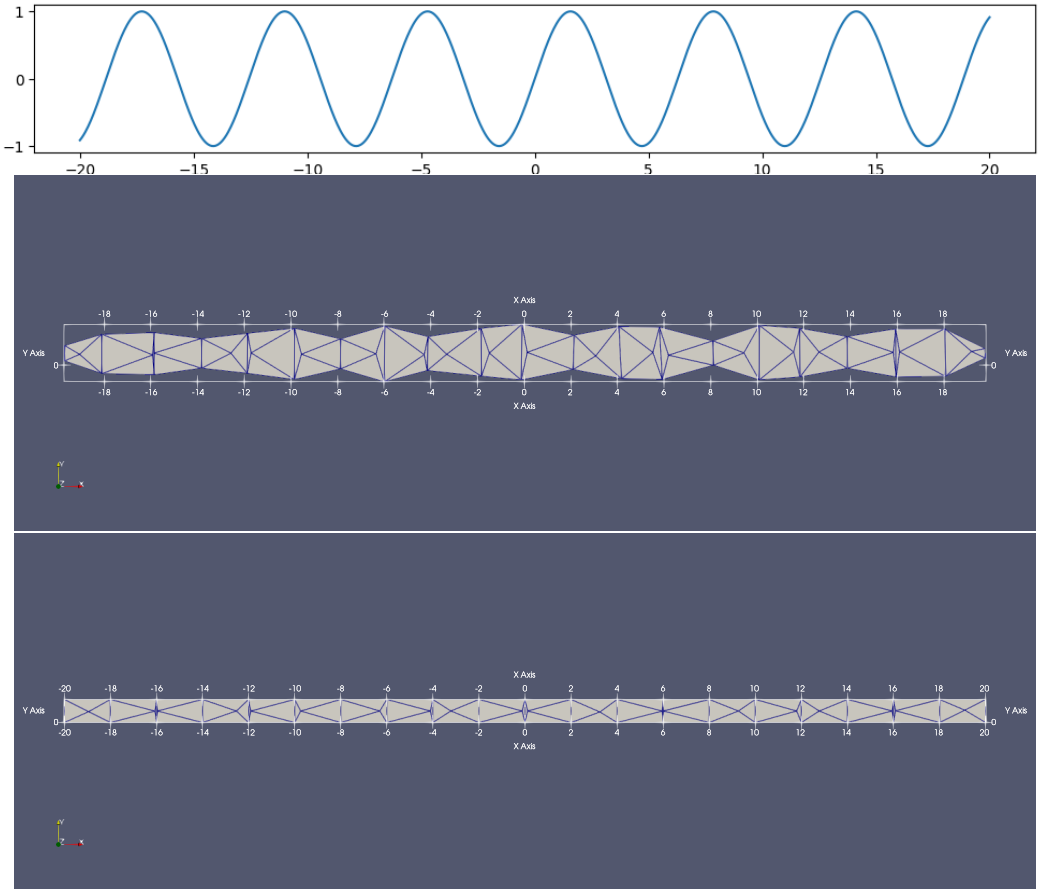


Figure 5.3: Optimized meshes. In the top one, all vertices were allowed to move, in the bottom one, the border vertices were fixed. In both cases, the optimized meshes are dense in areas where the sine function has derivative ≈ 1, whereas they are relatively sparse in areas with derivatives of sine function ≈ 0.

The optimization procedure resulted in meshes displayed in Figure 5.5, with movement of border vertices prohibited in the top case and allowed in the bottom case. In Table 5.3 and Table 5.4, we present parameters of the computations.

Geometrically, it is apparent that both optimized meshes are dense around a circle with a diameter $d < 1$. The Gaussian's growth is the most rapid in this area[1] and it is thus logical, that the mesh

[1]See the bottom picture in Figure 5.5 for intuition. A more thorough approach would be to examine the second order derivative of the Gaussian to see where its first order derivatives change fast. From this process, one can conclude that we expect smaller approximation errors right at the origin and very far away from the origin. We expect the mesh

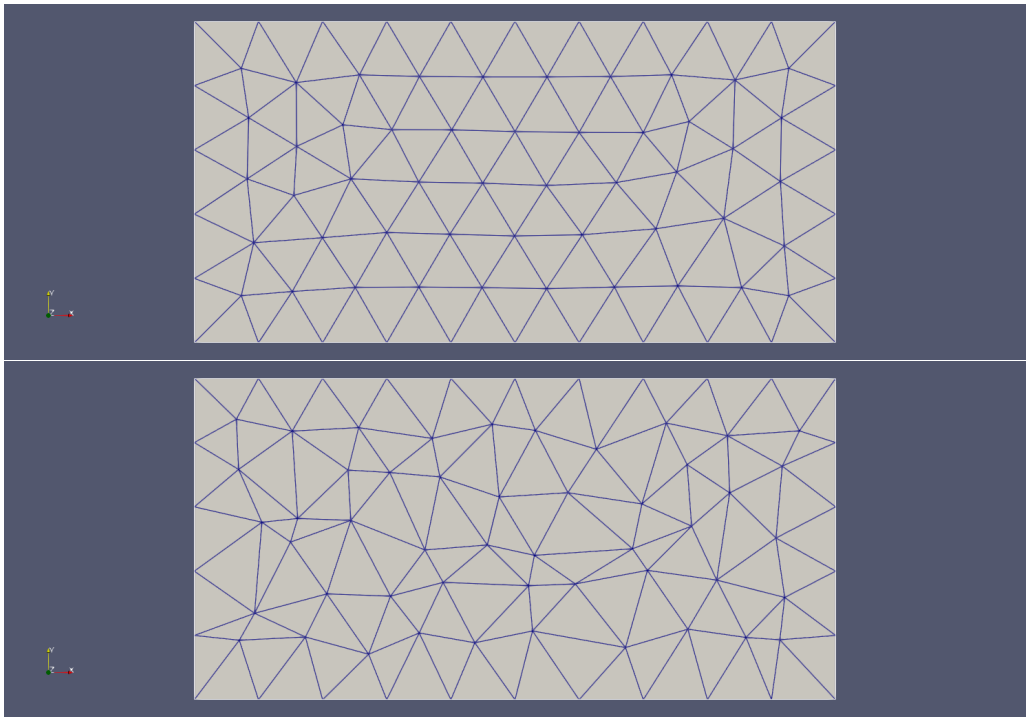


Figure 5.4: The initial rectangular mesh and the same mesh with interior vertices randomly perturbed.

λ	10^{-4}
L_{ini}	0.0475275
L_{final}	0.0097512
iterations to converge	1755

Table 5.3: Parameters of the computation on a stripe, where only the interior vertices were allowed to move (Figure 5.5, top mesh).

λ	10^{-4}
L_{ini}	0.0475275
L_{final}	0.000325888
iterations to converge	22166

Table 5.4: Parameters of the computation on a stripe, where all the vertices were allowed to move (Figure 5.5, bottom mesh).

needs to be dense there to reflect this fact and allow for a better gradient approximation. On the rectangular mesh from this section, we can obviously only see the top half of this annulus.

5.4 Gaussian on a square

The following example is perhaps even more visual. Once again, our f is the Gaussian, $f(x,y) = e^{-x^2-y^2}$, but this time, the meshed region is the square $[-2,2]\times[-2,2]$. Once again, we can clearly

to dense up in an annulus with center in the origin.

see the mesh densing up in a circle around the origin, where f has a high value of the second order derivative. In the immediate neighborhood of the origin, the second derivative of f is smaller and the approximation works fine on relatively larger cells.

The initial mesh is in Figure 5.6, the optimized meshes (one where all the vertices could move and one where the movement of the boundary vertices was prohibited) are then in Figure 5.7.

Tables Table 5.5 and Table 5.6 summarize parameters of the computations.

λ	10^{-4}
L_{ini}	0.0646745
L_{final}	0.0022915
iterations to converge	37083

Table 5.5: Parameters of the computation on a square, where only the interior vertices were allowed to move (Figure 5.7, left mesh).

λ	10^{-4}
L_{ini}	0.0646745
L_{final}	0.000868945
iterations to converge	94647

Table 5.6: Parameters of the computation on a square, where all the vertices were allowed to move (Figure 5.7, right mesh).

5.5 Improvement of L quantified

Other than intuitively seeing that a mesh becomes denser where we expected it to, we can, of course, benchmark the process of improvement quantitatively by checking how the values of the loss function L have decreased throughout the run of the program. In other words, we look at how the finite volume approximation $\nabla_h f$ has been becoming more similar to the actual gradient ∇f during the iterations of the algorithm.

Specifically for the computation of the mesh on the right side of Figure 5.7, we present two graphs in Figure 5.8 made with [10]. On the x axis, there is the number of iterations. The value of L in individual iterations is represented on the y axis with samples taken every 10000 iterations, and in the final iteration 94647 respectively. In the top graph, the scale of the y axis is linear, in the bottom one, the same data is presented on a logarithmic scale on the y axis.

Both graphs indicate that most of the improvement occurs in the first iterations. That is why we examine the first ten thousand iterations in another graph, Figure 5.9 with samples taken every 500 iterations, and once again with both linear and logarithmic scale for the y axis. We can see that these graphs look very similar to those in Figure 5.8, which further supports the statement that the mesh improves the fastest in the very beginning and with each new iteration, the improvement decreases.

Addition of these graphs for the other computations would not provide any new insight into the topic, since the shape of the curve in them is always very similar to those from Figure 5.8.

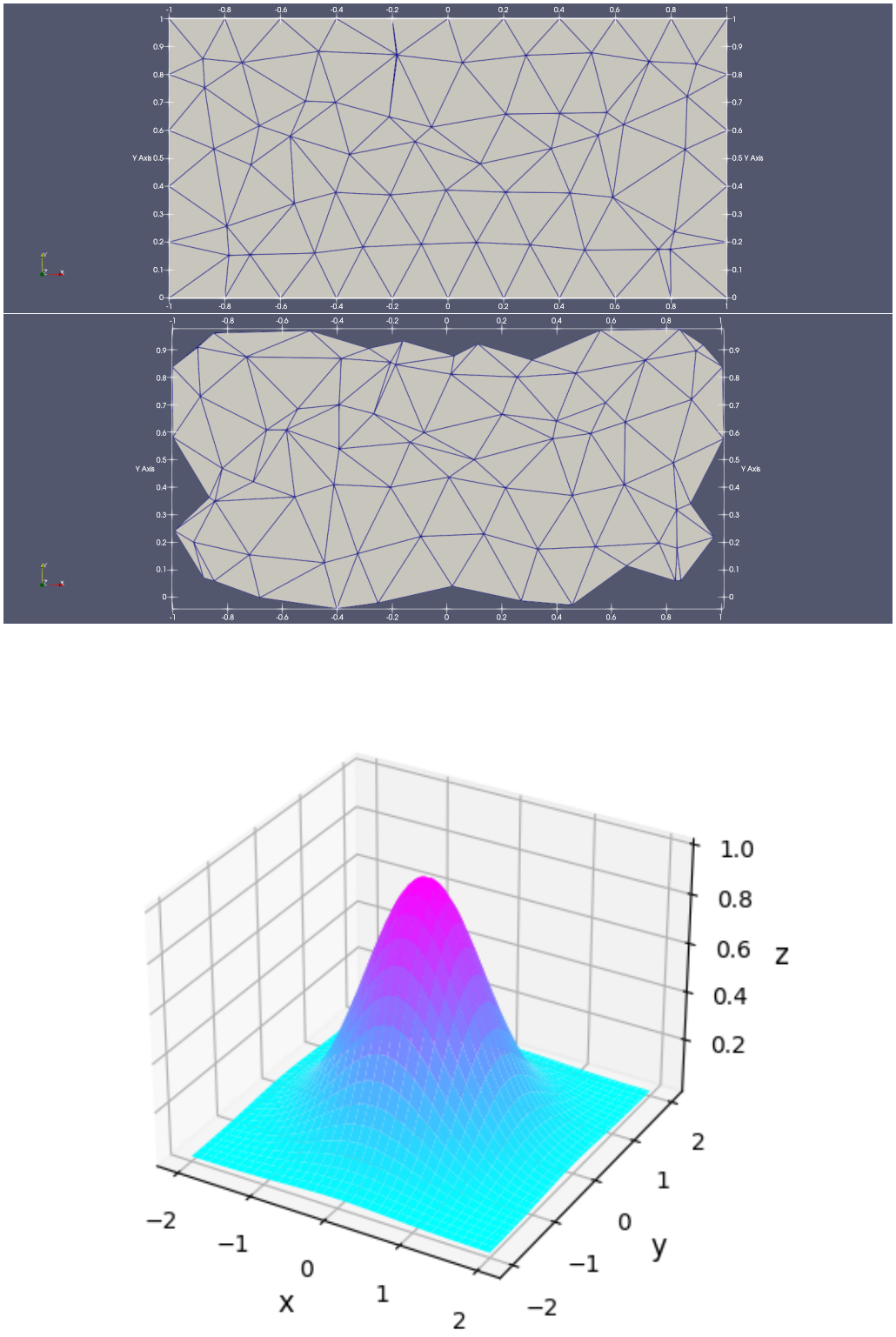


Figure 5.5: The optimized meshes. In the top case, only the interior vertices moved, in the bottom case, movement of all vertices was allowed. In both cases, a semicircular area of smaller, denser cells can be identified. A plot of the Gaussian on $[-2, 2] \times [-2, 2]$ is added for perspective.

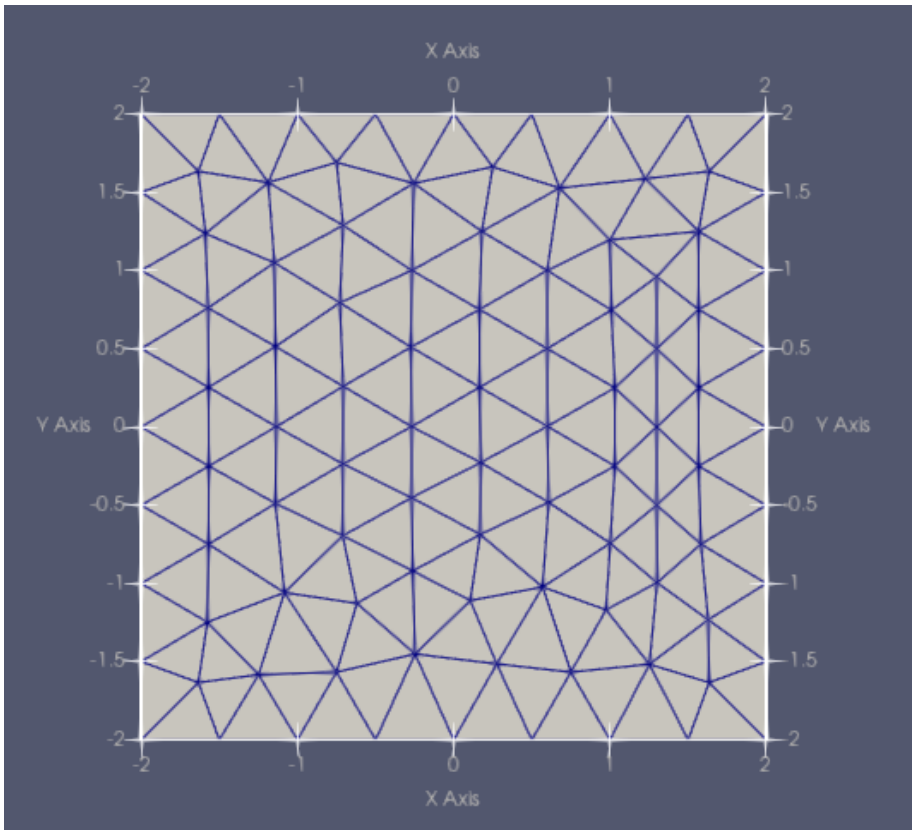


Figure 5.6: The initial mesh on a $[-2, 2] \times [-2, 2]$ square.

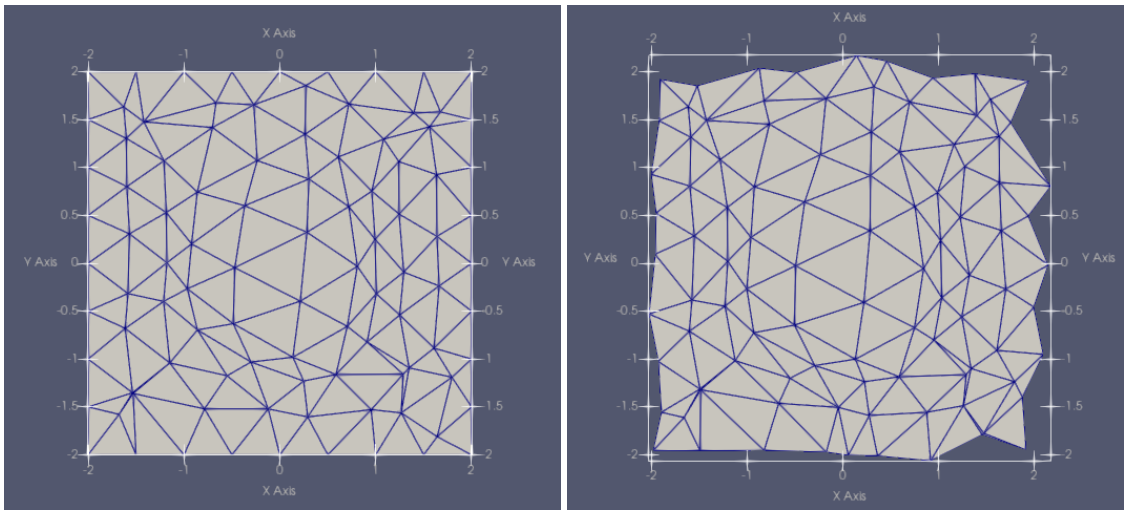


Figure 5.7: The optimal meshes, only interior vertices were allowed to move on the left, all vertices moved on the right.

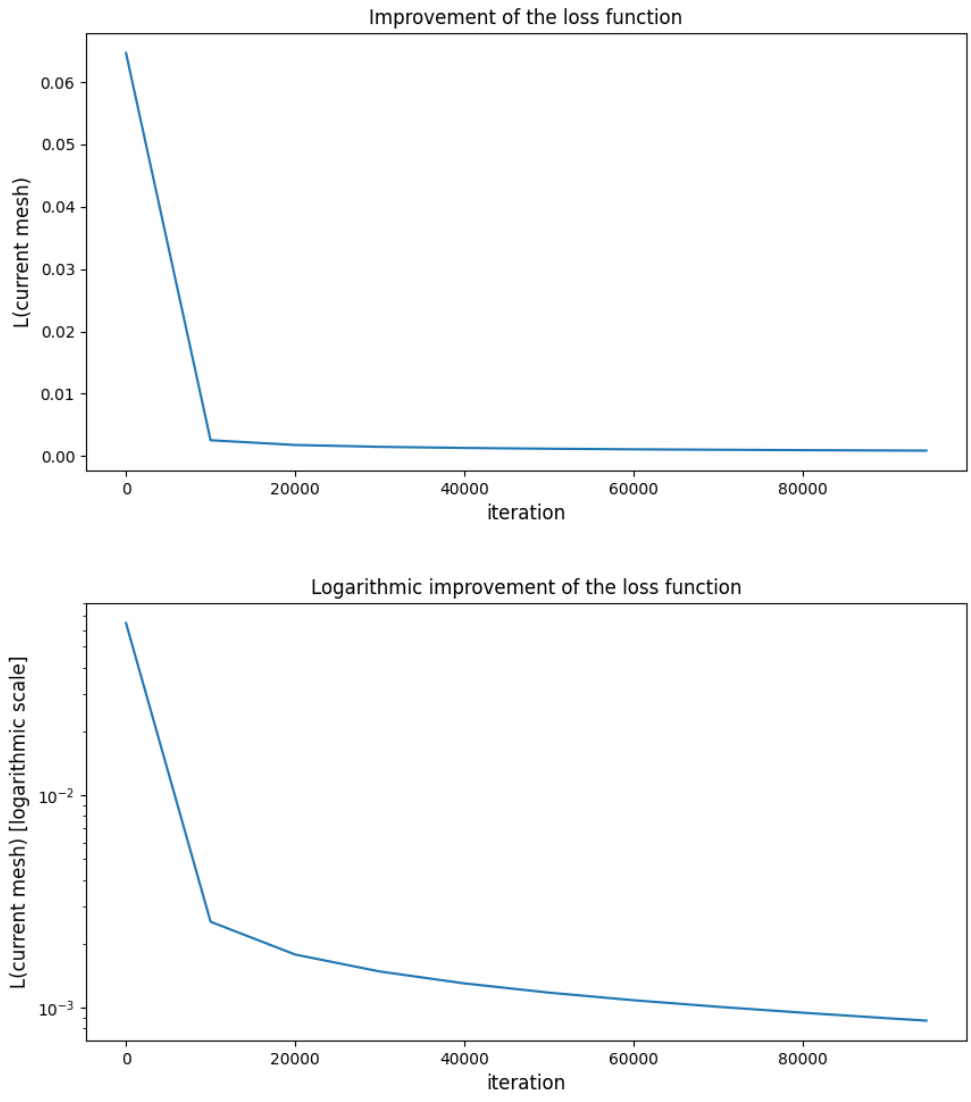


Figure 5.8: Improvement of L during the computation from Table 5.6. The scale of the y axis is linear in the top graph and logarithmic in the bottom one.

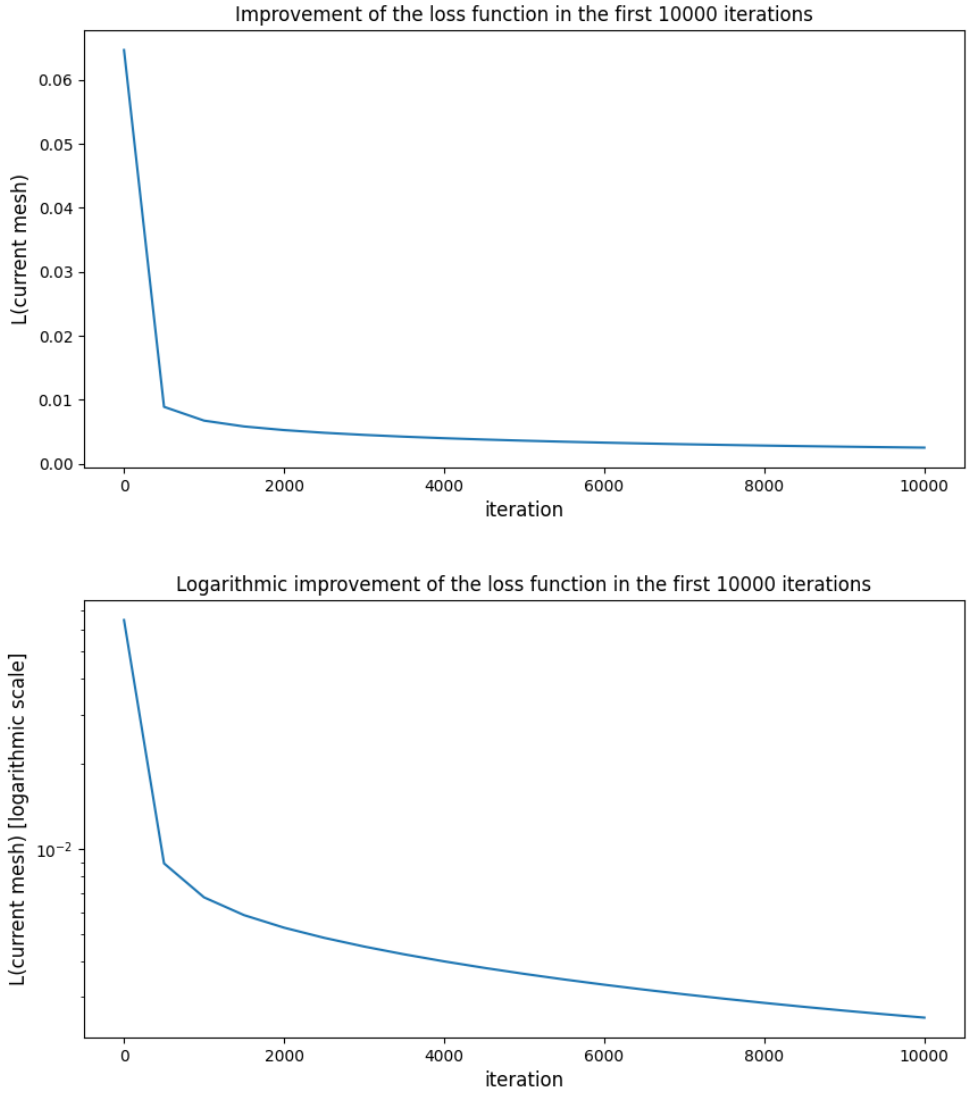


Figure 5.9: A closer look at the first ten thousand iterations of the computation.

Conclusion

In this Bachelor project, we have developed an algorithm of mesh optimization for finite volume method gradient approximation. Our aim was to modify the geometry of a triangular mesh in such a way, that the finite volume approximation would be as close to the actual gradient as possible.

We introduced the necessary theoretical foundations for this optimization from the fields of calculus, numerical mathematics and computer implementation of meshes.

Subsequently, we formulated our objective as an optimization problem. A minimum of the objective function L represents the mesh, for which the finite volume gradient approximation is optimal.

To solve this problem, we used a gradient descent method. We explored different ways to obtain a function's gradient, namely in-hand analytical differentiation, finite differences and automatic differentiation.

The analytical approach proved to be extremely inefficient and we did not manage to overcome the compatibility issues between the library TNL for mesh manipulation and the autodiff library for automatic differentiation.

We did, however, find success implementing an optimization algorithm using first order finite differences as an approximation of actual derivatives. We present this algorithm as a C++ program which employs highly efficient, GPU-ready functions implemented in TNL.

Finally, we provide illustrative outputs of a computational study performed using the program for several meshes and functions which allow for an intuitive validation of the results.

We employed gmsh for generation of the meshes used as inputs for our program. For visualization of results of the computations, ParaView and MatPlotLib were used.

Bibliography

[1] A. Quarteroni, R. Sacco, F. Saleri: *Numerical mathematics*. Springer, 2010.

[2] J. Nocedal, S. J. Wright: *Numerical optimization*. Springer, 2006.

[3] C. Geuzaine, J.-F. Remacle: *Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities*. International Journal for Numerical Methods in Engineering 79(11), pp. 1309-1331, 2009.

[4] T. Oberhuber, J. Klinkovský: *TNL – Template Numerical Library* website (July 14, 2023). `https://tnl-project.org`

[5] T. Oberhuber, J. Klinkovský: *Template Numerical Library: User's Guide* (cited August 2, 2023). `https://tnl-project.gitlab.io/tnl/md_UsersGuide_users_guide.html#UsersGuide`

[6] J. Klinkovský, T. Oberhuber, R. Fučík, V. Žabka: *Configurable Open-source Data Structure for Distributed Conforming Unstructured Homogeneous Meshes with GPU Support*. ACM Transactions on Mathematical Software, 48(3), 1–30, 2022. `https://doi.org/10.1145/3536164`

[7] J. Bobot: *Implementace datové struktury pro polyhedrální sítě v knihovně TNL*. Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

[8] A. M. M. Leal: *autodiff, a modern, fast and expressive C++ library for automatic differentiation*. 2018. `https://autodiff.github.io`

[9] R. Soklaski: *MyGrad: Drop-in autodiff for numpy.* `https://github.com/rsokl/MyGrad`

[10] J. D. Hunter: *Matplotlib: A 2D graphics environment*. Computing in Science & Engineering, 9(3), 90-95, 2007. doi 10.1109/MCSE.2007.55.

[11] J. Ahrens, B. Geveci, C. Law: *ParaView: An End-User Tool for Large Data Visualization*, Visualization Handbook, Elsevier, 2005. `www.paraview.org`

[12] G. Guennebaud, B. Jacob et al.: *Eigen v3*, 2010. `http://eigen.tuxfamily.org`

[13] `https://github.com/PrinceOfCzechia/Bachelor-Thesis-Auxiliary-Code`, GitHub repository containing full versions of all the presented code, implemented by P. Král, author of this Bachelor project.

Appendix A

Analytical derivatives

This part demonstrates detailed process of calculation of the derivatives of the loss function L introduced in section 1.4. While this procedure is highly impractical, it was done and is thus presented in this Bachelor project's body.

Derivatives of $m^{(2)}(C_i)$

Let us pick any vertex in the cell C_i and two vectors which connect it to the other two vertices, for example $\mathbb{A} = \begin{pmatrix} \boldsymbol{\sigma}_1^{\mathbf{2}} & -\boldsymbol{\sigma}_1^{\mathbf{0}} \\ \boldsymbol{\sigma}_2^{\mathbf{2}} & -\boldsymbol{\sigma}_2^{\mathbf{0}} \end{pmatrix}$. Measure of the triangle C_i is now $m^{(2)}(C_i) = \frac{1}{2}\det(\mathbb{A})$. A geometric interpretation can be seen in Figure A.1, the area of the magenta rhombus is $\det(\mathbb{A})$, area of C_i is $\frac{1}{2}\det(\mathbb{A})$.

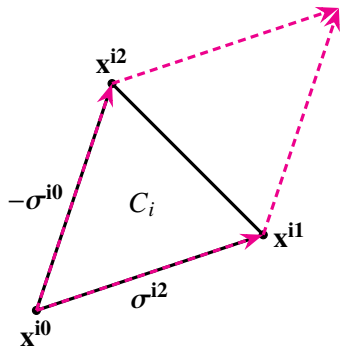


Figure A.1: Measure of a triangular cell.

Let us now express the formula in terms of $\mathbf{x^{ij}}$ and differentiate. Note that there is an absolute value in the expression we intend to derive, but since the points of a triangle are never identical, we never calculate $\frac{\partial}{\partial x}|x|\Big|_{x=0}$ which does not exist. Having eliminated this eventuality, we can use the formula $\partial_x|x| = \mathrm{sgn}(x)$.

$$\begin{aligned} m^{(2)}(C_i) &= \frac{1}{2}|\det \mathbb{A}_i| = \frac{1}{2}\Big| - \boldsymbol{\sigma^2}_1\boldsymbol{\sigma^0}_2 + \boldsymbol{\sigma^2}_2\boldsymbol{\sigma^0}_1\Big| \\ &= \frac{1}{2}\Big|(\mathbf{x^{i2}} - \mathbf{x^{i0}})_1(\mathbf{x^{i1}} - \mathbf{x^{i0}})_2 - (\mathbf{x^{i1}} - \mathbf{x^{i0}})_1(\mathbf{x^{i2}} - \mathbf{x^{i0}})_2\Big| \\ &= \frac{1}{2}\Big|\mathbf{x^{i2}}_1\mathbf{x^{i1}}_2 - \mathbf{x^{i0}}_1\mathbf{x^{i1}}_2 - \mathbf{x^{i2}}_1\mathbf{x^{i0}}_2 - \mathbf{x^{i1}}_1\mathbf{x^{i2}}_2 + \mathbf{x^{i1}}_1\mathbf{x^{i0}}_2 + \mathbf{x^{i0}}_1\mathbf{x^{i2}}_2\Big|. \end{aligned}$$

Let us substitute $\mathrm{sgn}(\mathbf{x^{i2}}_1\mathbf{x^{i1}}_2 - \mathbf{x^{i0}}_1\mathbf{x^{i1}}_2 - \mathbf{x^{i2}}_1\mathbf{x^{i0}}_2 - \mathbf{x^{i1}}_1\mathbf{x^{i2}}_2 + \mathbf{x^{i1}}_1\mathbf{x^{i0}}_2 + \mathbf{x^{i0}}_1\mathbf{x^{i2}}_2) \equiv \mathrm{S}$. The derivative by the first component of the point $\mathbf{x^k}$ is

$$\partial_{k_1} m^{(2)}(C_i) = \frac{1}{2}\mathrm{S}\Big(\delta_{i2k1}\mathbf{x^{i1}}_2 - \delta_{i0k1}\mathbf{x^{i1}}_2 - \delta_{i2k1}\mathbf{x^{i0}}_2 - \delta_{i1k1}\mathbf{x^{i2}}_2 + \delta_{i1k1}\mathbf{x^{i0}}_2 + \delta_{i0k1}\mathbf{x^{i2}}_2\Big).$$

Similarly, differentiating by the second component of $\mathbf{x^k}$, we get

$$\partial_{k_2} m^{(2)}(C_i) = \frac{1}{2}\mathrm{S}\Big(\mathbf{x^{i2}}_1\delta_{i1k2} - \mathbf{x^{i0}}_1\delta_{i1k2} - \mathbf{x^{i2}}_1\delta_{i0k2} - \mathbf{x^{i1}}_1\delta_{i2k2} + \mathbf{x^{i1}}_1\delta_{i0k2} + \mathbf{x^{i0}}_1\delta_{i2k2}\Big).$$

In this particular calculation, we need the derivatives of $\frac{1}{m^{(2)}(C_i)}$, or $(m^{(2)}(C_i))^{-1}$, which are, based on the results above, equal to

$$\partial_{k_1}(m^{(2)}(C_i))^{-1} = \begin{cases} \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(-\mathbf{x^{i1}}_2 + \mathbf{x^{i2}}_2) & \text{if } k_1 = i0 \\ \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(-\mathbf{x^{i2}}_2 + \mathbf{x^{i0}}_2) & \text{if } k_1 = i1 \\ \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(\mathbf{x^{i1}}_2 - \mathbf{x^{i0}}_2) & \text{if } k_1 = i2 \\ 0 & \text{else} \end{cases}, \tag{A.1}$$

and

$$\partial_{k_2}(m^{(2)}(C_i))^{-1} = \begin{cases} \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(-\mathbf{x^{i2}}_1 + \mathbf{x^{i1}}_1) & \text{if } k_1 = i0 \\ \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(\mathbf{x^{i2}}_1 - \mathbf{x^{i0}}_2) & \text{if } k_1 = i1 \\ \frac{-1}{2\left(m^{(2)}(C_i)\right)^2}\mathrm{S}(-\mathbf{x^{i1}}_1 + \mathbf{x^{i0}}_1) & \text{if } k_1 = i2 \\ 0 & \text{else} \end{cases}, \tag{A.2}$$

respectively.

Derivatives of $m^{(1)}(\sigma^{ij})$

The faces σ of each cell C_i, $i \in \hat{\mathrm{N}}$, can be interpreted as vectors $\boldsymbol{\sigma^{ij}}$, where $j \in \{0, 1, 2\}$. For each j, the following holds: $\boldsymbol{\sigma^{ij}} = \mathbf{x^{i(j+2)\%3)}} - \mathbf{x^{i(j+1)\%3}}$, i.e. the counter clockwise property of faces holds. The derivatives $\partial_{k_1} m^{(1)}(\sigma^{ij})$ are equal to $\partial_{k_1}\|\boldsymbol{\sigma}^{ij}\|$.

$$\partial_{k_1}\|\boldsymbol{\sigma}^{ij}\| = \partial_{k_1}\|(\mathbf{x^{i(j+2)\%3}} - \mathbf{x^{i(j+1)\%3}})\| = \partial_{k_1}\left[(\mathbf{x^{i(j+2)\%3}}_1 - \mathbf{x^{i(j+1)\%3}}_1)^2 + (\mathbf{x^{i(j+2)\%3}}_2 - \mathbf{x^{i(j+1)\%3}}_2)^2\right]^{\frac{1}{2}}$$

Finishing this calculation gives

$$\partial_{k_1}[(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)^2 + (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)^2]^{\frac{1}{2}}$$
$$= \frac{1}{2}[(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)^2 + (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)^2]^{-\frac{1}{2}} \cdot 2 \cdot (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)(\delta_{i(j+2)\%3,k1} - \delta_{i(j+1)\%3,k1}).$$

Similarly for ∂_{k_2}

$$\partial_{k_2}[(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)^2 + (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)^2]^{\frac{1}{2}}$$
$$= \frac{1}{2}[(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)^2 + (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)^2]^{-\frac{1}{2}} \cdot 2 \cdot (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)(\delta_{i(j+2)\%3,k2} - \delta_{i(j+1)\%3,k2}).$$

Derivatives of $\mathbf{n}^{ij}$

A unit outward normal to the vector $\begin{pmatrix}\boldsymbol{\sigma}_1^{\mathbf{ij}} & \boldsymbol{\sigma}_2^{\mathbf{ij}}\end{pmatrix}$ is the vector $\mathbf{n}^{ij} = \frac{1}{\|\boldsymbol{\sigma}^{\mathbf{ij}}\|}\begin{pmatrix}\boldsymbol{\sigma}_2^{\mathbf{ij}} & -\boldsymbol{\sigma}_1^{\mathbf{ij}}\end{pmatrix}$, given the border of the polygon, whose part the vector $\boldsymbol{\sigma}^{\mathbf{ij}}$ is, is counter clockwise oriented.

This fact can be easily derived, e.g. by drawing a sample polygon. This is presented in the following Figure A.2.

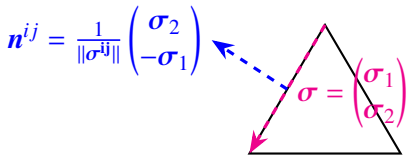


Figure A.2: Normalized outward normal vector of a face.

In terms of mesh points $\mathbf{x}^{\mathbf{ij}}$,

$$\mathbf{n}^{\mathbf{ij}} = [(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)^2 + (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)^2]^{-\frac{1}{2}} \begin{pmatrix}\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2 \\ -\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 + \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1\end{pmatrix},$$

and its derivatives are

$$\partial_{k_1}\mathbf{n}^{\mathbf{ij}} = -\frac{1}{2}\Big[(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)^2 + (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)^2\Big]^{-\frac{3}{2}} 2(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)(\delta_{i(j+2)\%3,k1} - \delta_{i(j+1)\%3,k1})$$
$$\cdot \begin{pmatrix}\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2 \\ -\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 + \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1\end{pmatrix} + \Big[(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)^2 + (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)^2\Big]^{-\frac{1}{2}} \begin{pmatrix}0 \\ \delta_{i(j+2)\%3,k1} - \delta_{i(j+1)\%3,k1}\end{pmatrix},$$

$$\partial_{k_2}\mathbf{n}^{\mathbf{ij}} = -\frac{1}{2}\Big[(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)^2 + (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)^2\Big]^{-\frac{3}{2}} 2(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)(\delta_{i(j+2)\%3,k2} - \delta_{i(j+1)\%3,k2})$$
$$\cdot \begin{pmatrix}\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2 \\ -\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 + \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1\end{pmatrix} + \Big[(\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_1 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_1)^2 + (\mathbf{x}^{\mathbf{i(j+2)\%3}}{}_2 - \mathbf{x}^{\mathbf{i(j+1)\%3}}{}_2)^2\Big]^{-\frac{1}{2}} \begin{pmatrix}\delta_{i(j+2)\%3,k2} - \delta_{i(j+1)\%3,k2} \\ 0\end{pmatrix}.$$

Derivatives of $\mathbf{x}^{\sigma^{\mathbf{ij}}}$

The vector $\mathbf{x}^{\sigma^{ij}}$ is equal to

$$\mathbf{x}^{\sigma^{ij}} = \frac{\mathbf{1}}{\mathbf{2}}(\mathbf{x}^{\mathbf{i(j+1)\%3}} + \mathbf{x}^{\mathbf{i(j+2)\%3}}).$$

By differentiating these vectors $\mathbf{x}^{\sigma^{ij}}$, we obtain

$$\partial_{k_1}\mathbf{x}^{\sigma^{\mathbf{ij}}} = \frac{1}{2}\left(\begin{pmatrix}\delta_{i((j+1)\%3),k1}\\0\end{pmatrix} + \begin{pmatrix}\delta_{i((j+2)\%3),k1}\\0\end{pmatrix}\right),$$

$$\partial_{k_2}\mathbf{x}^{\sigma^{ij}} = \frac{\mathbf{1}}{\mathbf{2}}\left(\begin{pmatrix}0\\\delta_{i((j+1)\%3),k2}\end{pmatrix} + \begin{pmatrix}0\\\delta_{i((j+2)\%3),k2}\end{pmatrix}\right).$$

Appendix B

Breaking meshes

Since mesh generators such as *gmsh* [3] tend to apply their own internal optimization procedures and produce quite uniform, symmetrical meshes, we concluded it might be interesting to apply optimization algorithms of this Bachelor project not only on the raw gmsh meshes, but also break them a little first.

By breaking a mesh, we mean adding a random perturbation vector to each vertex of the mesh in such a way, that the resulting mesh has the same topology as the original one. The geometry, however, varies.

In Figure B.1, we present one possible case of a raw gmsh mesh and the same mesh "broken" by the program.

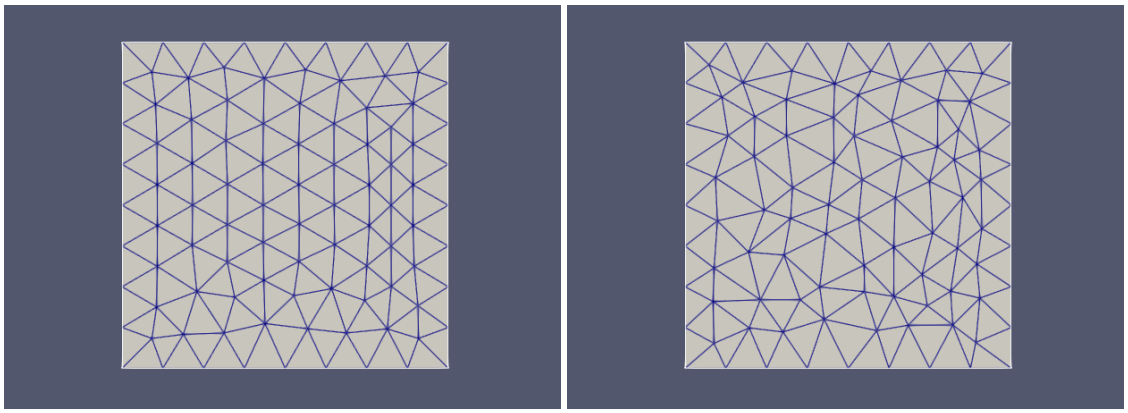


Figure B.1: A mesh produced by gmsh on the left, the same mesh after each interior vertex was moved by a random perturbation on the right. Both images were produced in Paraview [11].

Below is the code of a program we wrote to perform this mesh deformation. It always checks that any given point moves by less than by the length of the shortest face in the mesh (marked `minL` in the code). For extra safety, there is a parameter `P` which is set to < 1 and further restricts movement of the vertices.

Note that this procedure essentially generates a different starting point for the gradient descent but does not change the resulting optimal mesh which is reachable from any starting mesh. In

other words, whether we use the output of this program, or the raw gmsh mesh as the input of our optimization algorithm, its output remains the same.

In the listed part of the code, we excluded the initial `include`, `namespace` and "formal `TNL`" declarations which do not bring anything new to the table. The whole code is at disposal for anyone at [13] in the file `meshbreaker.cpp`.

```
const double P = 2.0e-1;

template< typename MeshConfig >
bool breakMesh( Mesh< MeshConfig, Devices::Host >& mesh, const std::string&
    fileName )
{
    using MeshType = Mesh< MeshConfig, Devices::Host >;
    using RealType = typename MeshType::RealType;
    using GlobalIndexType = typename MeshType::GlobalIndexType;
    using LocalIndexType = typename MeshType::LocalIndexType;
    using VectorType = TNL::Containers::Vector< RealType, TNL::Devices::Host,
        GlobalIndexType >;
    using PointType = typename MeshTraits< MeshConfig >::PointType;

    // getting the length of the shortest face in the mesh
    double minL = getEntityMeasure( mesh, mesh.template getEntity<
        MeshType::getMeshDimension() - 1 >( 0 ) );
    auto getShortestFace = [ &mesh, &minL ] ( GlobalIndexType i ) mutable
    {
        auto measure = getEntityMeasure( mesh, mesh.template getEntity<
            MeshType::getMeshDimension() - 1 >( 0 ) );
        if( measure < minL ) minL = measure;
    };
    mesh.template forAll< MeshType::getMeshDimension() - 1 >( getShortestFace );

    // adding a random perturbation
    auto breakMesh = [ &mesh, &minL ] ( GlobalIndexType i ) mutable
    {
        mesh.getPoints()[ i ][ 0 ] += P * minL * ( ((double) rand() / (RAND_MAX))
            * 2 - 1 );
        mesh.getPoints()[ i ][ 1 ] += P * minL * ( ((double) rand() / (RAND_MAX))
            * 2 - 1 );
    };
    mesh.template forInterior< 0 >( breakMesh );

    // writing the perturbed mesh into a new file
    using VTKWriter = Meshes::Writers::VTKWriter< MeshType >;
    std::ofstream out = std::ofstream( "broken.vtk" );
    VTKWriter writer = VTKWriter( out );
    writer.template writeEntities< MeshType::getMeshDimension() >( mesh );

    std::cout << "Mesh broken succesfully" << "\n";
    return true;
}

int main( int argc, char* argv[] )
{
```

```
    if( argc < 2 )
    {
        std::cerr << "Usage: " << argv[ 0 ] << " [mesh file adress]" << "\n";
        return EXIT_FAILURE;
    }

    bool result = true;

    for( int i = 1; i < argc; i++ )
    {
        const std::string fileName = argv[ i ];
        auto wrapper = [&]( auto& reader, auto&& mesh ) -> bool
        {
            return breakMesh(mesh, "");
        };
        result &= resolveAndLoadMesh< MyConfigTag, Devices::Host >( wrapper,
            fileName );
    }

   return 0;
}
```