Universität Ulm
Fakultät für Mathematik und Wirtschaftswissenschaften

Automatic differentiation in flow simulation

Diplomarbeit

in Mathematik

vorgelegt von
Ralf, Leidenberger
am August 30, 2007

Gutachter

Prof. Dr. Karsten Urban
Prof. Dr. Stefan Funken

Acknowledgements

First of all, my special thanks goes to Professor Dr. Karsten Urban, Insitute of Numerical Mathematics, Ulm University, Germany, for entrusting me with the challenging topic of my diploma thesis and for lending me his precious time and valueable advice, especially in makeing this work feasible.
I wish to thank Professor Dr. Stefan Funken, Insitute of Numerical Mathematics, Ulm University, Germany, for the support and in advance for the survey of this work.

Additional thanks are going to Professor Dr.-Ing. Milovan Perić and Dr. Eberhard Schreck from the company CD-adapco for their support in questions concerning the program Comet.

I appreciate the sacrifiing offer from my father Wilhelm Leidenberger and from my girlfriend Julia Hoffmann for reading my revisions and for giving usefull advice for English formulations.

Finally, special thanks goes to my girlfriend Julia Hoffmann for her encouragement in the course of this half year.

August 30th 2007 Ralf Leidenberger

Contents

List of Figures

List of Tables

Chapter 1

Introduction

At the beginning of this diploma thesis we explain the structure and the assembling of this work. And we give a small overview about the general topics and the destinations of this work. We start with the explanation of the general topics.

1.1 General topics

The title of this work is **"Automatic differentiation in flow simulation"**, this title reflects the two main topics automatic differentiation[1] and the numerical flow simulation.
The pure automatic differentiation part is splited into two parts. The theoretical part explains the method and different kinds of automatic differentiation, and it also compares these approaches. The other one explains a special implementation of an operator overloading tool.
The other main topic, the numerical flow simulation, starts with a derivation of the physical flow model, whereby one is called Euler equations and the other (the more detailed) one is called Navier-Stokes equations. Then we explain the simulation of the Navier-Stokes equations with different kinds, at first with the easier finite differences method and then with the finite volume method.
The subtopics of this diploma thesis are the numerical optimization, especially the comparison of algorithms, which uses and does not use informations about the derivative. And this implies the other subtopic, the application of automatic differentiation on numerical flow simulations for numerical optimization with different approaches for the optimization and for the different kinds of numerical flow simulation.
The structure of this work is: we start with an introduction in the numerical optimization and a small comparison of some different methods. Then we give a derivation of the flow model and two numerical methods to simulate the flow numerically. The next two chapters will explain the automatic differentiation and the special implementation of an operator overloading tool. Then, we have the basics in order to apply the automatic differentiation on the numerical flow simulation, with the aim optimization. Finally, we summarize the results, give an overview about the results, the problems and an outlook.

[1] A conventional shortcut is AD, and another popular name is algorithmic differentiation.

1.2 Destination

The destination of this diploma thesis is divided in different aims:
The first aim is to give an overview and an introduction of the automatic differentiation, to write the theoretical results about the automatic differentiation in a formally correct way down, to compare the different kinds of AD, and to explain the functionality of an operator overloading automatic differentiation tool on a special implementation.
The second one is to derive the model of the flow, that means we explain the physical assumptions and rules, which are necessary to desribe the flow of a fluid, in a mathematical formulation. This allows us to consider the methods for the flow simulation. Then, we give an introduction to the numerical methods, which are needed in order to simulate the flow of fluids.
The main destination is to show, that it is possible to use automatic differentiation in numeric flow simulation for optimization problems, because it is a very important problem, to optimize the efficiency of a propeller on a ship, an airplane or only to reduce the aerodynamic resistance of airplanes and cars. And many other problems are also possible.
Another destination is to explain the topics and the subtopics of this diploma thesis in a way, such that it is possible to understand the main ideas without a too large knowledge about this topics. It is tried to write each chapter in such a way, that it is a closed part and also a part of the full diploma thesis.
We complete the printed version of this diploma thesis with a compact disk, which contains the diploma thesis in a digital version in the PDF format. Furthermore it contains the programs, which I have written by myself, and the programs, which I have changed, and which the licence allows to copy. And also additional graphics are presented on the compact disk.

Chapter 2

Basics of numerical optimization

In this chapter we give an introduction in the basics of numerical optimization. At first, we show the theoretical aspects of optimization. Then, we consider some algorithms for the numerical optimization with information about the derivative and without informations about the derivative. In the third section we concentrate us on the stepsize control by the optimization. In the last section of this chapter we have a look on some simple examples and experiments of numerical optimization.
The fundamentals of this chapter are the book of Alt [Alt02] the OR lecture note of Rieder [Rie95], the numeric I and II lecture notes of Urban [Urb05], [Urb06] and the preprint of Forst and Hoffmann [WF07].

2.1 Theoretical results of optimization

In this section we concentrate us on the theoretical properties, which we need in following. At first we define some basic notions.

Definition 2.1.1 (Local minimum)
For a continuous function $f : \mathbb{R}^n \to \mathbb{R}$ a point $x^* \in \mathbb{R}$ is called **local minimum** *of the function f, if there exists a $R > 0$, such that*

$$f(x^*) \leq f(x) \; for \; all \; x \in B(x^*, R) := \{y \in \mathbb{R}^n \; : \; |x^* - y| \leq R\} \tag{2.1}$$

By this follows:

Definition 2.1.2 (Critical point)
A point $x^* \in \mathbb{R}^n$ is called **critical point** *of the function $f \in C^1$, if*

$$\nabla f(x^*) = 0 \tag{2.2}$$

holds.
The Equation (2.2) is called the **Euler-Lagrange-Equation** *of the function f.*

Definition 2.1.3 (Level Set)
For a given point $x^* \in \mathbb{R}^n$, $D \subset \mathbb{R}^n$, where D is compact, and for a continuous function f the set

$$LS\left(f, f(x^*)\right) := \{x \in \mathbb{R}^n | f(x) \leq f(x^*)\}, \tag{2.3}$$

is called the **level set** *of the function f respect to the point x^*.*

2.2 Numerical optimization algorithms

In this section we consider some algorithms for numerical optimization. We start with an algorithm, which needs no informations about the derivative of the function. But the main aspect is based on algorithms, which use information of the derivative, because this are the algorithms, we will use later for the optimization with AD.

2.2.1 Nelder-Mead-Algorithm

The Nelder-Mead-Algorithm is also known as the Downhill-Simplex-algorithm.
Before we explain this algorithm we need a notion.

Definition 2.2.1 (n-Dimensional simplex)
Let $x^0, \ldots, x^n \in \mathbb{R}^n$ affine independent vectors, i.e. $x^i - x^0,\ i = 1, \ldots, n$ are linear independent. Then the set

$$S := \left\{ \sum_{i=0}^{n} \lambda_i x^i \mid \lambda_i \, i = 0, \ldots, n, \sum_{i=0}^{n} \lambda_i = 1 \right\} \tag{2.4}$$

is called the **n-dimensional simplex** *with the edges $x^0, \ldots, x^n$.*

The idea of this algorithm is: Start with **n-dimensional simplex** as start value. And then replace the worst point by a better one.
Now we show the full Algorithm:

Algorithm 2.2.2 (Nelder-Mead)

choose a start point $x^{(0,0)} \in \mathbb{R}$

set $x^{(0,i)} := x^{(0,0)} + e^i$ the edges of the simplex S_0

do k=0:N

- *calculate $x^{(k,worst)}$ the point of the simplex S_k, which have the biggest function value $x^{(k,worst)} := \arg\left(\max_{i=0,\ldots,n} \{f(x^{(k,i)})\}\right)$*
- *calculate $x^{(k,2.worst)}$ the point of the simplex S_k, which have the second biggest function value $x^{(k,2.worst)} := \arg\left(\max_{\substack{i=0,\ldots,n \\ i \neq worst}} \{f(x^{(k,i)})\}\right)$*
- *calculate $x^{(k,best)}$ the point of the simplex S_k, which have the smallest function value $x^{(k,best)} := \arg\left(\min_{i=0,\ldots,n} \{f(x^{(k,i)})\}\right)$*
- *calculate $x^{(k,centre)}$ the centre of the simplex S^k respect to $x^{(k,worst)}$*
  $x^{(k,centre)} = \frac{1}{n} \sum_{\substack{i=0 \\ i \neq worst}}^{n} x^{(k,i)}$
- *reflect $x^{(k,worst)}$ on $x^{(k,centre)}$ with*
  $x^{(k,reflect)} = x^{(k,centre)} + \gamma(x^{(k,centre)} - x^{(k,worst)})$
- *if ($f(x^{(k,reflect)}) < f(x^{(k,best)})$) then*

$$x^{(k,expansion)} := \alpha x^{(k,reflect)} + (1-\alpha)x^{(k,centre)}$$

$$x^{(k,tmp)} := \begin{cases} x^{(k,expansion)} & if\ f(x^{(k,expansion)}) < f(x^{(k,reflect)}) \\ x^{(k,reflect)} & else \end{cases}$$

$$x^{(k+1,i)} := \begin{cases} x^{(k,i)} & if\ i \neq worst \\ x^{(k,tmp)} & else \end{cases}$$

– *else if* $(f(x^{(k,best)}) \leq f(x^{(k,reflect)}) \leq f(x^{(k,2.worst)}))$ *then*

$$x^{(k+1,i)} := \begin{cases} x^{(k,i)} & if\ i \neq worst \\ x^{(k,reflect)} & else \end{cases}$$

– *else if* $(f(x^{(k,reflect)}) > f(x^{(k,2.worst)}))$ *then*

$$x^{(k,tmp)} := \begin{cases} x^{(k,worst)} & if\ f(x^{(k,worst)}) < f(x^{(k,reflect)}) \\ x^{(k,reflect)} & else \end{cases}$$

$x^{(k,contraction)} := \beta x^{(k,tmp)} + (1-\beta)x^{(k,centre)}$

if$(f(x^{(k,contraction)}) < f(x^{(k,worst)}))then$

$$x^{(k+1,i)} := \begin{cases} x^{(k,i)} & if\ i \neq worst \\ x^{(k,contraction)} & else \end{cases}$$

else

$x^{(k+1,i)} := \frac{1}{2}(x^{(k,i)} + x^{(k,best)})$

end

end

end

At next we give some remarks to the Nelder-Mead-Algorithm:

Remark 2.2.3
Here we explain the sense of the parameters, and give a typical range for this parameters:

– *The parameter* γ *is the reflextion factor,* $\gamma = 1$.

– *The parameter* α *is the expansion factor,* $2 \leq \alpha \leq 3$.

– *The parameter* β *is the contraction factor,* $0.4 \leq \beta \leq 0.6$.

Nelder and Mead (1965) propose the values: $\gamma = 1,\ \alpha = 2\ \beta = 0.5$

Another algorithm without informations about the derivative is for example the Hooke-Jeeves-Algorithm.

2.2.2 Gradient-Algorithm

In this section we orientate us directly on the book from Alt [Alt02]. Now, we consider a method, which is already considered by Cauchy in the year 1847, in a work about methods to solve systems of equations (*Méthode générale pour la résolution des systèmes d'équations simultanées*), for more details see in [Cau47].
In the following we consider a function $f : \mathbb{R}^n \to \mathbb{R}$, which is continuously differentiable. If we have not a **critical point** in the point x according to the Definition 2.1.2, i.e. $\nabla f(x) \neq 0$,

then $-\nabla f$ describes a descent direction. At next, we prove, that the gradient, in the point x is the steepest descent in x. For this we consider the following optimization problem

$$\begin{array}{l} \min\limits_{d\in\mathbb{R}^n} \nabla f(x)^T d \\ \mathbf{SC}^1 \ \|d\| = r, \quad r > 0,\ r \in \mathbb{R}. \end{array} \tag{2.5}$$

The side condition $\|d\| = r$ is necessary, that the problem has an solution, because the minimization is not restricted without this side condition.

Lemma 2.2.4 (The steepest descent)
Let $f : \mathbb{R}^n \to \mathbb{R}$ continuously differentiable in the variable x and $\nabla f(x) \neq 0$. Then

$$\bar{d} := -r \cdot \frac{\nabla f(x)}{\|\nabla f(x)\|} \tag{2.6}$$

is the solution of the optimization problem (2.5), i.e. the negative gradient of the function f in the variable x represents the direction of the steepest descent in the point x.

Proof.
For an arbitrary $d \in \mathbb{R}^n$ holds

$$\nabla f(x)^T d \geq -\|\nabla f(x)\|\|d\|. \tag{2.7}$$

With the Equation (2.7) follows, that for all $d \in \mathbb{R}^n$ with $\|d\| = r$ holds

$$\nabla f(x)^T d \geq -r \cdot \|\nabla f(x)\|. \tag{2.8}$$

And for $\bar{d}$ holds

$$\nabla f(x)^T \bar{d} = -r \cdot \|\nabla f(x)\|. \tag{2.9}$$

by this follows the claim. □

Now we describe in a general form the Gradient-Algorithm.

Algorithm 2.2.5 (Gradient-Algorithm)

choose a start point $x^{(0)} \in \mathbb{R}^n$

do i=0:N

if $(\nabla f(x^{(i)}) = 0)$ *stop*
$d^{(i)} := -\nabla f(x^{(i)})$;
$\sigma^{(i)} := stepsize()$; *//for details look section 2.3*
$x^{(i+1)} := x^{(i)} + \sigma^{(i)} d^{(i)}$

end

Now we present some general remarks to the Gradient-Algorithm.

[1] stands for side condtion

Remark 2.2.6

(i) The gradient is the best descent direction in each local point. But this has not to be the best descent direction in a global setting, for each function.

(ii) In general, we only know, that the gradient algorithm finds a local minimum. But, if we have a convex function, so this one is also the global minimum.

(iii) The choice of a good stepsize is not a trivial problem. Some ideas to choose the stepsize, we will show in the Section 2.3.

2.2.3 Conjugated-Gradients

Here we consider a special version of the steepest descent algorithm. The idea is to compute in each step the steepest descent direction with the gradient, and then to orthogonalize this with respect to the previous directions. The advantage of this method is to eliminate a *shiver* of the descent direction.

Algorithm 2.2.7 (Conjugated-Gradients)

choose a start point $x^{(0)} \in \mathbb{R}^n$

set $\nabla f_0 \quad := \nabla f(x^{(0)})$

set $d^{(0)} \quad := -\nabla f_0$

do i=0:N

if $(\nabla f_i = 0)$ *stop;*

$\sigma^{(i)} := stepsize();$ *//for details look section 2.3*

$x^{(i+1)} := x^{(i)} + \sigma^{(i)} d^{(i)};$

$\nabla f_{i+1} := \nabla f(x^{(i+1)});$

$$\gamma^{(i+1)} := \begin{cases} 0 & if(i+1 \equiv 0 \, mod \, n) \\ \frac{\langle \nabla f_{i+1}, \nabla f_{i+1} \rangle}{\langle \nabla f_i, \nabla f_i \rangle} & else \end{cases};$$

$d^{(i+1)} := -\nabla f_{i+1} + \gamma^{(i+1)} d^i;$

end

2.2.4 Newton-like algorithms

In this subsection we consider the Newton-Algorithm and the Damped-Newton-Algorithm. The idea of the Newton-Algorithm is: consider a linear approximation of the function in order to find the null of the function.
Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a twice continuously differentiable function, then we can use the Taylor approximation, by this follows:

$$f(x) = f(x^{(i)}) + \nabla f(x^{(i)})(x - x^{(i)}) + \mathcal{O}(\|x - x^{(i)}\|_2^2). \tag{2.10}$$

If the rang of matrix $\nabla f(x^{(i)})$ is full, then the Equation (2.10) implies, that the null of this linear approximation is given by:

$$x^{(i+1)} = x^{(i)} - (\nabla f(x^{(i)}))^{-1} f(x^{(i)}). \tag{2.11}$$

It is clear, that the inverse matrix in the Equation (2.11) is only for the theoretical view. The Newton-Algorithm is given by:

Algorithm 2.2.8 (Newton-Algorithm)

choose a start point $x^{(0)} \in \mathbb{R}^n$, tol

do i=0:N

$f_i \quad := f(x^{(i)});$
$\nabla f_i \quad := \nabla f(x^{(i)});$
if $(\nabla f_i \leq tol)$ *stop;*
compute d_i *with* $\nabla f_i \cdot d_i = -f_i$
$x^{(i+1)} := x^{(i)} + d^{(i)};$

end

Now, we consider the convergence order of the Newton-Algorithm.

Remark 2.2.9 (Condition)
Let $\Omega \subset \mathbb{R}^n$ be open and convex. The function $f : \Omega \to \mathbb{R}^n$ is a continuously differentiable function, for which the following conditions hold:

(i) $\nabla f(x)$ *has full rang for all* $x \in \Omega$,

(ii) $\|(\nabla f(x))^{-1}\| \leq \beta$ *for all* $x \in \Omega$,

(iii) $\nabla f(x)$ *is Lipschitz-continuous on* Ω *with the constant* γ,

(iv) there exists a point $x^\star \in \Omega$, *such that* $f(x^\star) = 0$.

Lemma 2.2.10
By the conditions from the Remark 2.2.9 follows:

$$\|f(x) - f(y) - \nabla f(y)(x-y)\| \leq \frac{\gamma}{2}\|x-y\|^2, \tag{2.12}$$

for all $x, y \in \Omega$.

Proof.
Let $\phi := f(y + t(x-y))$ on $t \in [0,1]$, this implies, that the function $\phi(t)$ is continuously differentiable on the interval $[0,1]$. The derivative of the function ϕ is given by $\phi'(t) = \nabla f(y + t(x-y)) \cdot (x-y)$, this follows by the chain rule.
This implies

$$\begin{aligned} \|\phi'(t) - \phi'(0)\| &= \|[\nabla f(y + t(x-y)) - \nabla f(y)](x-y)\| \\ &\leq \|[\nabla f(y + t(x-y)) - \nabla f(y)]\| \cdot \|(x-y)\| \\ &\overset{2.2.9(iii)}{\leq} \gamma t \|(x-y)\|^2. \end{aligned}$$

On the other hand holds:

$$f(x) - f(y) - \nabla f(y) \cdot (x - y) = \phi(1) - \phi(0) - \phi'(0) = \int_0^1 (\phi'(t) - \phi'(0))dt. \quad (2.13)$$

And this implies:

$$\begin{aligned} \|f(x) - f(y) - \nabla f(y) \cdot (x - y)\| &= \int_0^1 \|\phi'(t) - \phi'(0)\|dt \\ &\leq \gamma\|x - y\|^2 \int_0^1 tdt \\ &= \frac{\gamma}{2}\|x - y\|^2. \end{aligned}$$

□

Theorem 2.2.11
Additionally to the conditions from the Remark 2.2.9 there must hold, that a start point $x^{(0)} \in \Omega$ is given, such that

$$\|x^\star - x^{(0)}\| \leq \omega \quad (2.14)$$

and

$$\frac{1}{2}(\beta\gamma\omega) < 1, \quad (2.15)$$

holds.
Than the sequence $\{x^{(i)}\}_{i=0}^\infty$, which is defined by the Newton-Algorithm 2.2.8, is in the set

$$B_w(x^\star) := \{x \in \mathbb{R}^n : \|x^\star - x\| < \omega\},$$

x_i converge to x^* and the convergence order is two.

Proof.

$$\begin{aligned} x^{(i+1)} - x^\star &= x^{(i)} - x^\star - (\nabla f(x^{(i)}))^{-1}(f(x^{(i)}) - \underbrace{f(x^\star)}_{=0}) \\ &= -(\nabla f(x^{(i)}))^{-1}(f(x^{(i)}) - f(x^\star) - \nabla f(x^{(i)})(x^{(i)} - x^\star)) \end{aligned}$$

By the Lemma 2.2.10 follows:

$$\begin{aligned} \|x^{(i+1)} - x^\star\| &\leq \underbrace{\|(\nabla f(x^{(i)}))^{-1}\|}_{\overset{2.2.9.(ii)}{\leq}\beta} \underbrace{\|(f(x^{(i)}) - f(x^\star) - \nabla f(x^{(i)})(x^{(i)} - x^\star))\|}_{\overset{(2.12)}{\leq}\frac{\gamma}{2}\|x-y\|^2} \\ &\leq \frac{\beta\gamma}{2}\|x^{(i)} - x^\star\|^2. \end{aligned} \quad (2.16)$$

This implies the convergence order two.
At last, we show, that for the sequence $\{x^{(i)}\}_{i=0}^\infty$ holds: $\{x^{(i)}\}_{i=0}^\infty \subset B_\omega(x^\star)$.
For $i = 0$ it is clear by the Equation (2.14), and inductively follows:

$$\|x^{(i+1)} - x^\star\| \leq \frac{\beta\gamma}{2}\|x^{(0)} - x^\star\| < \frac{\beta\gamma}{2}\omega^2 \overset{(2.15)}{<} \omega.$$

□

Remark 2.2.12

- *If the function f is twice continuously differentiable, this implies directly the condition of the Remark 2.2.9 (iii).*
- *The constants β and γ come from the problem. This implies, that we need a good start point.*
- *By the conditions from the Remark 2.2.9 it is possible to prove, that there exists only one null $x^\star$ on $B_{\frac{2}{\beta\gamma}}$.*

It is clear, that the classical Newton-Algorithm has one big drawback, it converges only in local domain. And it is possible, that this local domain is very small. Now we consider a modification of the Newton-Algorithm, the Damped-Newton-Algorithm in order to eliminate this drawback.

Algorithm 2.2.13 (Damped-Newton-Algorithm)

choose a start point $x^{(0)} \in \mathbb{R}^n$, tol

do i=0:N

$f_i \quad := f(x^{(i)});$
$\nabla f_i \quad := \nabla f(x^{(i)});$
if $(\|\nabla f_i\| \leq tol)$ *stop;*
compute d_i *with* $\nabla f_i \cdot d_i = -f_i$
set $\lambda := 1$
set $x := x_i + \lambda d_i$ *(2.2.13)*
if $(\|\nabla f_i^{-1} f(x)\| \leq 1 - \frac{\lambda}{2}\|d_i\|)$ *then*
    $x_{i+1} = x;$
else
    $\lambda = \frac{\lambda}{2};$
    if $(\lambda < 2^{-10})$ *stop;*
    goto (2.2.13)
end

end

Remark 2.2.14
It is clear, that the Damped-Newton-Algorithm converges in the general not as fast as the Newton-Algorithm.

Remark 2.2.15 (Realization)
Do not compute the Jacobian-Matrix in each step, compute it only in each j-th step. It is recommanded to use a correction method. More details are given in the book from Deufelhard and Bornemann [PD95].
We can also compute the Jacobian-Matrix approximatively with finite differences.

2.3 Stepsize control

In this section, we analyze the optimal stepsize for a given descent direction. At first we consider some theoretical aspects, then we show, that there exists an exact optimal stepsize. In the following way, we issue two algorithms to approximate a *good* stepsize.
Now we give some necessary conditions for the choice of the stepsize σ. The first one are the **Goldstein conditions**.

Definition 2.3.1 (Goldstein conditions)
Let $f \in C^1(D)$, $D \subset \mathbb{R}^n$, $x \in D$ and $d \in \mathbb{R}^n$ with, $\langle \nabla f(x), d\rangle < 0$, i.e. d is a given descent direction. Then the stepsize σ satisfies the **Goldstein conditions***, if the inequalities*

$$f(x + \sigma d) \leq f(x) + c_1 \sigma \langle \nabla f(x), d\rangle \tag{2.17}$$

and

$$f(x + \sigma d) \geq f(x) + c_2 \sigma \langle \nabla f(x), d\rangle \tag{2.18}$$

hold. For $c_1 \in (0, \frac{1}{2}]$ and $c_2 \in (c_1, 1)$.

Similar conditions are the **Armijo conditions**.

Definition 2.3.2 (Armijo conditions)
Let $f \in C^1(D)$, $D \subset \mathbb{R}^n$, $x \in D$ and $d \in \mathbb{R}^n$ where, $\langle \nabla f(x), d\rangle < 0$, i.e. d is a given descent direction. Furthermore are $0 < c < \frac{1}{2}$, $q > 1$ and $0 < \bar{\sigma} \leq 1$ given. Then the stepsize σ satisfies the **Armijo conditions***, if σ is the maximum of the sequence $\bar{\sigma}_j := \left\{\bar{\sigma} q^{-j}\right\}$, which fullfiles the inequality*

$$f(x + \bar{\sigma}_j d) \leq f(x) + c\, \bar{\sigma}_j \langle \nabla f(x), d\rangle. \tag{2.19}$$

The following lemma is out of the book from [Alt02] and it shows us, that there always exists a stepsize with some properties, which we need to show a warranty of convergence.

Lemma 2.3.3
Let f be a continuously differentiable function on a convex superset of the level set $LS(f, f(x_0))$ and the gradient of the function f is on the level set Lipschitz continuous.
Furthermore let a point $x \in LS(f, f(x_0))$, a descent direction $d \in \mathbb{R}^n$, i.e. $\langle \nabla f(x), d\rangle < 0$ holds, and a constant $\delta \in (0, 1)$ is given.
Then there exist a $\tau = \tau(x, d, \delta)$ with the following properties:

(i) $f(x + \sigma d) < f(x) + \delta\sigma\langle \nabla f(x), d\rangle$ *for all* $\sigma \in (0, \tau)$

(ii) $f(x + \tau d) = f(x) + \delta\tau\langle \nabla f(x), d\rangle$

(iii) $\tau \geq \rho := -\frac{2(1-\delta)}{L} \frac{\langle \nabla f(x), d\rangle}{\|d\|^2}$

(iv) $\frac{d}{d\sigma} f(x + \sigma d) = \langle \nabla f(x + \sigma d), d\rangle < \delta\langle \nabla f(x), d\rangle$ *for all* $\rho \in (0, \tau)$

Proof.

(i) Due to the continuous differentiability of the function f on the convex superset of the level set there holds

$$\langle \nabla f(x), d\rangle = \lim_{\sigma \to 0} \frac{f(x + \sigma d) - f(x)}{\sigma},$$

and with $\delta \in (0,1)$, $\langle \nabla f(x), d\rangle < 0$ follows $\langle \nabla f(x), d\rangle < \delta\langle \nabla f(x), d\rangle$. So there exists a $\bar{\sigma} > 0$ with the following property

$$\frac{f(x+\sigma d)-f(x)}{\sigma} < \delta\langle \nabla f(x), d\rangle, \text{ for all } \sigma \in (0, \bar{\sigma}),$$

i.e.

$$f(x+\sigma d) - f(x) < \delta\sigma\langle \nabla f(x), d\rangle, \text{ for all } \sigma \in (0, \bar{\sigma}).$$

This implies, that the set $T := \{\tau | (i)\ holds\}$ is not the empty set.

(ii) If $\tau \in T$ and by (i) $f(x+\sigma d) < f(x)$ holds, i.e. $(x+\sigma d) \in LS(f, f(x_0))$ for all $\sigma \in [0,\tau]$. Because of the compactness of the level set $LS(f, f(x_0))$ follows, that the function f is bounded on this set.
But $\delta\tau\langle \nabla f(x), d\rangle \to -\infty$ for $\tau \to \infty$ implies, that the set T is upwards bounded, and for $\tau := \sup(\arg T)$ holds (ii).

(iii) By (i) and (ii) holds, $f(x+\sigma d) \leq f(x)$ and $(x+\sigma d) \in LS(f, f(x_0))$ for all $\sigma \in [0,\tau]$. This implies, that the function

$$\phi : [0,\tau] \to \mathbb{R}, \phi(s) := f(x+sd),$$

is differentiable with

$$\phi'(s) = \langle \nabla f(x+sd), d\rangle.$$

From this follows

$$f(x+\tau) - f(x) = \phi(\tau) - \phi(0) = \int_0^\tau \phi'(s)\, ds = \int_0^\tau \langle \nabla f(x+sd), d\rangle ds,$$

and we define us

$$A := f(x+\tau d) - f(x) - \tau\langle \nabla f(x), d\rangle = \int_0^\tau [\nabla f(x+sd) - \nabla f(x)]^T d\, ds.$$

With (ii) holds

$$A = -(1-\delta)\tau\langle \nabla f(x), d\rangle, \tag{2.20}$$

and the Lipschitz continuity of the gradient of the function f on the level set implies

$$A \leq \int_0^\tau Ls\|d\|\|d\|\, ds = \frac{1}{2}\tau^2 L\|d\|^2, \tag{2.21}$$

if L is the Lipschitz constant of $\nabla f(x)$ for all $x \in LS(f, f(x_0))$. (2.20) and (2.21) show the statement (iii).

(iv) By the Lipschitz continuous and (iii) holds

$$\begin{aligned}\langle \nabla f(x+\sigma d), d\rangle &= \langle \nabla f(x), d\rangle + [\nabla f(x+\sigma d) - \nabla f(x)]^T d \\ &< \langle \nabla f(x), d\rangle + \tfrac{\rho}{2} L\|d\|^2 \overset{(iii)}{=} \delta\langle \nabla f(x), d\rangle\end{aligned}$$

for all $\sigma \in (0, \rho/2)$. This implies (iv).

□

The following lemma shows, that the Goldstein conditions give us a warranty for a minimal stepsize in each point of the level set depents of the gradient and the descent direction.

Lemma 2.3.4
Let f be a function, such that the Lemma 2.3.3 holds, let $d \in \mathbb{R}^n$ be a descent direction. Furthermore the Definition 2.3.1 holds. Then there exists a stepsize σ such that,

$$\sigma \geq -c\frac{\langle \nabla f(x), d\rangle}{\|d\|^2}, \tag{2.22}$$

holds, for each $x \in LS(f, f(x_0))$ and a constant $c > 0$; c is independent on x, if for the constants c_1, c_2 in the Definition 2.3.1 holds $c_1 \in (0, \frac{1}{2}]$, $c_2 > c_1 1$.

Proof.
The Lemma 2.3.3 (ii) says:

$$f(x + \tau d) = f(x) + \delta\tau\langle \nabla f(x), d\rangle$$

Now we choose the constant $\delta \in (0, 1)$, then follows by: $c_1 = \delta\tilde{c}_1$ and $c_2 = \delta\tilde{c}_2$, where $\tilde{c}_1 \leq \frac{1}{2\delta}$ and $\tilde{c}_2 \geq 1$, that:

$$f(x + \tau d) \leq f(x) + \underbrace{(\delta\tilde{c}_1)}_{=c_1}\tau\langle \nabla f(x), d\rangle, \tag{2.23}$$

$$f(x + \tau d) \geq f(x) + \underbrace{(\delta\tilde{c}_2)}_{=c_2}\tau\langle \nabla f(x), d\rangle. \tag{2.24}$$

This implies, that τ fulfill the Goldstein conditions, and τ is an acceptable stepsize. The aim follows now directly with the Lemma 2.3.3 (iii). □

The next lemma shows a similar claim for the Armijo conditions from the Definition 2.3.2.

Lemma 2.3.5
Let f be a function, such that the Lemma 2.3.3 holds, let $d \in \mathbb{R}^n$ be a descent direction. Furthermore holds the Definition 2.3.2. Then there exists a stepsize σ such that,

$$\sigma \geq -\bar{c}\frac{\langle \nabla f(x), d\rangle}{\|d\|^2}, \tag{2.25}$$

holds, for each $x \in LS(f, f(x_0))$ and a constant $\bar{c} > 0$, $\bar{c}$ is independent on x.

Proof.
The Lemma 2.3.3 implies, that for all $\tilde{\sigma} \in (0, \tau)$ holds

$$f(x + \tilde{\sigma}d) - f(x) < \delta\tilde{\sigma}\langle \nabla f(x), d\rangle. \tag{2.26}$$

The stepsize σ of the Definition 2.3.2 is defined as $\sigma := \max\{\sigma_j | for\ \sigma_j\ holds\ (2.19)\}$ and $\sigma_j := \bar{\sigma}q^{-j}$, q, $\bar{\sigma}$ is defined as in the Definition 2.3.2 by this follows, that $\sigma_j < \sigma_{j-1} < \ldots \sigma_1 < \sigma_0 = \bar{\sigma}$ and for the stepsize σ_{j-1} holds the (2.19) **not**, if $\sigma = \sigma_j$. With (2.26) follows, that

$$\sigma_j \leq \tau < \sigma_{j-1},$$

and this implies

$$\sigma = \sigma_j \geq \tau\frac{1}{q}.$$

The claim follows now with the Lemma 2.3.3 (iii). □

At next we give in the following remark a conclusion of the last definitions and lemmas.

Remark 2.3.6

- *The constants c_1, c_2 in the Definition 2.3.1 of the Goldstein conditions are often chosen as $c_1 = \frac{1}{4}$ and $c_2 = \frac{3}{4}$.*
- *The constant c in the Definition 2.3.2 of the Armijo conditions is often chosen as $c = \frac{1}{4}$.*
- *With the Lemma 2.3.4 and 2.3.5 it is clear, that the Definitions 2.3.1 and 2.3.2 have the same conclusion. Both definitions ensure, that we have a monotone descent for the function f, if we choose the stepsize such that Definition 2.3.1 or 2.3.2 holds. On the other-side, we get a warranty, that the stepize is not to small, so that the function f can converge to a fix point.*

Now we show a theorem, which give us a warranty, that we find a local minimum, if we choose the stepsize according as the Definition 2.3.1 or 2.3.2 and the descent direction d in the point $x \in LS(f, f(x_0))$ satisfy

$$-\langle \nabla f(x), d\rangle > c\|d\|\|\nabla f(x)\|, \tag{2.27}$$

for a constant $c > 0$, which is independent of x and d.

Theorem 2.3.7
Let f be a continuously differentiable function on a convex super set of the level set $LS\,(f, f(x_0))$, let the gradient of f be Lipschitz-continuous on the level set. And the function f has one or more critical points on the level set. Furthermore let $d_k \in \mathbb{R}^n$ be a sequence of descent directions, which satisfys the Equation (2.27), in the point $x_k \in Ls(f, f(x_0))$. And the stepsize σ_k for the descent direction d_k is conform with the Goldstein conditions of Definition 2.3.1 or the Armijo conditions of Definition 2.3.2.
Then holds: the sequence $\{x_k\}$ converges to a critical point, if x_k is defined as $x_{k+1} := x_k + \sigma_k d_k$, and x_0 is the start point.

Proof.
With Definition 2.1.2 follows, the point x_k is a critical point, if

$$\nabla f(x_k) = 0, \tag{2.28}$$

holds. The Equation (2.27) implies, that

$$\frac{\langle \nabla f(x_k), d_k\rangle}{\|d_k\|} \to 0 \; for \; k \to \infty, \tag{2.29}$$

fulfill the Equation (2.28). If the Definition 2.3.1 or 2.3.2 holds, then there exists a constant $\bar{c} > 0$, such that

$$f(x_k + \sigma_k d_k) - f(x_k) \leq \bar{c}\sigma_k \langle \nabla f(x_k), d_k\rangle \; for \; all \; k \in \mathbb{N}_0, \tag{2.30}$$

holds. The compactness of the level set implies, that the function f is bounded for all $x \in LS(f, f(x_0))$. And this combined with $f(x_{k+1}) < f(x_k)$ implies, that

$$f(x_k + \sigma_k d_k) - f(x_k) \to 0 \; for \; k \to \infty. \tag{2.31}$$

With (2.30) and (2.31) follow, that

$$\sigma_k \langle \nabla f(x_k), d_k \rangle \to 0 \; for \; k \to \infty. \tag{2.32}$$

The Lemma 2.3.4 or Lemma 2.3.5 combined with (2.29) and (2.32) give us:

$$\begin{aligned} \sigma_k \langle \nabla f(x_k), d_k \rangle &\leq -\tilde{c} \frac{\langle \nabla f(x_k), d_k \rangle}{\|d_k\|^2} \langle \nabla f(x_k), d_k \rangle \\ &= \tilde{c} \left(\frac{\langle \nabla f(x_k), d_k \rangle}{\|d_k\|} \right)^2 \to 0 \; for \; k \to \infty, \end{aligned} \tag{2.33}$$

The Equation (2.33) shows the claim. □

Remark 2.3.8

- *A stepsize σ is called* **efficient stepsize***, if*

$$f(x + \sigma d) \leq f(x) + c \left(\frac{\langle \nabla f(x_k), d_k \rangle}{\|d_k\|} \right)^2,$$

  for a constant c, which is independent of x and d, holds.

- *Lemma 2.3.4 implies, that Definition 2.3.1 defines a* **efficient stepsize**.

- *Lemma 2.3.5 implies, that Definition 2.3.2 defines a* **efficient stepsize**.

We have shown some theoretical aspects for the choice of the stepize. The problem is, that we have not an efficient algorithm to compute an efficient stepsize, we try to correct this disadvantage.
At first we specify the exact stepize, but it is clear, that this is only in special cases possible to compute.

2.3.1 Exact stepsize control

Here we specify the exact stepsize σ_e for a given descent direction d, a point $x \in LS(f, f(x_0))$ and a function f, which is on a convex superset of the level set continuously differentiable and has one or more critical points on the level set.
We get the optimal stepsize, if we solve the one dimensional optimization problem

$$\min_{s \geq 0} \varphi(s) = f(x + sd). \tag{2.34}$$

With (2.34) follows

$$\varphi'(s) = \frac{d}{ds} f(x + sd) = \langle \nabla f(x + sd), d \rangle \begin{cases} = 0 & for \quad s = \sigma_e \\ < 0 & for \; all \quad s \in [0, \sigma_e) \end{cases}. \tag{2.35}$$

The stepsize σ_e is a local minimum of (2.34). If we further know, that the function f is convex, then follows, that σ_e is also the global solution.
It is clear, that we can only compute σ_e in an analytical way in some special cases, e.g., if f is a linear function. Otherwise we must solve the Equation (2.34) by numerical methods.
Now we show an estimate for the exact stepsize.

Theorem 2.3.9
Let f be a continuously differentiable function on a convex superset of the level set, the gradient of the function f is Lipschitz-continuous on the level set, and f has one or more critical points on the level set. Furthermore there is a point $x \in LS(f, f(x_0)$ and a descent direction $d \in \mathbb{R}^n$ given.
Then for the exact stepsize σ_e

$$\sigma_e \geq \tilde{\sigma} := \frac{-\langle \nabla f(x), d\rangle}{L\|d\|^2} \tag{2.36}$$

holds and

$$f(x + \sigma_e d) \leq f(x) - \frac{1}{2L}\left(\frac{\langle \nabla f(x), d\rangle}{\|d\|}\right)^2 = f(x) - \frac{1}{2}\tilde{\sigma}\langle \nabla f(x), d\rangle \tag{2.37}$$

Proof.
Due to $f(x + \sigma_e d) \leq f(x)$ the point $(x + \sigma_e d)$ is element of $LS(f, f(x_0))$. And by the Lipschitz-continuous of the gradient from the function f follows

$$\begin{aligned} 0 = \langle \nabla f(x + \sigma_e d), d\rangle &= \langle \nabla f(x), d\rangle \langle \nabla f(x + \sigma_e d) - f(x), d\rangle \\ &\leq \langle \nabla f(x), d\rangle + \sigma_e L\|d\|^2 \end{aligned} \tag{2.38}$$

and this implies (2.36). Analogeous to poof of Lemma 2.7 (iii) follows

$$f(x + \tilde{\sigma} d) \leq f(x) + \tilde{\sigma}\langle \nabla f(x), d\rangle + \frac{1}{2}L\tilde{\sigma}\|d\|^2. \tag{2.39}$$

The definition of of $\tilde{\sigma}$ implies

$$f(x + \tilde{\sigma}) \leq f(x) + \tilde{\sigma}\left(\langle \nabla f(x), d\rangle + \frac{1}{2}L\tilde{\sigma}\|d\|^2\right) = f(x) + \frac{1}{2}\tilde{\sigma}\langle \nabla f(x), d\rangle. \tag{2.40}$$

Because of $\sigma_e \geq \tilde{\sigma}$ holds $f(x + \sigma_e d) = \varphi(\sigma_e) \leq \varphi(\tilde{\sigma}) = f(x + \tilde{\sigma})$ and this shows (2.37). □

In the next section we consider the Armijo method, to compute the stepsize.

2.3.2 Stepsize control of Armijo

Here we consider a first simple algorithm to compute an *efficient* stepsize.

Algorithm 2.3.10 (Armijo method)
choose the constants $0 < \delta < 1$, $0 < \gamma$ and $0 < \beta_1 \leq \beta_2 < 1$ fixed for all x and d
set $\sigma_0 \geq -\gamma \frac{\langle \nabla f(x), d\rangle}{\|d\|^2}$
do i=0:N
  if $(f(x + \sigma_i \leq f(x) + \delta\sigma_i\langle \nabla f(x), d\rangle))$ *then*
    $\sigma := \sigma_i$;
    stop;
  end
  choose $\sigma_{i+1} \in [\beta_1\sigma_i, \beta_2\sigma_i]$;
end

Remark 2.3.11
The choice of σ_i in the i-th iteration, is not specified in the general Algorithm 2.3.10. Also the constants are not specified. Here is one typical choice:

$\delta := 0.01$

$\gamma := 10^{-4}$ *or set* $\sigma_0 := 1$ *(fix)*

$\beta_1 = \beta_2 := \frac{1}{2}$ *this implies also a choice of* σ_i *in the i-th step.*

Remark 2.3.12
A big disadvantage is, that the **Armijo method** *only reduces the stepsize.*

We try to eliminate this disadvantage in the next subsection with the **Powell method**.

2.3.3 Stepsize control of Powell

Another method to choose a *efficient* stepsize is the **Powell method**. To describe this method, we need two help functions to compute the *efficiency* of the actual stepsize. One big difference to the **Armijo method** is, that the start interval is not only dependent on the start stepsize and the help functions. Another difference is, that the **Powell method** allows to enlarge also the stepsize.

Remark 2.3.13
Here we give the definitions of the two help-functions for the Powell method.

$$G_1(\sigma) := \begin{cases} 1, & if\ \sigma = 0, \\ \frac{f(x+\sigma d)^T - f(x)}{\sigma \nabla f(x)^T d} & else, \end{cases} \tag{2.41}$$

$$G_2(\sigma) := \frac{\nabla f(x+\sigma d)^T d}{\nabla f(x)^T d}. \tag{2.42}$$

Now we can consider the full algorithm.

Algorithm 2.3.14 (Powell method)

choose the constants $0 < \delta < \beta < 1$

set: σ_0

if $(G_1(\sigma_0) \geq \delta$ *and* $G_2(\sigma_0) \leq \beta)$ *then*

$\sigma := \sigma_0;$

stop;

else if $(G_1(\sigma_0) \geq \delta$ *and* $G_2(\sigma_0) > \beta)$ *then* *//i.e.* $\sigma_0 \in I_1 \Rightarrow \sigma > \sigma_0$

$a_0 := \sigma_0;$

$\ell := \min_{\ell \in \mathbb{N}_0} (\ell : \ G_1(2^\ell \sigma_0) < \delta);$

$b_0 := 2^\ell \sigma_0;$ *//i.e.* $b_0 \in I_3$

else if $(G_1(\sigma_0) < \delta)$ *then* *//i.e.* $\sigma_0 \in I_3 \Rightarrow \sigma < \sigma_0$

$\ell \quad := \min_{\ell \in \mathbb{N}_0} (\ell : \ G_1(2^{-\ell}\sigma_0) \geq \delta \ and \ G_2(2^{-\ell}\sigma_0) > \beta);$

$a_0 \ := 2^{-\ell}\sigma_0;$ *//i.e.* $a_0 \in I_1$

$b_0 \ := \sigma_0;$

end

do i=0:N

$\sigma_i := \frac{1}{2}(a_i + b_i);$

if $(G_1(\sigma_i) \geq \delta \ and \ G_2(\sigma_i) \leq \beta)$ *then*

$\sigma \quad := \sigma_i;$

stop;

else if $(G_1(\sigma_i) \geq \delta \ and \ G_2(\sigma_i) > \beta)$ *then* *//i.e.* $\sigma_i \in I_1 \ \Rightarrow \sigma > \sigma_0$

$a_{i+1} := \sigma_i;$

$b_{i+1} := b_i;$

else if $(G_1(\sigma_i) < \delta)$ *then* *//i.e.* $\sigma_i \in I_3 \ \Rightarrow \sigma < \sigma_0$

$a_{i+1} := a_i;$

$b_{i+1} := \sigma_i;$

end

end

Remark 2.3.15
The choice of the constants in the algorithm is not specified. In this remark we give a typical choice for the constants:

$\sigma_0 = 1.$

$\delta = 0.1$

$\beta = 0.9$

In the next section we presentate some numerical tests with different approaches of optimization algorithms for three test functions.

2.4 Comparison of different optimization algorithms

In this section we apply different approaches of optimization algorithms, on three test functions. The reason for this numerical experiments is, that we get a feeling for the different algorithms. On each test function we apply the following algorithms:

- Steepest-Descent-Algorithm with Armijo-stepsize-control (SDA-ASC)
- Steepest-Descent-Algorithm with Powell-stepsize-control (SDA-PSC)
- Conjungated-Gradient-Algorithm with Armijo-stepsize-control (CGA-ASC)
- Conjungated-Gradient-Algorithm with Powell-stepsize-control (CGA-PSC)

- Newton-Algorithm (NA)
- Damped-Newton-Algorithm (DNA)
- Nelder-Mead-Algorithm (NMA)

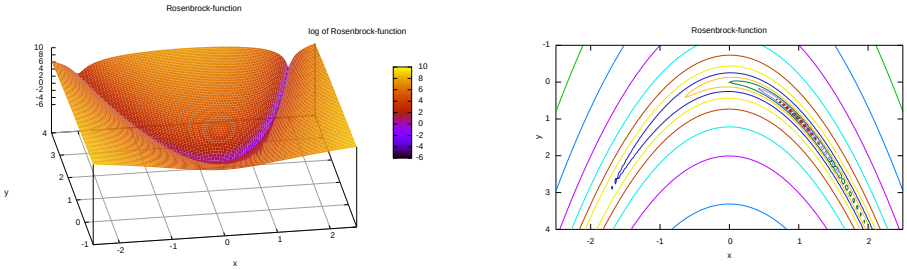
The runtime comparison was done on a Pentium 4 with one gigabyte main memory and the operating system Linux with the distribution ubuntu. The time values are mean values, of several computations. This is the configuration for each of the next three test functions.

2.4.1 The Rosenbrock-function

The Rosenbrock-function is given by the following formula:

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{2.43}$$

It is a polynom of order 4, which has one global minimum at the point (1., 1.) where the function value is zero.
The start point for the iteration is $x = (-1.9, 2)$.



a. logarithm of the Rosenbrock-function b. Isolines of the Rosenbrock-function

Figure 2.1: Rosenbrock-function

The Image 2.1 shows us the characteristics of the Rosenbrock-function. With the characteristics it is clear, that the methods, which are based on the gradient as descent direction, are not the best choice. This is also shown in the Table 2.1 and 2.2.

Algorithm	Iterations	function value	point	time	factor
SDA-ASC	2361	9.99631E-006	1.00316, 1.00634	4.4938E-003	2665.3
SDA-PSC	391	8.87032E-006	1.00298, 1.00597	3.1071E-003	1842.8
CGA-ASC	149	9.69558E-006	0.99696, 0.99386	2.5960E-004	153.9
CGA-PSC	417	6.09559E-006	1.00244, 1.00485	1.6882E-003	1001.3
NA	5	4.73137E-018	1.00000, 1.00000	1.6860E-006	1.0
DNA	27	6.00622E-007	0.99927, 0.99851	1.9157E-005	11.4
NMA	72	7.20092E-006	0.99809, 0.99600	2.1280E-005	12.6

Table 2.1: Rosenbrock-function with an error $< 10^{-5}$.

Algorithm	Iterations	function value	point	time	factor
SDA-ASC	9476	9.99212E-011	1.00001, 1.00002	1.9660E-002	11660.7
SDA-PSC	6850	9.99500E-011	1.00001, 1.00002	5.6240E-002	33357.1
CGA-ASC	411	9.63266E-011	0.99999, 0.99998	6.6380E-004	393.7
CGA-PSC	2007	9.88060E-011	1.00001, 1.00002	7.0180E-003	4162.5
NA	5	4.73137E-018	1.00000, 1.00000	1.6860E-006	1.0
DNA	28	4.20201E-011	1.00000, 0.99999	2.0150E-005	12.0
NMA	93	2.01753E-012	1.00000, 1.00000	2.6260E-005	15.6

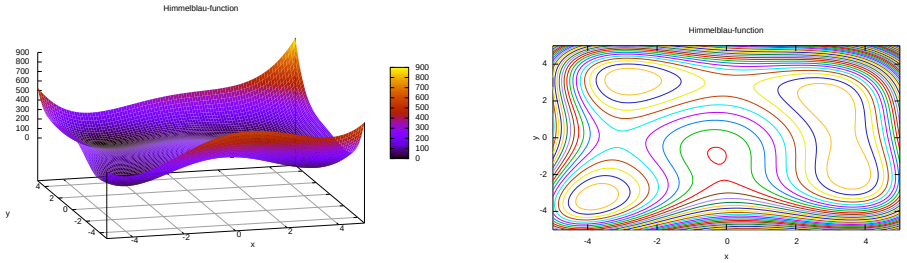
Table 2.2: Rosenbrock-function with an error $< 10^{-10}$.

2.4.2 The Himmelblau-function

The Himmelblau-function is given by the following formula:

$$f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2. \tag{2.44}$$

The Himmelblau-function is a polynom of order 4, with one local maximum at the point (-0.270845, -0.923039) and four local minimums, which are also global minimums and the function value on this points are zero.
The start point for the iteration is $x = (1.5, 4)$.



a. Himmelblau-function b. Isolines of the Himmelblau-function

Figure 2.2: Himmelblau-function

The Image 2.2 shows us the characteristics of the Himmelblau-function. By the characteristics

Algorithm	Iterations	function value	point	time	factor
SDA-ASC	20	8.62406E-006	3.58483, -1.84805	3.0730E-005	22.9
SDA-PSC	10	9.21111E-006	3.00028, 2.00047	7.8630E-005	58.6
CGA-ASC	11	8.63855E-006	3.58416, -1.84748	1.6150E-005	12.0
CGA-PSC	7	1.19109E-006	2.99991, 1.99982	6.0670E-005	45.2
NA	8	1.10505E-007	-2.80518, 3.13132	2.8450E-006	2.1
DNA	5	4.43162E-012	-2.80512, 3.13131	4.0900E-006	3.0
NMA	2	0.	3.00000, 2.00000	1.3420E-006	1.0

Table 2.3: Himmelblau-function with an error $< 10^{-5}$.

Algorithm	Iterations	function value	point	time	factor
SDA-ASC	34	4.39531E-011	3.58443, -1.84813	5.1850E-005	38.6
SDA-PSC	18	1.84065E-011	3.00000, 2.00000	1.2789E-004	95.3
CGA-ASC	21	1.96806E-011	3.58442, -1.84812	2.9690E-005	22.1
CGA-PSC	11	5.25415E-011	3.00000, 2.00000	9.0250E-005	67.3
NA	9	1.02448E-016	-2.80511, 3.13131	3.1460E-006	2.3
DNA	5	4.43162E-012	-2.80512, 3.13131	4.0900E-006	3.0
NMA	2	0.	3.00000, 2.00000	1.3420E-006	1.0

Table 2.4: Himmelblau-function with an error $< 10^{-10}$.

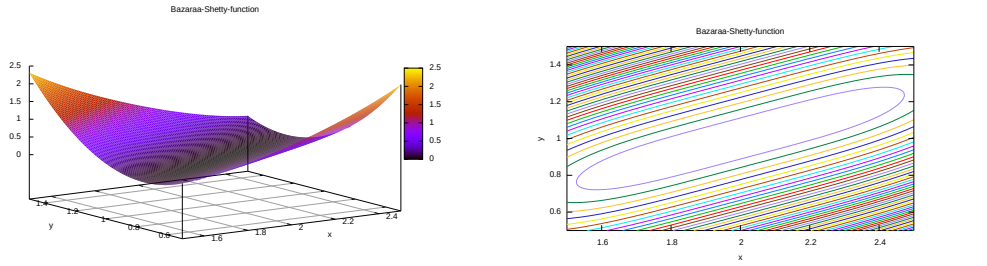
it is clear, that we have not a convex function. By the theory follows, that we can not ensure, that we find the global minimum, or that the solution is unique. This problem is shown in the Table 2.3 and 2.4, if we consider the points of the different methods. So we have found three of the four minimums with the same start point, only by using different algorithms.

2.4.3 The Bazaraa-Shetty-function

The Bazaraa-Shetty-function is given by the following formula:

$$f(x_1, x_2) = (x_1 - 2)^4 + (x_1 - 2x_2)^2 \tag{2.45}$$

The Bazaraa-Shetty-function is polynom of order 4, with one local minimum on the point (2, 1), which is also a global minimum, and the function value is zero at this point.
The start point for the iteration is $x = (4, 2)$.



a. Bazaraa-Shetty-function b. Isolines of the Bazaraa-Shetty-function

Figure 2.3: Bazaraa-Shetty-function

The Image 2.3 shows us the characteristics of the Bazaraa-Shetty-function. By the characteristics it is clear, that we have a convex function, this ensures us the unique convergence.
But the characteristics show also, that we have a rose flatter along the line $2x_1 = x_2$. This results to the problem, that the gradient based methods have a slow convergence, this is shown in the Table 2.5 and 2.6. But in the tables it is also shown, that the convexness of the functions give us a fast convergence by Newton-like algorithms.

2.4.4 Result of this small experiment

In this subsection we summarize the results of the last three numerical experiments.

Algorithm	Iterations	function value	point	time	factor
SDA-ASC	177	9.83874E-006	2.05596, 1.02807	1.5919E-004	306.7
SDA-PSC	163	9.73412E-006	2.05579, 1.02800	1.0662E-003	2054.1
CGA-ASC	59	9.71482E-006	2.05562, 1.02800	4.3045E-005	82.9
CGA-PSC	25	7.46822E-009	2.00490, 1.00241	1.4415E-004	277.7
NA	9	7.32546E-006	2.05202, 1.02601	5.1905E-007	1.0
DNA	9	7.32546E-006	2.05202, 1.02601	1.2960E-006	2.5
NMA	23	3.57827E-006	2.01635, 1.00911	7.1960E-006	13.9

Table 2.5: Bazarra-Shetty-function with an error $< 10^{-5}$.

Algorithm	Iterations	function value	point	time	factor
SDA-ASC	10000	2.53480E-009	2.00710, 1.00355	6.3383E-003	1132.6
SDA-PSC	10000	2.50452E-009	2.00707, 1.00354	6.6983E-002	11969.8
CGA-ASC	10000	2.62658E-010	2.00403, 1.00201	6.7220E-003	1201.2
CGA-PSC	101	5.48218E-011	1.99858, 0.99929	5.6730E-004	101.4
NA	16	8.59566E-011	2.00304, 1.00152	5.5960E-006	1.0
DNA	16	8.59566E-011	2.00304, 1.00152	1.0385E-005	1.9
NMA	37	4.03296E-011	2.00222, 1.00111	1.0959E-005	2.1

Table 2.6: Bazarra-Shetty-function with an error $< 10^{-10}$ or max 10000 iterations.

Gradient based algorithms

Considering the time values and the factors with respect to the other methods, one can notice, that this algorithms are slower. In some cases we can see, that the convergence is directly dependent on the stepsize algorithm.
But it is clear, that this three models are not objective and in general legal. An important point is, that the convergence is dependent on the stepsize control. That is the reason, why better stepsize controls are needed.

Uniqueness of the solution

The example of the Himmelblau-function shows very clearly the problem, if we do not have a convex function. In this example it is not a problem, because each local minimum is also a global one.
But in a general case it is a problem, that it is possible, that we find only a local minimum.

Conclusion

This chapter and especially the examples ought to give a small overview about the optimization and the numerical optimization. It is clear, that we have considered only a very small part of this topic. And many question are open, for example, how we choose the stepsize efficiently, or which other methods are also good, and efficient.
But the main focus of this work about numerical optimization is the automatic differentiation and the application on the flow of fluids.

Chapter 3

The Navier-Stokes equations

In this chapter we give an introduction to the Navier-Stokes equations for incompressible fluids, which describe fluid dynamics. At first we explain the model, then we give an overview about the numerical methods to solve these equations.

3.1 The model

In this section we give a small derivation for the flow dynamic equations, at first for the Euler equations and then for the Navier-Stokes equations.
The derivation of this model based on the book of Landau and Lifschitz *Lehrbuch der theoretischen Physik VI* [Lan74] and Griebel, Dornseifer and Neunhoeffer *Numerische Simualtion in der Strömungsmechanik* [MG95] and the lecture notes from Urban in the third part of Scientyfic Computing course in winter term 2006/2007 on the University Ulm.
At first, we define some necessary notions.

Definition 3.1.1 (Fluid)
The notation **fluid** *comes from the latin word* **fuldius** *and stands for the flow. It is a main notation for gasses and liquids.*

Definition 3.1.2 (Newton-Fluid)
A **Newton-Fluid** *is a fluid, which has some special characteristics; the shear stress τ is proportional to the shear rate $\frac{du}{d\nu}$, where u is the velocity parallel to the wall and ν is the normal vector to the wall.*

$$\tau = \eta \frac{du}{d\nu}$$

The proportional constant η is called dynamic viscosity.

Remark 3.1.3 (Examples for Newton and non-Newton fluids)
Newton fluids : water, oil, gases
Non-Newton fluids : blood, glycerin

Definition 3.1.4 (Ideal fluid)
A fluid is called **Ideal fluid***, if the fluid fulfill no internal friction and no head conduction.*

3.1.1 The Euler- and Navier-Stokes equations

Here we derive the Euler equations and in the following we extend the model to the Navier-Stokes equations.
We start with a heuristic for the flow of fluids and, after this, we concretize this heuristic.

Heuristic

We consider a fluid in a domain $\Omega \subset \mathbb{R}^d$, $d = 2, 3$. This domain is called flow-domain. We observe the flow of the fluid in the flow-domain over the time $t \in [0, T_{end}] =: I_T$. With the incompressibility of the fluid follows for the density $\rho : \Omega \times I_T \to \mathbb{R}_+$, that $\rho(x, t) = \rho_\infty = const$ holds.
The flow is characterized by:

- the **velocity field** $u : \Omega \times I_T \to \mathbb{R}^d$,
- the **pressure** $p : \Omega \times I_T \to \mathbb{R}$ and
- a **external forces** $g : \Omega \times I_T \to \mathbb{R}^d$.

Now we present present kinds of boundary-conditions:

(i) **Slip-Boundary-Condtions,**

(ii) **Non-Slip-Boundary-Condtions,**

(iii) **Inflow-Boundary-Condtions,**

(iv) **Outflow-Boundary-Condtions,**

(v) **Periodic-Boundary-Condtions.**

With the help of this heuristic, we can start with the modeling of the Euler equations.

Modeling

We seperate the modeling into two parts, the conservation of mass and conservation of momentum.
We consider $\Omega_{i,0} \subset \Omega$ and the function $\phi : \Omega_{i,0} \times I_T \to \Omega_{i,t} \subset \Omega$, which describes the changing of the position of a particle. $\Omega_{i,t}$ is a closed system, which means, that there exist no flow over the borders of $\Omega_{i,t}$. The course of a particle c is given by the graph of the the function $t \to \phi(c, t)$, and the velocity on fixed point $x := \phi(c, t)$ is given by

$$u(x, t) = \frac{\partial}{\partial t}\phi(c, t). \tag{3.1}$$

The derivation is based on the **Transport theorem** and the **Gauss' theorem-Gradient**, this is why we show this theorem, before we start with the modeling.

Theorem 3.1.5 (Transport theorem)
Let h be a continuous differentiable function and let $h : \Omega_{i,t} \times I_T \to \mathbb{R}$, then holds:

$$\frac{d}{dt}\int_{\Omega_{i,t}} h(x, t)\, dx = \int_{\Omega_{i,t}} \left(\frac{d}{dt}h + div(h(u))\right)(x, t)\, dx. \tag{3.2}$$

For the proof of these theorem we need the **Wronski-determinant**.

Lemma 3.1.6 (Wronski-determinant)
Let $A, Y : I \to \mathbb{R}^{d\times d}$ be two matrix functions and $I \subset \mathbb{R}$. And Y is a Wronski-matrix, i.e. Y fulfill

$$\frac{d}{dt}Y(t) = A(t)Y(t), \tag{3.3}$$

then holds

$$\frac{d}{dt}(det\, Y(t)) = (trA(t))(detY(t)). \tag{3.4}$$

And $trA := \sum_{i=1}^{n} a_{i,i}$ is the trace of A.

Proof.
Let $Y_i := (y_{1,i}(t), \dots, y_{n,i}(t))$ be the i-th row of $Y(t)$, this implies

$$\frac{d}{dt}Y_i(t) \overset{(3.3)}{=} \sum_{j=1}^{m} a_{i,j}Y_j(t),\ for\ i = 1, \dots, n. \tag{3.5}$$

By the definition of the determinant: $detA = \sum_{\sigma\in S_n}(sgn\,\sigma)a_{1,\sigma_1}\cdots a_{n,\sigma_n}$ follows:

$$\begin{aligned}
\frac{d}{dt}(det\, Y(t)) &\overset{product\ rule}{=} \sum_{i=1}^{n}\sum_{\sigma\in S_n}(sgn\,\sigma)y'_{i,\sigma_i}(t)\cdot\prod_{j=1\ j\neq i}^{n} y_{j,\sigma_j}(t)\\
&= \sum_{i=1}^{n} det\left(Y_1(t), \dots, Y_{i-1}(t), \frac{d}{dt}Y_i(t), Y_{i+1}(t), \dots, Y_n(t)\right)^T\\
&\overset{(3.5)}{=} \sum_{i,j=1}^{n} a_{i,j}(t)\underbrace{det\left(Y_1(t), \dots, Y_{i-1}(t), Y_j(t), Y_{i+1}(t), \dots, Y_n(t)\right)^T}_{=0\ for\ all\ j\neq i}\\
&= \sum_{i=1}^{n} a_{i,i}(t)\cdot(det\, Y(t))\\
&= trA(t)\cdot(det\, Y(t)).
\end{aligned}$$

□

Now we can prove the **Transport theorem**.

Proof of Theorem 3.1.5. With (3.1) follows:

$$\frac{\partial}{\partial x}\frac{\partial}{\partial t}\phi(x,t) \overset{chain\ rule}{=} \underbrace{u_x}_{=\frac{\partial u}{\partial x}}(\phi(x,t),t)\cdot\underbrace{\phi_x}_{=\frac{\partial\phi}{\partial x}}(x,t),$$

and this implies

$$\frac{\partial}{\partial t}\underbrace{\phi_x(x,t)}_{=Y(t)} = \underbrace{u_x(\phi(x,t),t)}_{=A(t)}\cdot\underbrace{\phi_x(x,t)}_{=Y(t)}.$$

We define $J(x,t) := det\, \phi_x(x,t)$, so follows with the Lemma 3.1.6:

$$\frac{\partial}{\partial t} J(x,t) \overset{(3.4)}{=} (tr\, u_x(\phi(x,t),t)) \cdot (J(x,t)). \tag{3.6}$$

We use the transformation $\Omega_{i,t} = \phi(\Omega_{i,0}, t)$, $x(t) = \phi(\hat{x}, t)$ and

$$\int_{\Omega_{i,t}} h(x,t)dx = \int_{\Omega_{i,0}} h(\phi(\hat{x},t),t) \cdot \underbrace{J(\hat{x},t)}_{=det\, \phi_x(\hat{x},t)} d\hat{x}.$$

This implies

$$\begin{aligned}
\frac{d}{dt}\int_{\Omega_{i,t}} h(x,t)dx &\overset{int.\ trafo.}{=} \int_{\Omega_{i,0}} \frac{\partial}{\partial t} h(\phi(\hat{x},t),t) \cdot J(\hat{x},t) d\hat{x} \\
&\quad + \int_{\Omega_{i,0}} h(\phi(\hat{x},t),t) \cdot \frac{\partial}{\partial t} J(\hat{x},t) d\hat{x} \\
&= \int_{\Omega_{i,0}} \big(h_t(\phi(\hat{x},t),t) + \nabla h_t(\phi(\hat{x},t),t) \underbrace{\frac{\partial}{\partial t}\phi(\hat{x},t)}_{\overset{(3.1)}{=} u(\hat{x},t)}\big) \cdot J(\hat{x},t)\, d\hat{x} \\
&\quad + \int_{\Omega_{i,0}} h(\phi(\hat{x},t),t) \cdot (div\, u(\phi(\hat{x},t),t)) \cdot J(\hat{x},t) d\hat{x} \\
&= \int_{\Omega_{i,0}} (h_t + \underbrace{\nabla h \cdot u + h \cdot div\, u}_{=div(h\cdot u)}) \cdot (\phi(\hat{x},t)) \cdot \underbrace{J(\hat{x},t)}_{=det\, \phi_x(\hat{x},t)} d\hat{x} \\
&\overset{inv.\ trafo}{=} \int_{\Omega_{i,t}} (\frac{\partial}{\partial t} h + div(hu))(x,t)dx.
\end{aligned}$$

□

After this we consider the **Gauss' theorem-Gradient**.

Theorem 3.1.7 (Gauss' theorem-Gradient)
Let $\Omega \subset \mathbb{R}^3$, a compact domain, where $dim(\Omega) = 3$, and the boundary of Ω is piecewise smooth, $\nu = (\nu_1, \nu_2, \nu_3)^T$ is the standardized outer normal vector and $p \in C^1(\Omega) \to \mathbb{R}$, then holds

$$\oint_{\partial\Omega} p \cdot \nu\, dS = \int_{\Omega} \nabla p\, dV. \tag{3.7}$$

Proof.
w.l.o.g the domain is given $\Omega = I_1 \times I_2 \times I_3$ and $I_\ell = [0,1]\ \ell = 1,2,3$, otherwise we consider Ω_i and $\Omega = \lim_{n\to\infty} \bigcup_{i=1}^{n} \Omega_i$ and $\Omega_i := I_{1,i} \times I_{2,i} \times I_{3,i}$ with $I_{\ell,i} := [a_i, b_i]\ a_i, b_i \in \mathbb{R},\ \ell = 1,2,3$ and by the linearity of the integral follows the result for Ω.
Now $\Omega = I_1 \times I_2 \times I_3$ and $\Gamma_{i,a} := \{x_i = a \in \mathbb{R},\ x_j \in [0,1]\ \forall\, j \in \{1,2,3\} \setminus i\}$, then the boundary of Ω, is $\partial\Omega = \bigcup_{i=1}^{3} (\Gamma_{i,0} \cup \Gamma_{i,1})$. It is clear, that $\nu_i = 1$ and $\nu_\ell = 0\ \ell \in \{1,2,3\} \setminus i$ on $\Gamma_{i,1}$ and $\nu_i = -1$ and $\nu_\ell = 0\ \ell \in \{1,2,3\} \setminus i$ on $\Gamma_{i,0}$.

With these notations follows,

$$\begin{aligned}
\oint_{\partial\Omega} p\cdot\nu\, dS = & \quad \sum_{i=1}^{3}\int_{\Gamma_{i,0}} p\cdot\nu\, dS + \int_{\Gamma_{i,1}} p\cdot\nu\, dS \\
= & \quad \begin{pmatrix} \int_{\Gamma_{1,1}} p\, dS - \int_{\Gamma_{1,0}} p\, dS \\ \int_{\Gamma_{2,1}} p\, dS - \int_{\Gamma_{2,0}} p\, dS \\ \int_{\Gamma_{3,1}} p\, dS - \int_{\Gamma_{3,0}} p\, dS \end{pmatrix} \\
= & \quad \begin{pmatrix} \int_{I_2}\int_{I_3} p(1,x_2,x_3) \; - p(0,x_2,x_3)\, dx_2dx_3 \\ \int_{I_1}\int_{I_3} p(x_1,1,x_3) \; - p(x_1,0,x_3)\, dx_1dx_3 \\ \int_{I_1}\int_{I_2} p(x_1,x_2,1) \; - p(x_1,x_2,0)\, dx_1dx_2 \end{pmatrix} \\
= & \quad \begin{pmatrix} \int_{I_2}\int_{I_3}\left(\int_{I_1} \frac{\partial}{\partial x_1} p(x_1,x_2,x_3)\, dx_1\right) dx_2dx_3 \\ \int_{I_1}\int_{I_3}\left(\int_{I_2} \frac{\partial}{\partial x_2} p(x_1,x_2,x_3)\, dx_2\right) dx_1dx_3 \\ \int_{I_1}\int_{I_2}\left(\int_{I_3} \frac{\partial}{\partial x_3} p(x_1,x_2,x_3)\, dx_3\right) dx_1dx_2 \end{pmatrix} \\
= & \quad \int_\Omega \nabla p\, dV.
\end{aligned}$$

□

Followed by this dispositions, we start start with the modeling of the **Conservation of the mass**.

3.1.2 Conservation of mass

Now we consider a fixed domain Ω_i. The mass of the fluid in this domain Ω_i is given by the integral over the density ρ. By the function ϕ follows, that the fluid, which is in $\Omega_{i,0}$ on the time $t = 0$, is for $t \geq 0$ in the domain $\Omega_{i,t}$. This implies, that for all $t \geq 0$ the equation:

$$\int_{\Omega_{i,0}} \rho(x,0)dx = \int_{\Omega_{i,t}} \rho(x,t)dx, \tag{3.8}$$

holds. With the Equation (3.8) follows, that the derivative of the mass with respect to the time t is constant zero. This result implies by using the Transport Theorem 3.1.5,

$$0 = \int_{\Omega_{i,t}} \left(\frac{\partial}{\partial t}\rho + div\,(\rho u)\right)(x,t)dx \;\; for\, all\, \Omega_{i,t},\; t \geq 0. \tag{3.9}$$

The Equation (3.9) holds for all domains $\Omega_{i,t}$, in particular for arbitrary small domains, this implies, that the integrant is zero. The outcome of this is the continuity equation for compressible fluids:

$$\frac{\partial}{\partial t}\rho + div\,(\rho u) = 0. \tag{3.10}$$

And for incompressible fluids follows,

$$div\,(u) = 0. \tag{3.11}$$

3.1.3 Conservation of momentum

Let Ω be the domain, in which we consider the flow of a fluid, and where $\Omega_{i,t} \subset \Omega$ is a fixed volume. The moment of the fluid in the domain $\Omega_{i,t}$ is given by the integral:

$$m(t) := \int_{\Omega_{i,t}} \rho(x,t)u(x,t)dx. \tag{3.12}$$

Applying the **2. Newton's law of motion** on the time-based change of the momentum $m(t)$ is given by the sum of all forces F acting on the fluid in the domain $\Omega_{i,t}$,

$$\frac{\partial}{\partial t}m(t) = F(x,t). \tag{3.13}$$

The complete forces of the ideal fluid in Ω_i are given by:

- Extern forces F_e (e.g. gravitation) are given in the following form

  $$F_e := \int_{\Omega_{i,t}} \rho(x,t)f(x,t)dx,$$

  where $f(x,t)$ is the force density per unit volume.

- Sureface forces F_S(e.g. pressure, inner friction) are given in the following form

  $$F_s := \int_{\delta\Omega_{i,t}} \sigma(x,t)\nu \, dx,$$

  where $\sigma(x,t) := \begin{pmatrix} \sigma_{1,1} & \sigma_{1,2} & \sigma_{1,3} \\ \sigma_{2,1} & \sigma_{2,2} & \sigma_{2,3} \\ \sigma_{3,1} & \sigma_{3,2} & \sigma_{3,3} \end{pmatrix}$ is the stress tensor.

This implies

$$F(x,t) := F_e + F_s.$$

By the theorem of Gauss 3.1.7 follows:

$$\begin{aligned} F_s(x,t) &= \oint_{\partial\Omega_{i,t}} \sigma \cdot \nu \, dS \\ &= \oint_{\partial\Omega_{i,t}} \underbrace{\begin{pmatrix} \sigma_{1,1} \\ \sigma_{2,1} \\ \sigma_{3,1} \end{pmatrix}}_{=:\sigma_{(\cdot,1)}} \cdot\nu + \underbrace{\begin{pmatrix} \sigma_{1,2} \\ \sigma_{2,2} \\ \sigma_{3,2} \end{pmatrix}}_{=:\sigma_{(\cdot,2)}} \cdot\nu + \underbrace{\begin{pmatrix} \sigma_{1,3} \\ \sigma_{2,3} \\ \sigma_{3,3} \end{pmatrix}}_{=:\sigma_{(\cdot,3)}} \cdot\nu \, dS \\ &= \int_{\Omega_{i,t}} \nabla\sigma_{(\cdot,1)} \, dV + \int_{\Omega_{i,t}} \nabla\sigma_{(\cdot,2)} \, dV + \int_{\Omega_{i,t}} \nabla\sigma_{(\cdot,3)} \, dV \\ &= \int_{\Omega_{i,t}} \mathbf{div}\, \sigma \, dV. \end{aligned} \tag{3.14}$$

So we can write the **2. Newton's law of motion** in the following form:

$$\frac{\partial}{\partial t}\int_{\Omega_{i,t}} \rho(x,t)u(x,t)dx = \int_{\Omega_{i,t}} \rho(x,t)f(x,t)dx - \int_{\Omega_{i,t}} \mathbf{div}\, \sigma \, dV \tag{3.15}$$

Appling the product rule and the Transport Theorem 3.1.5 on the left hand side of (3.15) gives us:

$$\int_{\Omega_{i,t}} \frac{\partial}{\partial t}(\rho u) + (u \cdot \nabla)(\rho u) + (\rho u)\mathbf{div}\, u \, dx = \int_{\Omega_{i,t}} \rho(x,t) f(x,t) dx + \int_{\Omega_{i,t}} \mathbf{div}\, \sigma \, dV \quad (3.16)$$

The Equation (3.16) holds for all domains $\Omega_{i,t}$, in particular for arbitrary small domains, this implies, that the equation holds also for the integrants. This gives the **momentum equation** in an abstract form.

$$\frac{\partial}{\partial t}(\rho u) + (u \cdot \nabla)(\rho u) + (\rho u)\mathbf{div}\, u - \mathbf{div}\, \sigma - \rho f = 0. \quad (3.17)$$

3.1.4 Euler equations

If we consider an **ideal fluid**, we get the stress tensor in the following form:

$$\sigma(x,t) := -p(x,t)\mathbf{I}.$$

Compressible

So we get the **Euler equations** for compressible fluids:

$$\frac{\partial}{\partial t}(\rho u) + (u \cdot \nabla)(\rho u) + (\rho u)\mathbf{div}\, u + \nabla p = \rho f. \quad (3.18)$$

Incompressible

The **Euler equations** for incompressible fluids, which follows by combining the Equation (3.11) and (3.18), is given by:

$$\frac{\partial}{\partial t}(u) + (u \cdot \nabla)(u) + \frac{1}{\rho}\nabla p = f. \quad (3.19)$$

3.1.5 Navier-Stokes equations

If we consider a **viscous fluid**, we get the stress tensor in the following form:

$$\sigma := (-p + \lambda \mathbf{div}\, u)\mathbf{I} + \mu \left[\left(\frac{\partial \mathbf{u_i}}{\partial \mathbf{x_j}} + \frac{\partial \mathbf{u_i}}{\partial \mathbf{x_j}} \right) \right]_{\mathbf{i,j=1,2,3}}.$$

With this stress tensor, the momentum Equation (3.17) is transformed in a system of partial differential equation of second order.

Compressible

This is the momentum equation of the **Navier-Sokes equations** for **compressible fluids**:

$$\frac{\partial}{\partial t}(\rho u) + (u \cdot \nabla)(\rho u) + (\rho u)\mathbf{div}\, u + \nabla p = (\mu + \lambda)\nabla(\mathbf{div}\, u) + \mu \Delta u + \rho f. \quad (3.20)$$

Incompressible

The momentum equation of the **Navier-Sokes equations** for **incompressible fluids**, which follows by combining the Equation (3.11) and (3.20) is given by:

$$\frac{\partial}{\partial t}(\rho u) + (u \cdot \nabla)(\rho u) + \nabla p = \mu \Delta u + \rho f. \tag{3.21}$$

The complete system of the **Navier-Stokes equations** is:

$$\begin{aligned} \frac{\partial}{\partial t}(\rho u) + (u \cdot \nabla)(\rho u) + \nabla p = & \quad \mu \Delta u + \rho f \quad \text{in } \Omega \\ div\, \overrightarrow{u} \; = & \quad 0 \quad \text{in } \Omega \\ u \; = & \quad g \quad \text{on } \delta\Omega \end{aligned} \tag{3.22}$$

Definition 3.1.8 (Divergence)
The divergence **div** *is a continuous operator, which is defined as: Let* $f \in C^1(\Omega)$ *and* $\Omega \subset \mathbb{R}^n$, $f : \Omega \to \mathbb{R}^n$, *then*

$$\mathbf{div} f := \sum_{i=1}^{n} \frac{\partial f_i}{\partial x_i} \tag{3.23}$$

A more detailed derivation to the Navier-Stokes equations is given in [Lan74] and [MG95]. Also we have not considered the energy transport in the Navier-Stokes equations. This is also given in [MG95].

3.2 Numerical methods

In this section we give an introduction in two methods to solve the Navier-Stokes equations approximatively. We consider the **finite differences method** and the **finite volume method**.

3.2.1 Finite differences method

This is a simple method to solve ODE's and PDE's, which approximates the derivatives with *finite differences (FD)*.
We start with some general topics about the finite differences method, then we consider the finite differences method especially for the Navier-Stokes equations. A reference for the finite differences method is for example the book from Knabner and Angermann [Kna00].

Theory of the finite differences method

Definition 3.2.1 (Finite difference)
Let $f \in C(I)$ *and* $I \subset \mathbb{R},\ h \in \mathbb{R}$ f'_h *is called:*

- *symmetric finite difference, if* $f'_h := \frac{f(x+h)-f(x-h)}{2h}$ *and* $(x-h), (x+h) \in I$
- *backward finite difference, if* $f'_h := \frac{f(x)-f(x-h)}{h}$ *and* $(x-h), x \in I$

– *forward finite difference, if* $f'_h := \frac{f(x+h)-f(x)}{h}$ *and* $x, (x+h) \in I$

If $f \in C^n(I)$ *and* $n \in \mathbb{N}$ *and* $h \to 0$*, then* $f'_h \to f'$.

The principle of the finite difference method is to approximate a boundary value problem on a finite number of grid points in $\overline{\Omega}$. We approximate the derivative of the PDE with finite differences, and we approximate the function values on the inner grid points, the boundary condition on boundary grid-points. In the Figure 3.1 a grid for finite differences is visuallized.

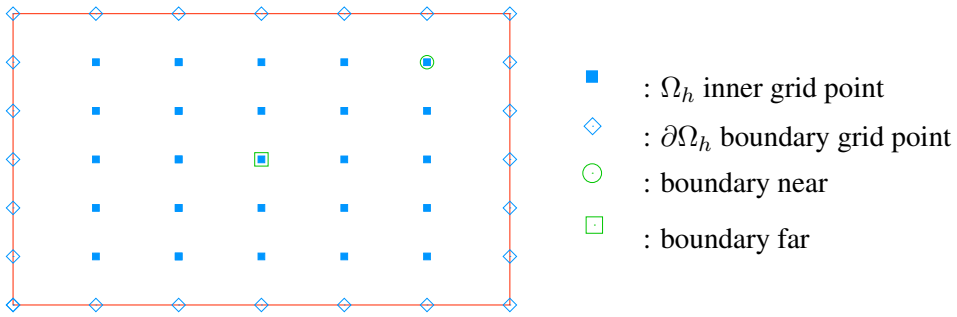


Figure 3.1: Example for a finite differences grid

Lemma 3.2.2 (Approximation Order of Finite Differences)
Let $I \supseteq (x-h, x+h)$ *for* $x \in \mathbb{R},\ h > 0$*, then there exists a value* R*, which is depended on the function* f *but independent of the stepsize* h*. For this the following estimations hold:*

(i) for $f \in C^2(\bar{I})$*:*

$$f'(x) = \frac{f(x+h) - f(x)}{h} + hR \ \text{ and } \ |R| \leq \frac{1}{2}\|f''\|_\infty, \tag{3.24}$$

(ii) for $f \in C^2(\bar{I})$*:*

$$f'(x) = \frac{f(x) - f(x-h)}{h} + hR \ \text{ and } \ |R| \leq \frac{1}{2}\|f''\|_\infty, \tag{3.25}$$

(iii) for $f \in C^3(\bar{I})$*:*

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + h^2R \ \text{ and } \ |R| \leq \frac{1}{6}\|f^{(3)}\|_\infty, \tag{3.26}$$

(iv) for $f \in C^4(\bar{I})$*:*

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + h^2R \ \text{ and } \ |R| \leq \frac{1}{6}\|f^{(4)}\|_\infty. \tag{3.27}$$

Proof.
The results (i)-(iv) we prove by the Taylor series.

(i) $f(x+h) = f(x) + hf'(x) + \underbrace{\frac{h^2}{2}f''(x+\xi)}_{\leq \frac{h^2}{2}\|f''\|_\infty}$ and $\xi \in (0,h)$

this implies the Equation (3.24), for $R \geq \frac{1}{2}\|f''\|_\infty$.

(ii) $f(x-h) = f(x) - hf'(x) + \underbrace{\frac{h^2}{2} f''(x-\xi)}_{\leq \frac{h^2}{2}\|f''\|_\infty}$ and $\xi \in (0,h)$

this implies the Equation (3.25), for $R \geq \frac{1}{2}\|f''\|_\infty$.

(iii)

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f^{(3)}(x+\xi), \tag{3.28}$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f^{(3)}(x-\xi) \tag{3.29}$$

and $\xi \in (0,h)$, combining the Equations (3.28) and (3.29) with a subtraction, implies the Equation (3.26), for $R \geq \frac{1}{6}\|f^{(3)}\|_\infty$.

(iv)

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f^{(3)}(x) + \frac{h^4}{12} f^{(4)}(x+\xi), \tag{3.30}$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f^{(3)}(x) + \frac{h^4}{12} f^{(4)}(x-\xi) \tag{3.31}$$

and $\xi \in (0,h)$, combining the Equations (3.30) and (3.31) with a addition, implies the Equation (3.27), for $R \geq \frac{1}{6}\|f^{(4)}\|_\infty$.

□

Remark 3.2.3
The Lemma 3.2.2 shows the problem of the finite difference method, because the results about the approximation order need at least a twice continuously differentiable function, and for the approximation of the second derivative we need actually four times continuously differentiable functions.
This is a very strong condition, in the reality.

In the Example 3.2.4 we show a realization of a diffusion and convection problem with central differences and the upwind method in two versions.

Example 3.2.4 (Staggered grid)
Let

$$-\frac{d^2u}{dx^2} + k\frac{du}{dx} = f \; and \; x \in (a,b), \tag{3.32}$$

and with the Dirichlet boundary conditions:

$$u(a) =: u_a, \; u(b) =: u_b. \tag{3.33}$$

For this example we choose $a = 0$, $b = 1$, $u_a = 1$, $u_b = 0$ and the right hand side is given by $f(x) = -1560x^{38} + k \cdot 40x^{39}$. So we know the exact solution. And the exact solution is $u(x) = 1 - x^{40}$.
In the matrix vector notation we get:

$$Au = f, \tag{3.34}$$

where A represents the finite difference matrix (dependent on the kind of finite differences), u is the solution vector and f is the discretisation of the right hand side.

(i) At first we choose central differences for the first and second derivative. We choose equidistant grid-points for the discretisation.

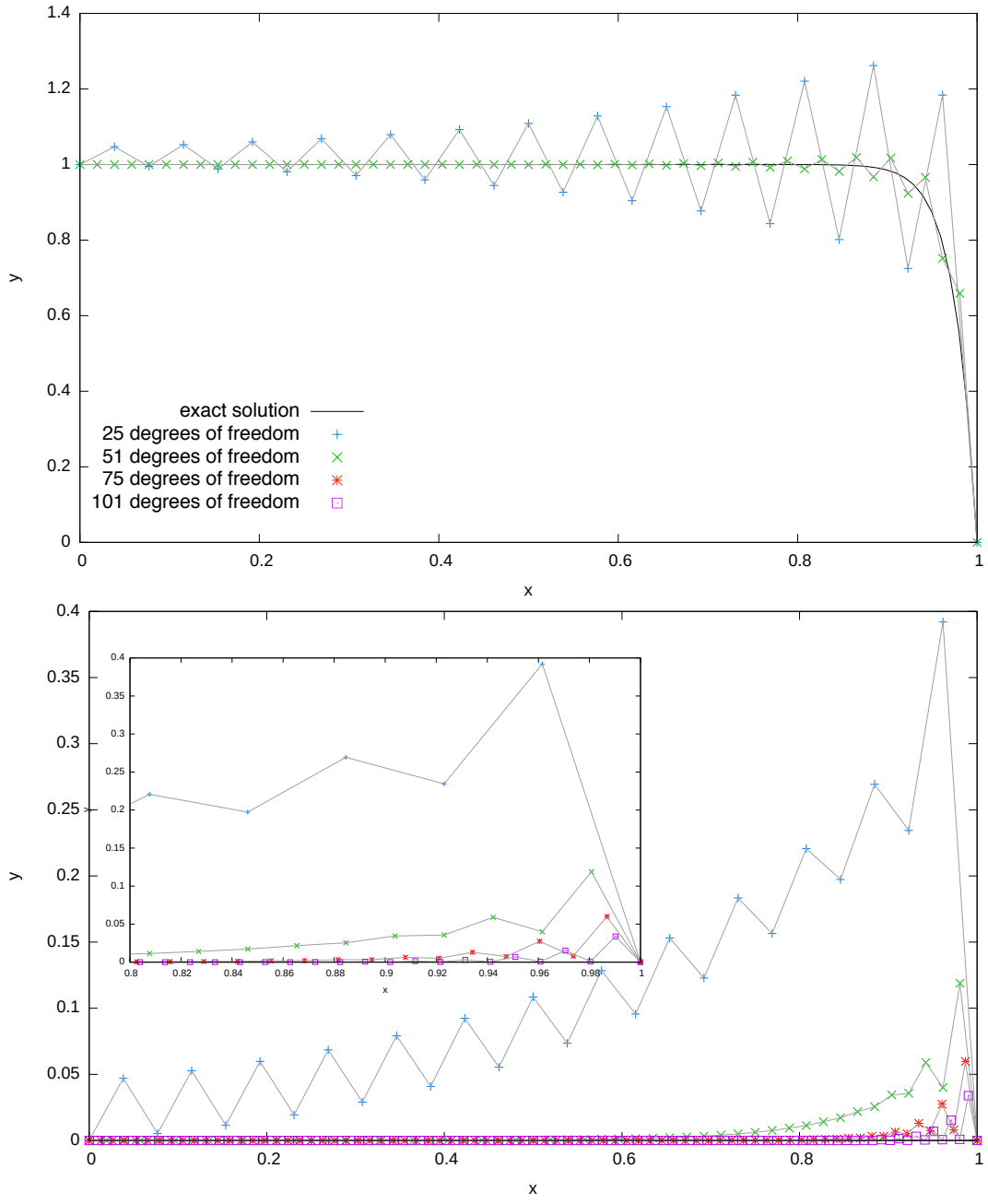


Figure 3.2: Example 3.2.4 (i) central differnences.

(ii) Now we choose central differences for the second derivative (the diffusion term) and backward differences for the first derivative (the convection term). We choose equidistant grid-points for the discretisation.

(iii) Now we choose central differences for the second derivative (the diffusion term) and backward differences for the first derivative (the convection term). We choose non equidis-

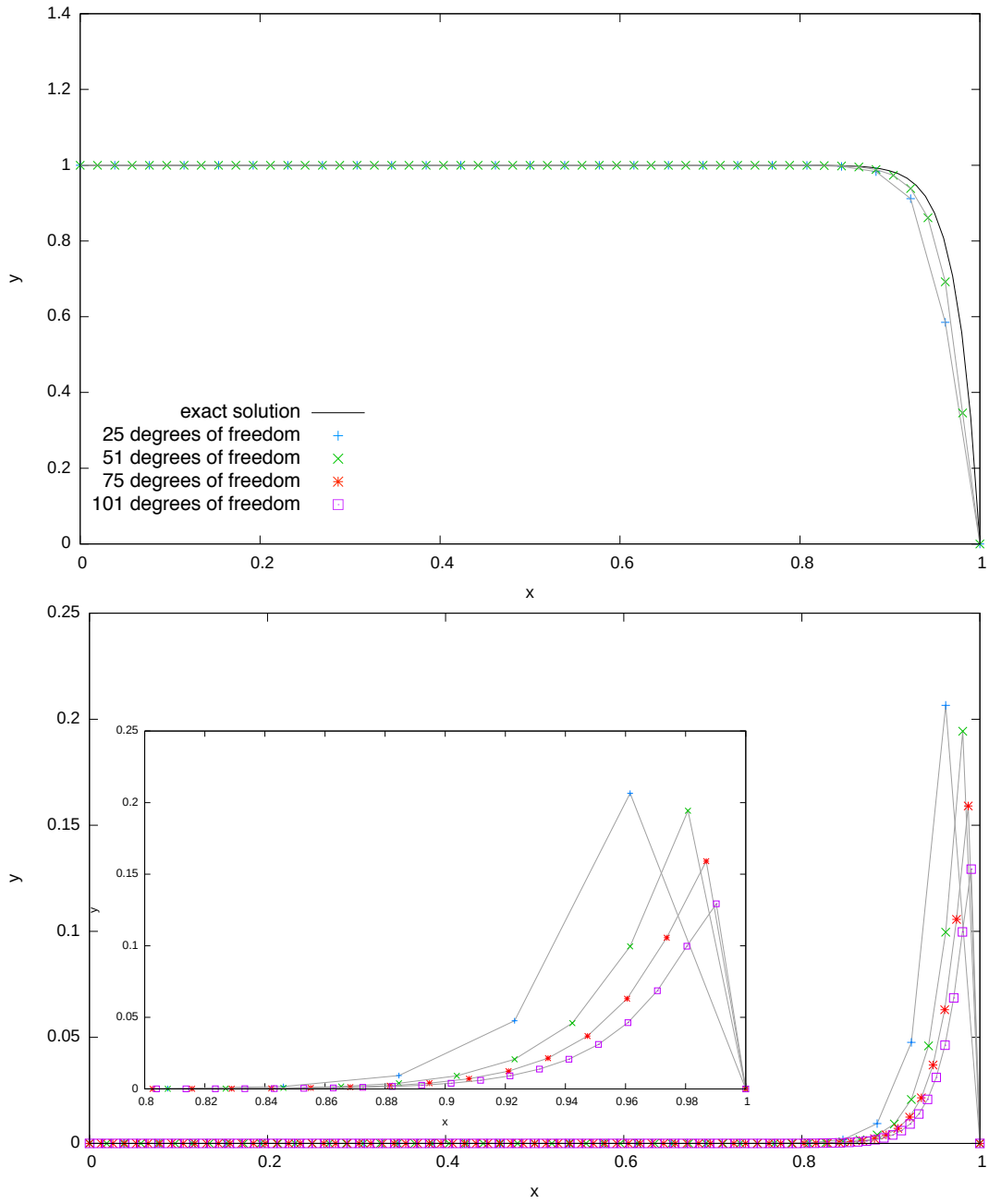


Figure 3.3: Example 3.2.4 (ii) backward differences.

tant grid-points for the discretisation. We divide the interval $I = (0,1)$ *into two subintervals* $I_1 := (0, 0.75]$ *and* $I_2 := (0.75, 1)$ *the grid-points on* I_1 *are defined with* $x_i := i\frac{0.75}{(N+1)/2}$, $i = 1, \ldots, (N+1)/2$ *and the grid-points on* I_2 *are defined with* $x_{(N+1)/2+i} := 0.75 + i\frac{0.25}{(N+1)/2}$, $i = 1, \ldots, (N-1)/2$.

This example shows some typical points of the finite differences method.
In (i) we see the problem of oscillating shown in the Figure 3.2. In the cases (ii) and (iii), which are shown in the Figures 3.3 and 3.4, we see, that this problem is solved by using backward

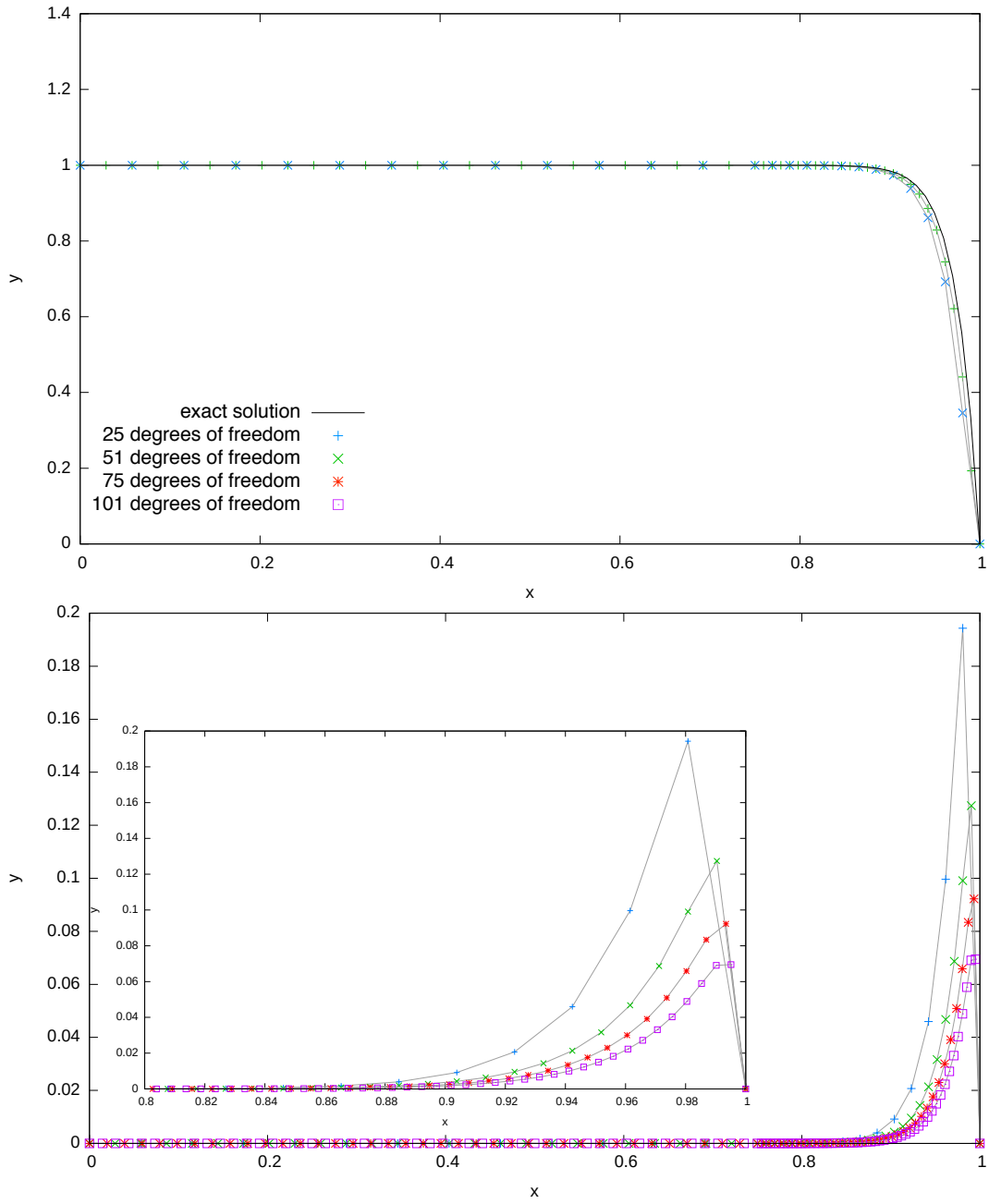


Figure 3.4: Example 3.2.4 (iii) backward differences non equidistance.

differences, if the k is positive.
Another interesting point is shown in the error plots in the Figures 3.2, 3.3 and 3.4: the error of (i) decreases quadratically and the error of (ii) and (iii) decrease only linearly. This observation is clear with the Lemma 3.2.2.
In the direct accord of (ii) and (iii), we see, that the error of (iii) decreases faster than the error of the example (ii). The reason of this are the non equidistant grid points in (iii).

Remark 3.2.5
It is also possible to reduce the oscillations with a linear combination of the central and the upwind differences. In this case we get the matrix A in the following form:

$$A := \gamma A_{upwind} + (1-\gamma) A_{central}.$$

The parameter γ is in the interval $[0,1]$ and, if the convective term is big, the parameter must chosen nearly by 1.
Another ansatz is the Donor-Cell-Schema. It is normally used by convection terms with the following structure $\frac{d(ku)}{dx}$. For more details see [MG95].

Discretisation of the Navier-Stokes equations

Here we explain the discretisation of the domain, in which we consider the flow of the fluid.
To simplify the discretisation we consider only rectangular domains:

$$\Omega := [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3] \subset \mathbb{R}^3.$$

We discretisize this domain with a equidistant grid:

$$\begin{aligned} \Omega_{discret} &:= \{(x_{1,i}, x_{2,j}, x_{3,k}) \in \Omega : x_{i,1} := a_1 + i\delta x_1,\ x_{j,2} := a_2 + j\delta x_2,\ x_{k,3} := a_3 + k\delta x_3\} \\ &= \{(x_{1,i}, x_{2,j}, x_{3,k}) : i = 0, \ldots, i_{max},\ j = 0, \ldots, j_{max},\ k = 0, \ldots, k_{max}\}, \end{aligned}$$

where $\delta x_1,\ \delta x_2,\ \delta x_3$ are the stepsizes in the given direction, shown in the Figure 3.5.
We choose a **Staggered grid** for the discretisation, i.e. we choose three grids for the discreti-

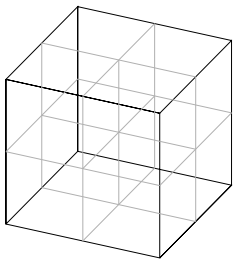


Figure 3.5: Discrete domain.

sation of the velocity, one for the distrectisation of the pressure and the energy. These four grids are staggered. This is shown in the Figure 3.6. The lines represent the grid of the location points $x_{i,j,k} := (x_{i,1}, x_{j,2}, x_{k,3})$. The staggered grids are invisible. The three grids for the velocity are moved in coordinate direction about a half stepsize. The grid for the presure and energy is in the middle of each cell.
The reason for this choice of disrectisation is to reduce the oscillations.
Now we start with the discretisation of the Navier-Stokes equations in the space.
Now we consider the impulse equation of the incompressible Navier-Stokes equations (3.22), this is the first equation.

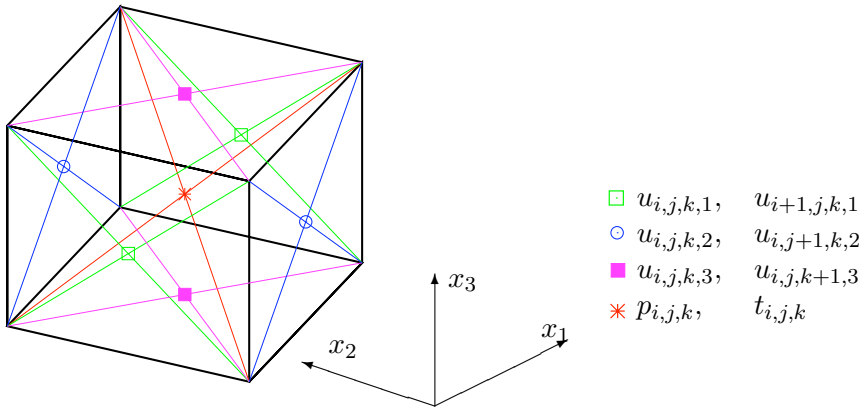


Figure 3.6: Staggered grid.

At first we consider the Laplace operator on the point $x_{i,j,k}$:

$$[\Delta u]_{i,j,k,\ell} = \begin{array}{l} \frac{u_{i+1,j,k,\ell}-2u_{i,j,k,\ell}+u_{i-1,j,k,\ell}}{\delta x_1} + \\ \frac{u_{i,j+1,k,\ell}-2u_{i,j,k,\ell}+u_{i,j-1,k,\ell}}{\delta x_2} + \\ \frac{u_{i,j,k+1,\ell}-2u_{i,j,k,\ell}+u_{i,j,k-1,\ell}}{\delta x_3} + \end{array} \tag{3.35}$$

With the staggered grid follows, that:
$u_{i,j,k,1}$ is evaluated on $(x_{i,1}, x_{j,2} - \frac{\delta x_2}{2}, x_{k,3} - \frac{\delta x_3}{2})$,
$u_{i,j,k,2}$ on $(x_{i,1} - \frac{\delta x_1}{2}, x_{j,2}, x_{k,3} - \frac{\delta x_3}{2})$ and
$u_{i,j,k,2}$ on $(x_{i,1} - \frac{\delta x_1}{2}, x_{j,2} - \frac{\delta x_2}{2}, x_{k,3})$.
The staggered grid implies, that not all variables lie on the boundary of the domain, so we need an additional boundary layer for the values, which lie outside of the original domain.
The convective term in the impulse equation is not straight forward to discretisize. Because this is the non linear term, the approximation is needed in the point $x_{i,j,k}$, and also for high Reynolds-Numbers this is the dominate term.

$$[(u \cdot \nabla)u]_{i,j,k,\ell} = \sum_{m=1}^{3} \left[\frac{\partial (u_\ell \cdot u_m)}{\partial x_m} \right]_{i,j,k,\ell}, \tag{3.36}$$

the details of the Equation (3.36) are given in the following[1]:

$\ell = 1$

$$\begin{aligned} \left[\frac{\partial (u_1^2)}{\partial x_1} \right]_{i,j,k,1} &= \frac{1}{\delta x_1} \left(\left(\frac{u_{i,j,k,1} + u_{i+1,j,k,1}}{2} \right)^2 - \left(\frac{u_{i-1,j,k,1} + u_{i,j,k,1}}{2} \right)^2 \right) \\ &+ \gamma \frac{1}{\delta x_1} \left(\frac{|u_{i,j,k,1} + u_{i+1,j,k,1}|}{2} \frac{(u_{i,j,k,1} - u_{i+1,j,k,1})}{2} \right. \\ &- \left. \frac{|u_{i-1,j,k,1} + u_{i,j,k,1}|}{2} \frac{(u_{i-1,j,k,1} - u_{i,j,k,1})}{2} \right) \end{aligned}$$

[1] We consider a combination of the Donor-Cell dicretisation and the central differences.

$$\begin{aligned}\left[\frac{\partial(u_1u_2)}{\partial x_2}\right]_{i,j,k,1} &= \frac{1}{\delta x_2}\left(\frac{(u_{i,j,k,2}+u_{i+1,j,k,2})}{2}\frac{(u_{i,j,k,1}+u_{i,j+1,k,1})}{2}\right.\\ &- \left.\frac{(u_{i,j-1,k,2}+u_{i+1,j-1,k,2})}{2}\frac{(u_{i,j-1,k,1}+u_{i,j,k,1})}{2}\right)\\ &+ \gamma\frac{1}{\delta x_2}\left(\frac{|u_{i,j,k,2}+u_{i+1,j,k,2}|}{2}\frac{(u_{i,j,k,1}-u_{i,j+1,k,1})}{2}\right.\\ &- \left.\frac{|u_{i,j-1,k,2}+u_{i+1,j-1,k,2}|}{2}\frac{(u_{i,j-1,k,1}-u_{i,j,k,1})}{2}\right)\end{aligned}$$

$$\begin{aligned}\left[\frac{\partial(u_1u_3)}{\partial x_3}\right]_{i,j,k,1} &= \frac{1}{\delta x_3}\left(\frac{(u_{i,j,k,3}+u_{i+1,j,k,3})}{2}\frac{(u_{i,j,k,1}+u_{i,j,k+1,1})}{2}\right.\\ &- \left.\frac{(u_{i,j,k-1,3}+u_{i+1,j,k-1,3})}{2}\frac{(u_{i,j,k-1,1}+u_{i,j,k,1})}{2}\right)\\ &+ \gamma\frac{1}{\delta x_3}\left(\frac{|u_{i,j,k,3}+u_{i+1,j,k,3}|}{2}\frac{(u_{i,j,k,1}-u_{i,j,k+1,1})}{2}\right.\\ &- \left.\frac{|u_{i,j,k-1,3}+u_{i+1,j,k-1,3}|}{2}\frac{(u_{i,j,k-1,1}-u_{i,j,k,1})}{2}\right)\end{aligned}$$

$\ell = 2$

$$\begin{aligned}\left[\frac{\partial(u_1u_2)}{\partial x_1}\right]_{i,j,k,2} &= \frac{1}{\delta x_1}\left(\frac{(u_{i,j,k,1}+u_{i,j+1,k,1})}{2}\frac{(u_{i,j,k,2}+u_{i+1,j,k,2})}{2}\right.\\ &- \left.\frac{(u_{i-1,j,k,1}+u_{i-1,j+1,k,1})}{2}\frac{(u_{i-1,j,k,2}+u_{i,j,k,2})}{2}\right)\\ &+ \gamma\frac{1}{\delta x_1}\left(\frac{|u_{i,j,k,1}+u_{i,j+1,k,1}|}{2}\frac{(u_{i,j,k,2}-u_{i+1,j,k,2})}{2}\right.\\ &- \left.\frac{|u_{i-1,j,k,1}+u_{i-1,j+1,k,1}|}{2}\frac{(u_{i-1,j,k,2}-u_{i,j,k,2})}{2}\right)\end{aligned}$$

$$\begin{aligned}\left[\frac{\partial(u_2^2)}{\partial x_2}\right]_{i,j,k,2} &= \frac{1}{\delta x_2}\left(\left(\frac{u_{i,j,k,2}+u_{i,j+1,k,2}}{2}\right)^2-\left(\frac{u_{i,j-1,k,2}+u_{i,j,k,2}}{2}\right)^2\right)\\ &+ \gamma\frac{1}{\delta x_2}\left(\frac{|u_{i,j,k,2}+u_{i,j+1,k,2}|}{2}\frac{(u_{i,j,k,2}-u_{i,j+1,k,2})}{2}\right.\\ &- \left.\frac{|u_{i,j-1,k,2}+u_{i,j,k,2}|}{2}\frac{(u_{i,j-1,k,2}-u_{i,j,k,2})}{2}\right)\end{aligned}$$

$$\begin{aligned}\left[\frac{\partial(u_2u_3)}{\partial x_3}\right]_{i,j,k,2} &= \frac{1}{\delta x_3}\left(\frac{(u_{i,j,k,3}+u_{i,j+1,k,3})}{2}\frac{(u_{i,j,k,2}+u_{i,j,k+1,2})}{2}\right.\\ &- \left.\frac{(u_{i,j,k-1,3}+u_{i,j+1,k-1,3})}{2}\frac{(u_{i,j,k-1,2}+u_{i,j,k,2})}{2}\right)\\ &+ \gamma\frac{1}{\delta x_3}\left(\frac{|u_{i,j,k,3}+u_{i,j+1,k,3}|}{2}\frac{(u_{i,j,k,2}-u_{i,j,k+1,2})}{2}\right.\\ &- \left.\frac{|u_{i,j,k-1,3}+u_{i,j+1,k-1,3}|}{2}\frac{(u_{i,j,k-1,2}-u_{i,j,k,2})}{2}\right)\end{aligned}$$

$\ell = 3$

$$\begin{aligned}\left[\frac{\partial(u_1u_3)}{\partial x_1}\right]_{i,j,k,3} &= \frac{1}{\delta x_1}\left(\frac{(u_{i,j,k,1}+u_{i,j,k+1,1})}{2}\frac{(u_{i,j,k,3}+u_{i+1,j,k,3})}{2}\right.\\ &- \left.\frac{(u_{i-1,j,k,1}+u_{i-1,j,k+1,1})}{2}\frac{(u_{i-1,j,k,3}+u_{i,j,k,3})}{2}\right)\\ &+ \gamma\frac{1}{\delta x_1}\left(\frac{|u_{i,j,k,1}+u_{i,j,k+1,1}|}{2}\frac{(u_{i,j,k,3}-u_{i+1,j,k,3})}{2}\right.\\ &- \left.\frac{|u_{i-1,j,k,1}+u_{i-1,j,k+1,1}|}{2}\frac{(u_{i-1,j,k,3}-u_{i,j,k,3})}{2}\right)\end{aligned}$$

$$\begin{aligned}\left[\frac{\partial(u_2u_3)}{\partial x_2}\right]_{i,j,k,3} &= \frac{1}{\delta x_2}\left(\frac{(u_{i,j,k,2}+u_{i,j,k+1,2})}{2}\frac{(u_{i,j,k,3}+u_{i,j+1,k,3})}{2}\right.\\ &- \left.\frac{(u_{i,j-1,k,2}+u_{i,j-1,k+1,2})}{2}\frac{(u_{i,j-1,k,3}+u_{i,j,k,3})}{2}\right)\\ &+ \gamma\frac{1}{\delta x_2}\left(\frac{|u_{i,j,k,2}+u_{i,j,k+1,2}|}{2}\frac{(u_{i,j,k,3}-u_{i,j+1,k,3})}{2}\right.\\ &- \left.\frac{|u_{i,j-1,k,2}+u_{i,j-1,k+1,2}|}{2}\frac{(u_{i,j-1,k,3}-u_{i,j,k,3})}{2}\right)\end{aligned}$$

$$\begin{aligned}\left[\frac{\partial(u_3^2)}{\partial x_3}\right]_{i,j,k,3} &= \frac{1}{\delta x_3}\left(\left(\frac{u_{i,j,k,3}+u_{i,j,k+1,3}}{2}\right)^2-\left(\frac{u_{i,j,k-1,3}+u_{i,j,k,3}}{2}\right)^2\right)\\ &+ \gamma\frac{1}{\delta x_3}\left(\frac{|u_{i,j,k,3}+u_{i,j,k+1,3}|}{2}\frac{(u_{i,j,k,3}-u_{i,j,k+1,3})}{2}\right.\\ &- \left.\frac{|u_{i,j,k-1,3}+u_{i,j,k,3}|}{2}\frac{(u_{i,j,k-1,3}-u_{i,j,k,3})}{2}\right)\end{aligned}$$

The parameter $\gamma \in [0,1]$ affects the choice between only central differences for $\gamma = 0$ and pure Donor-Cell discretisation. Details to the Donor-Cell discretisation are given in the book of Griebel [MG95].
One possible choice of the parameter is

$$\gamma \geq \max_{i,j,k,\ell}\left(\frac{u_{i,j,k,\ell}\delta t}{\delta x_\ell}\right), \tag{3.37}$$

this choice guarantees, that, if the convective term has a strong influence in the equation, the Donor-Cell schema avoids the oscillations.

Remark 3.2.6
The staggered grid is necessary to reduce oscillations. We have shown an example for this oscillations in the Example 3.2.4. But in the flow simulation this is essential, because otherwise there exist solutions of the simulation without any physical correctness. An example is given in the book of Griebel, Dornseifer and Neunhoeffer [MG95], in the Section 3.1.2.

At next we consider the discretisation of the divergence term. This is the second equation of the Navier-Stokes equations (3.21) and it is called the continuity equation. We realize the discretisation by using central differences with the stepsize $\frac{\delta x_\ell}{2}$

$$\begin{aligned}[\mathbf{div}\, u]_{i,j,k} &= \left[\frac{\partial u_1}{\partial x_3}\right]_{i,j,k} + \left[\frac{\partial u_2}{\partial x_2}\right]_{i,j,k} + \left[\frac{\partial u_3}{\partial x_3}\right]_{i,j,k} \\ &= \frac{u_{i,j,k,1} - u_{i-1,j,k,1}}{\delta x_1} + \frac{u_{i,j,k,2} - u_{i,j-1,k,2}}{\delta x_2} + \frac{u_{i,j,k,3} - u_{i,j,k-1,3}}{\delta x_3}\end{aligned} \tag{3.38}$$

Remark 3.2.7
Below we use the notions:
$\left[\frac{\partial(u_\ell u_m)}{\partial x_n}\right]_{i,j,k,\ell}$ *for the convective terms,*
$[\Delta u]_{i,j,k,\ell}$ *for the diffusion terms and*
$[\mathbf{div}\, u]_{i,j,k}$ *for the divergence term,*
instead of the full discrete notions, which are given above.

Boundary conditions

Here we consider the different kinds of the boundary conditions, and their realization with finite differences and a staggered grid.

(i) **Slip-Boundary-Condtions:**
That means the velocity is constant zero on this boundary, in the equations this is given by the following form:

$$\begin{aligned} u_{i,j,k,1} &= 0, \\ u_{i,j,k,2} &= 0, \\ u_{i,j,k,3} &= 0, \end{aligned} \tag{3.39}$$

for all $u_{i,j,k} = (u_{i,j,k,1}, u_{i,j,k,2}, u_{i,j,k,3})$ on the boundary of the discrete domain $\Omega_{discrete}$, on which the slip conditions holds.
By the staggered grid follows, that we have not all velocity values on each boundary. On this points, we get the value u_b on the boundary by a linear averaging of the velocity u_{in} inside the domain and u_{out} outside of the domain. With this follows[2]:

$$u_b = \frac{u_{out} + u_{in}}{2} \stackrel{!}{=} 0, \tag{3.40}$$

and this is equivalent to

$$u_{out} = -u_{in}. \tag{3.41}$$

[2]The variables u_b, u_{in}, u_{out} are scalar values and the corresponding entry of the full velocity vector.

The Equations (3.40) and (3.41) imply the following conditions:

$$\left.\begin{array}{ll} u_{i,j,0,1} = -u_{i,j,1,1}, & u_{i,j,k_{max}+1,1} = -u_{i,j,k_{max},1} \\ u_{i,j,0,2} = -u_{i,j,1,2}, & u_{i,j,k_{max}+1,2} = -u_{i,j,k_{max},2} \end{array}\right\} \quad i = 1,\ldots,i_{max} \quad j = 1,\ldots,j_{max}$$
$$\left.\begin{array}{ll} u_{i,0,k,1} = -u_{i,1,k,1}, & u_{i,j_{max}+1,k,1} = -u_{i,j_{max},k,1} \\ u_{i,0,k,3} = -u_{i,1,k,3}, & u_{i,j_{max}+1,k,3} = -u_{i,j_{max},k,3} \end{array}\right\} \quad i = 1,\ldots,i_{max} \quad k = 1,\ldots,k_{max} \tag{3.42}$$
$$\left.\begin{array}{ll} u_{0,j,k,2} = -u_{1,j,k,2}, & u_{i_{max}+1,j,k,2} = -u_{i_{max},j,k,2} \\ u_{0,j,k,3} = -u_{1,j,k,3}, & u_{i_{max}+1,j,k,3} = -u_{i_{max},j,k,3} \end{array}\right\} \quad j = 1,\ldots,j_{max} \quad k = 1,\ldots,k_{max},$$

if there are slip boundary conditions. The other components follow directly from the Equation (3.39).

(ii) **Non-Slip-Boundary-Condtions:**
That means , that the velocity in direction of the normal vector to the boundary is zero, and parallel to the boundary the velocity has no friction.
In the normal vector direction we get the following condition for the velocity:

$$\begin{array}{llll} u_{i,j,0,3} = u_{i,j,k_{max},3} = 0 & i = 1,\ldots,i_{max} & j = 1,\ldots,j_{max} \\ u_{i,0,k,2} = u_{i,j_{max},k,2} = 0 & i = 1,\ldots,i_{max} & k = 1,\ldots,k_{max} \\ u_{0,j,k,1} = u_{i_{max},j,k,1} = 0 & j = 1,\ldots,j_{max} & k = 1,\ldots,k_{max}, \end{array} \tag{3.43}$$

if there are non-slip boundary conditions.
With the condition, that the normal derivative of the tangential velocity is zero, i.e. $\frac{\partial u_{tangential}}{\partial \nu} = 0$, follows by using the discrete derivative $\frac{u_{in} - u_{out}}{\delta} \stackrel{!}{=} 0$:

$$u_{in} = u_{out}, \tag{3.44}$$

if u_{in} is the velocity inside of the boundary and u_{out} is the velocity outside of the boundary. The Equation (3.44) implies:

$$\left.\begin{array}{ll} u_{i,j,0,1} = u_{i,j,1,1}, & u_{i,j,k_{max}+1,1} = u_{i,j,k_{max},1} \\ u_{i,j,0,2} = u_{i,j,1,2}, & u_{i,j,k_{max}+1,2} = u_{i,j,k_{max},2} \end{array}\right\} \quad i = 1,\ldots,i_{max} \quad j = 1,\ldots,j_{max}$$
$$\left.\begin{array}{ll} u_{i,0,k,1} = u_{i,1,k,1}, & u_{i,j_{max}+1,k,1} = u_{i,j_{max},k,1} \\ u_{i,0,k,3} = u_{i,1,k,3}, & u_{i,j_{max}+1,k,3} = u_{i,j_{max},k,3} \end{array}\right\} \quad i = 1,\ldots,i_{max} \quad k = 1,\ldots,k_{max} \tag{3.45}$$
$$\left.\begin{array}{ll} u_{0,j,k,2} = u_{1,j,k,2}, & u_{i_{max}+1,j,k,2} = u_{i_{max},j,k,2} \\ u_{0,j,k,3} = u_{1,j,k,3}, & u_{i_{max}+1,j,k,3} = u_{i_{max},j,k,3} \end{array}\right\} \quad j = 1,\ldots,j_{max} \quad k = 1,\ldots,k_{max},$$

if there are non-slip-boundary conditions. The other components follow directly from the Equation (3.43).

(iii) **Inflow-Boundary-Condtions:**
That means, that the velocity on this boundary is explicitly given. For the velocity normal to the boundary, we can set the velocity value directly. For the tangential velocities, we do this similar to the averaging in the Equation (3.40). With this follows:

$$u_{inflow} = \frac{u_{in} + u_{out}}{2}, \tag{3.46}$$

(if u_{in} and u_{out} are defined as in slip or non-slip boundary conditions). This implies

$$u_{out} = \frac{u_{inflow} - u_{in}}{2} \tag{3.47}$$

for each of both tangential velocity components.

(iv) **Outflow-Boundary-Condtions:**
That means, that the derivative of the velocity is in direction of the normal vector zero. This is realizable by setting the function values of the velocity outside of the domain on the value of the nearest non boundary value. In equation formulation this is given with:

$$\left.\begin{array}{ll} u_{i,j,0,1} = u_{i,j,1,1}, & u_{i,j,k_{max}+1,1} = u_{i,j,k_{max},1} \\ u_{i,j,0,2} = u_{i,j,1,2}, & u_{i,j,k_{max}+1,2} = u_{i,j,k_{max},2} \\ u_{i,j,0,3} = u_{i,j,1,3}, & u_{i,j,k_{max}+1,3} = u_{i,j,k_{max},3} \end{array}\right\} \begin{array}{l} i = 1,\ldots,i_{max} \\ j = 1,\ldots,j_{max} \end{array}$$
$$\left.\begin{array}{ll} u_{i,0,k,1} = u_{i,1,k,1}, & u_{i,j_{max}+1,k,1} = u_{i,j_{max},k,1} \\ u_{i,0,k,1} = u_{i,1,k,2}, & u_{i,j_{max}+1,k,2} = u_{i,j_{max},k,2} \\ u_{i,0,k,3} = u_{i,1,k,3}, & u_{i,j_{max}+1,k,3} = u_{i,j_{max},k,3} \end{array}\right\} \begin{array}{l} i = 1,\ldots,i_{max} \\ k = 1,\ldots,k_{max} \end{array} \tag{3.48}$$
$$\left.\begin{array}{ll} u_{0,j,k,1} = u_{1,j,k,1}, & u_{i_{max}+1,j,k,1} = u_{i_{max},j,k,1} \\ u_{0,j,k,2} = u_{1,j,k,2}, & u_{i_{max}+1,j,k,2} = u_{i_{max},j,k,2} \\ u_{0,j,k,3} = u_{1,j,k,3}, & u_{i_{max}+1,j,k,3} = u_{i_{max},j,k,3} \end{array}\right\} \begin{array}{l} j = 1,\ldots,j_{max} \\ k = 1,\ldots,k_{max}, \end{array}$$

if there are outflow conditions.

(v) **Periodic-Boundary-Condtions:**
That means, that the velocity values of two boundaries in the opposite are the same. But here is a small difference to the continuous model. The discrete one is larger than the periodic interval at the stepsize of the discretisation δ in this direction. The reason is, that we set all values from the one side to the same of the opposite.
In a formula formulation follows:

$$\left.\begin{array}{ll} u_{i,j,0,1} = u_{i,j,k_{max}-1,1}, & u_{i,j,k_{max},1} = u_{i,j,1,1} \\ u_{i,j,0,2} = u_{i,j,k_{max}-1,2}, & u_{i,j,k_{max},2} = u_{i,j,1,2} \\ u_{i,j,0,3} = u_{i,j,k_{max}-1,3}, & u_{i,j,k_{max},3} = u_{i,j,1,3} \\ p_{i,j,0} = u_{i,j,k_{max}-1}, & u_{i,j,k_{max}} = u_{i,j,1} \\ t_{i,j,0} = u_{i,j,k_{max}-1}, & u_{i,j,k_{max}} = u_{i,j,1} \end{array}\right\} \begin{array}{l} i = 1,\ldots,i_{max} \\ j = 1,\ldots,j_{max}, \end{array} \tag{3.49}$$

if there are periodic boundary conditions. For the other both boundaries it is analog.

Time discretisation

Now we introduce one method for the time discretisation, here we consider only the explicit Euler method. It is clear, that this a very simple method and has not a so good approximation quality. But for an explanation it is well applicative.
When we consider an initial value problem:

$$\frac{\partial u}{\partial t} = f(t,u). \text{ and } u(t_0) = u_0. \tag{3.50}$$

The Euler-method is basically given in the following form:

$$u_{k+1} = u_k + hf(t,u), \tag{3.51}$$

for $t_k = t_0 + kh$ and $k = 1, 2 \ldots$.
We start with the time discretisation of the $\frac{\partial u}{\partial t}$ of the momentum equation. With the Euler-method and (3.21) follows:

$$u_\ell^{(n+1)} = u_\ell^{(n)} + \delta t \left[\frac{1}{Re}\Delta u - ((u\cdot\nabla)(u))_\ell + f_\ell - \frac{\partial p}{\partial x_\ell}\right] \tag{3.52}$$

for the ℓ's component of the velocity vector u. In the following we use the short notion:

$$F_\ell^{(n)} = u_\ell^{(n)} + \delta t \left[\frac{1}{Re} \Delta u^{(n)} - \left((u^{(n)} \cdot \nabla)(u^{(n)}) \right)_\ell + f_\ell \right], \tag{3.53}$$

with the explicit Euler method follows, that we consider the velocity at the time step n. But we consider the pressure at the time step $n+1$. So we get the full time discreticised momentum equation in the following form:

$$u_\ell^{(n+1)} = F_\ell^{(n)} - \delta t \frac{\partial p^{n+1}}{\partial x_\ell}, \tag{3.54}$$

this kind of time discretisation is called *explicit* in the velocity and *implicit* in the pressure. At next we compute with the continuity equation the pressure.

$$0 \stackrel{!}{=} \sum_{\ell=1}^{3} \frac{\partial u_\ell^{(n+1)}}{\partial x_\ell} = \sum_{\ell=1}^{3} \frac{\partial F_\ell^{(n)}}{\partial x_\ell} - \delta t \underbrace{\sum_{\ell=1}^{3} \frac{\partial^2 p^{(n+1)}}{\partial x_\ell^2}}_{=\Delta p^{(n+1)}}, \tag{3.55}$$

this Equation is equivalent to the Poisson problem for the pressure:

$$\Delta p^{(n+1)} = \frac{1}{\delta t} \sum_{\ell=1}^{3} \frac{\partial F_\ell^{(n)}}{\partial x_\ell}. \tag{3.56}$$

Now we need boundary condition to solve the Poisson problem (3.56). The boundary condition results from the multiplication of the time discrete momentum equation (3.54) with the outside normal vector to the boundary of the domain. This implies the following formula:

$$\nabla p^{(n+1)} \cdot \nu = \sum_{\ell=1}^{3} \frac{\partial p^{(n+1)}}{\partial x_\ell} \cdot \nu_\ell = \frac{1}{\delta t} \left(\sum_{\ell=1}^{3} (-u^{(n+1)} + F_\ell^{(n)}) \nu_\ell \right). \tag{3.57}$$

This method is called Chorin-Projection-Method.

Momentum-Equation discretisation

The time and place discretised momentum equation follows now directly by using the time discretisation and combining this with the place discrete components, this implies:

$$F_{i,j,k,\ell}^{(n)} = u_{i,j,k,\ell}^{(n)} + \delta t \left(\frac{1}{Re} \left[\Delta u^{(n)} \right]_{i,j,k,\ell} - \left[(u^{(n)} \cdot \nabla)(u^{(n)}) \right]_{i,j,k,\ell} + f_\ell \right), \tag{3.58}$$

and with this equation follows

$$u_{i,j,k,\ell}^{(n+1)} = F_{i,j,k,\ell}^{(n)} - \frac{\delta t}{\delta x_l} (p_{i+\delta_{1,\ell},j+\delta_{2,\ell},k+\delta_{3,\ell},\ell}^{(n+1)} - p_{i,j,k,\ell}^{(n+1)}), \tag{3.59}$$

for $i = 1, \cdots, i_{max}$, $j = 1, \cdots, j_{max}$, $k = 1, \cdots, k_{max}$ and $\ell = 1, 2, 3$.

Pressure-Equation discretisation

We discretize the Laplace operator for the presure $p_{i,j,k}$ in the same way like the Laplace operator for the velocity $u_{i,j,k,\ell}$ in the Equation (3.35).

$$[\Delta p]_{i,j,k,\ell} = \frac{1}{\delta t} \left(\sum_{\ell=1}^{3} \frac{F_{i,j,k,\ell}^{(n)} - F_{i-\delta_{1,\ell},j-\delta_{2,\ell},k-\delta_{3,\ell},\ell}^{(n)}}{\delta x_\ell} \right), \tag{3.60}$$

for $i = 1, \cdots, i_{max}$, $j = 1, \cdots, j_{max}$, $k = 1, \cdots, k_{max}$ and $\ell = 1, 2, 3$. On the boundary we use the following values:

$$\begin{array}{llll}
p_{0,j,k}, & p_{i_{max}+1,j,k}, & j = 1, \cdots, j_{max}, & k = 1, \cdots, k_{max} \\
p_{i,0,k}, & p_{i,j_{max}+1,k}, & i = 1, \cdots, j_{max}, & k = 1, \cdots, k_{max} \\
p_{i,j,0}, & p_{i,j,k_{max}+1}, & i = 1, \cdots, j_{max}, & j = 1, \cdots, k_{max} \\
F_{0,j,k,1}, & F_{i_{max}+1,j,k,1}, & j = 1, \cdots, j_{max}, & k = 1, \cdots, k_{max} \\
F_{i,0,k,2}, & F_{i,j_{max}+1,k,2}, & i = 1, \cdots, j_{max}, & k = 1, \cdots, k_{max} \\
F_{i,j,0,3}, & F_{i,j,k_{max}+1,3}, & i = 1, \cdots, j_{max}, & j = 1, \cdots, k_{max}
\end{array}$$

These boundary values we have not considered yet. This we catch up now. We consider exemplarily the bottom wall, i.e. $\nu = (0, 0, -1)^T$. With the boundary condition given in the Equation (3.57) follows:

$$\frac{p_{i,j,0}^{(n+1)} - p_{i,j,1}^{(n+1)}}{\delta x_3} = \frac{1}{\delta t}\left(u_{i,j,0,3}^{(n+1)} - F_{i,j,0,3}^{(n)}\right). \tag{3.61}$$

Combining the Equation (3.60) and (3.61), this implies:

$$\begin{aligned}
&\frac{p_{i+1,j,1}^{(n+1)} - 2p_{i-1,j,1}^{(n+1)} + p_{i,j,1}^{(n+1)}}{(\delta x_1)^2} + \frac{p_{i,j+1,1}^{(n+1)} - 2p_{i,j-1,1}^{(n+1)} + p_{i,j,1}^{(n+1)}}{(\delta x_2)^2} + \frac{p_{2,j,k}^{(n+1)} - p_{1,j,k}^{(n+1)}}{(\delta x_3)^2} \\
&\quad = \frac{1}{\delta t}\left(\frac{F_{i,j,1,1}^{(n)} - F_{i-1,j,1,1}^{(n)}}{\delta x_1} + \frac{F_{i,j,1,2}^{(n)} - F_{i,j-1,1,2}^{(n)}}{\delta x_2} + \frac{F_{i,j,1,3}^{(n)} - u_{i,j,0,3}^{(n+1)}}{\delta x_3}\right).
\end{aligned}$$

We see, that this equation is independent of $F_{i,j,0,3}$, so we can choose $F_{i,j,0,3}$ equal to $u_{i,j,0,3}^{(n+1)}$. This implies:

$$p_{0,j,k}^{(n+1)} = p_{1,j,k}^{(n+1)}.$$

Analogically follow the other boundary values:

$$\begin{array}{llll}
p_{0,j,k} = p_{1,j,k}, & p_{i_{max}+1,j,k} = p_{i_{max},j,k}, & j = 1, \cdots, j_{max}, & k = 1, \cdots, k_{max} \\
p_{i,0,k} = p_{i,1,k}, & p_{i,j_{max}+1,k} = p_{i,j_{max},k}, & i = 1, \cdots, j_{max}, & k = 1, \cdots, k_{max} \\
p_{i,j,0} = p_{i,j,1}, & p_{i,j,k_{max}+1} = p_{i,j,k_{max}}, & i = 1, \cdots, j_{max}, & j = 1, \cdots, k_{max} \\
F_{0,j,k,1} = u_{0,j,k,1}, & F_{i_{max},j,k,1} = u_{i_{max},j,k,1}, & j = 1, \cdots, j_{max}, & k = 1, \cdots, k_{max} \\
F_{i,0,k,2} = u_{i,0,k,2}, & F_{i,j_{max},k,2} = u_{i,j_{max},k,2}, & i = 1, \cdots, j_{max}, & k = 1, \cdots, k_{max} \\
F_{i,j,0,3} = u_{i,j,0,3}, & F_{i,j,k_{max},3} = u_{i,j,k_{max},3}, & i = 1, \cdots, j_{max}, & j = 1, \cdots, k_{max}
\end{array}$$

With the boundary values and the Equation (3.35) follows:

$$\begin{aligned}
&\frac{\epsilon_{i,1}^1\left(p_{i+1,j,k}^{(n+1)} - p_{i,j,k}^{(n+1)}\right) - \epsilon_{i,1}^{-1}\left(p_{i,j,k}^{(n+1)} - p_{i-1,j,k}^{(n+1)}\right)}{(\delta x_1)^2} \\
&+\frac{\epsilon_{j,2}^1\left(p_{i,j+1,k}^{(n+1)} - p_{i,j,k}^{(n+1)}\right) - \epsilon_{j,2}^{-1}\left(p_{i,j,k}^{(n+1)} - p_{i,j-1,k}^{(n+1)}\right)}{(\delta x_2)^2} \\
&+\frac{\epsilon_{k,3}^1\left(p_{i,j,k+1}^{(n+1)} - p_{i,j,k}^{(n+1)}\right) - \epsilon_{k,3}^{-1}\left(p_{i,j,k}^{(n+1)} - p_{i,j,k-1}^{(n+1)}\right)}{(\delta x_3)^2} \\
&\qquad = \frac{1}{\delta t}\left(\sum_{\ell=1}^{3} \frac{F_{i,j,k,\ell}^{(n)} - F_{i-\delta_{1,\ell},j-\delta_{2,\ell},k-\delta_{3,\ell},\ell}^{(n)}}{\delta x_\ell}\right),
\end{aligned} \tag{3.62}$$

for $i = 1, \ldots, i_{max}$, $j = 1, \ldots, j_{max}$ and $k = 1, \ldots, k_{max}$ and the Parameters ϵ are:

$$\epsilon^{-1}_{n,\ell} := \begin{cases} 0, & n = 1 \\ 1, & n > 1 \end{cases}, \quad \epsilon^{1}_{n,\ell} := \begin{cases} 1, & n < n_{max} \\ 0, & n = n_{max} \end{cases}, \begin{array}{lll} n = i & if & \ell = 1 \\ n = j & if & \ell = 2. \\ n = k & if & \ell = 3 \end{array}$$

All in all we get a linear system of equations with $i_{max} j_{max} k_{max}$ degrees of freedom. Normally this system is solved by a iterative method, for example the SOR-method.

Stability Condition

Here we give the *Courant-Friedrichs-Lewy conditions* also known as *CFL- conditions*:

$$\delta t := \tau \min \left(\frac{Re}{2} \left(\frac{1}{\delta x_1^2} + \frac{1}{\delta x_2^2} + \frac{1}{\delta x_3^2} \right)^{-1}, \frac{\delta x_1}{|u_{1,max}|}, \frac{\delta x_2}{|u_{2,max}|}, \frac{\delta x_3}{|u_{3,max}|} \right), \tag{3.63}$$

with $\tau \in (0, 1]$ a secure factor. This choice of the time step δt assures, that the simulation is stable.

Algorithm

Here, we present a finite differences based algorithm in order to simulate the Navier-Stokes equations.
This is the algorithm to simulate the Navier-Stokes equations with finite differences.

Algorithm 3.2.8 (Finite difference Navier-Stokes solver)

init u *with start values*
init p *with start values*
set boundary values
for (t=0,n=0;$t < T$*;t+=*δt*, n++)*
  set δt *according to (3.63)*
  set boundary values of $u^{(n)}$
  calculate F^n_ℓ *for* $\ell = 1, 2, 3$ *according to (3.58)*
   for (it=0;$it < it_{max}$ *and* $error > tol$*;it++)*
    compute iteration step of (3.62) (e.g. with the SOR-method)
    compute err(p)
   end
  calculate $u^{(n)}$ *according to (3.59)*
  output();
  visual();
end

Remark 3.2.9
The here presented algorithm is realized for 2D problems in the NaSt2D[3]
program. But it is with more technical complexity also in 3D possible.[4]

[3] A implementation of this program NaSt2d is available on the homepage of Prof. Dr. M. Griebel under: http://wissrech.ins.uni-bonn.de/research/projects/NaSt2D/index.html

[4] A 3D implementation is available on the homepage of Prof. Dr. M. Griebel under: http://wissrech.ins.uni-bonn.de/research/projects/NaSt3DGP/download.htm .

In the next section we give an introduction into the finite volume method and the simulation of flow with this method.

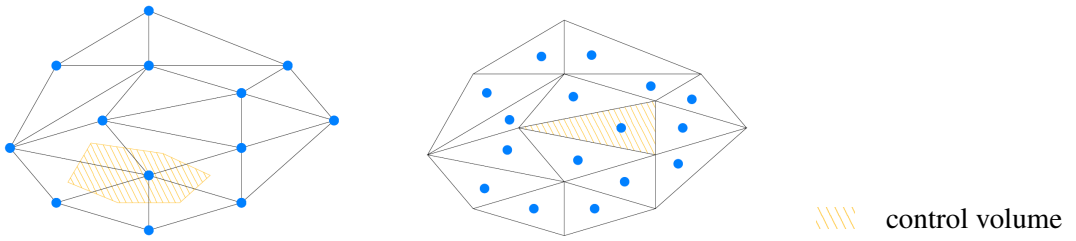
3.2.2 Finite volume method

This method is a numerical method to solve PDE's with a conservation law. A common notion for the **finite volume method** is **FVM**. The structure of this subsection is the following; we start with the mathematical basics, which are needed for the FVM and the main idea of this method, at next, we illustrate the method on an explicit example, then, we consider the different parts of the method more detailed, at last, we consider the basics for the discretisation of the Navier-Stokes equations with the FVM.
This section is based on the book of Freziger and Perić [Fre97], the book of Knaberner and Angermann [Kna00], the lecture note of Grundmann [Gru04] and the paper of Barth and Ohlberger [TB04].

Definition 3.2.10 (Control volume tessellation)
Let $\Omega \subset \mathbb{R}^n$ be a domain, $\mathcal{T} := \{T_i : i = 1, \dots N\}$ is called a control volume tessellation, if

1. *$T_i \subset \Omega$ and T_i is an open, connected domain, the boundary is piecewise given with polynomials , without slots.*
2. *$T_i \cap T_j = \emptyset \ \forall i \neq j$.*
3. *$\bigcup_{i=1}^{N} \overline{T_i} = \overline{\Omega}$.*

hold. There exist two typical kinds of control volume tessellation; the **cell-centered** *and the* **vertex-centered**. *both kinds are shown in the Figure 3.7. A common shortcut for the control volume is* **CV**.



a. Vertex-centered b. Cell-centered

Figure 3.7: Two kinds of control volume tessellation

The idea of the finite volume methods is based on the physical conservation law. That means, that we start with a normal PDE and we rewrite this equation into the integral formulation. And we apply the Gauss' theorem and the Theorem of Stokes on the integral notion.

Example 3.2.11
Here we consider the ℓ-th component of the momentum equation of the incompressible Navier-Stokes equations (3.21). We recall this formula:

$$\frac{\partial}{\partial t}(\rho u_\ell) + (u \cdot \nabla)(\rho u_\ell) + \frac{\partial p}{\partial x_\ell} = \mu \Delta u_\ell + \rho f_\ell.$$

Now we integrate this equation over a control volume T, which satisfies the conditions of the Definition 3.2.10, this implies

$$\int_T \frac{\partial}{\partial t}(\rho u_\ell)\, d\Omega + \int_T (u \cdot \nabla)(\rho u_\ell)\, d\Omega + \int_T \frac{\partial p}{\partial x_\ell}\, d\Omega = \int_T \mu \Delta u_\ell\, d\Omega + \int_T \rho f_\ell\, d\Omega. \tag{3.64}$$

Now we apply the Gauss'theorems on the Equation (3.64) with this and the $S := \delta T$ follow:

$$\frac{\partial}{\partial t}\int_T (\rho u_\ell)\, d\Omega + \int_S \rho u_\ell u \cdot \nu\, dS + \int_S p_\ell\, dS = \int_S \mu(\nabla u_\ell) \cdot \nu\, dS + \int_T \rho f_\ell\, d\Omega. \tag{3.65}$$

On this example, we have seen how to rewrite a normal PDE into the integral version and the application of the Gauss'theorems.
At next we consider the application of the finte volume method on an explicit problem.

Example 3.2.12
Here we consider the energy transport in a given velocity field, the equation is:

$$\frac{\partial(\rho\phi)}{\partial t} + \mathbf{div}(\rho\phi u) = \mathbf{div}\left(\frac{\mu}{Pr}\nabla\phi\right) + q_\phi, \tag{3.66}$$

this equation is equivalent to

$$\frac{\partial}{\partial t}\int_\Omega \rho\phi\, d\Omega + \int_S \rho\phi u \cdot \nu\, dS\nu = \int_S \Gamma\nabla\phi \cdot \nu\, dS + \int_\Omega q_\phi\, d\Omega, \tag{3.67}$$

if $\frac{\mu}{Pr} := \Gamma$.
In the following we consider the stationary case, i.e $\frac{\partial(\rho\phi)}{\partial t} = 0$. This is equivalent to $\frac{\partial}{\partial t}\int_\Omega \rho\phi\, d\Omega = 0$, and the external force q_ϕ is also zero. This implies, that we can write the problem in the following form:

$$\underbrace{\int_S \rho\phi u \cdot \nu\, dS\nu}_{=:F^c} = \underbrace{\int_S \Gamma\nabla\phi \cdot \nu\, dS}_{=:F^d}. \tag{3.68}$$

We solve this problem on the two dimensional case on the domain $\Omega = [0,1] \times [0,1]$ with the following boundary conditions:

- $\phi(x,y) = 0$ *on* $y = 1$ *and* $x \in [0,1]$ *(north);*
- $\phi(x,y) = 1 - y$ *on* $x = 0$ *and* $y \in [0,1]$ *(west);*
- $\frac{\partial\phi(x,y)}{\partial\nu} = 0$ *on* $y = 0$ *and* $x \in [0,1]$ *(south);*
- $\nabla\phi(x,y)^T u = 0$ *on* $x = 1$ *and* $y \in [0,1]$ *(east).*

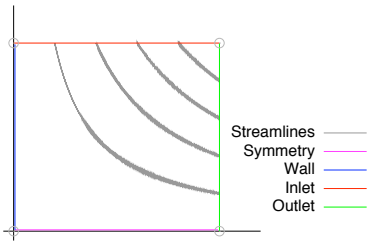


Figure 3.8: Exapmle 3.2.12

Where the given velocity field is $u = \begin{pmatrix} u_x \\ u_z \end{pmatrix} = \begin{pmatrix} x \\ -y \end{pmatrix}$ *and the parameter is* $\rho = 1$. *In the Figure 3.8 the domain is visualized with the boundary conditions.*

Now we show the discretisation of this problem. At first we decompose the integrals F^c *and* F^d *in the following way:*

$$F^c = F^c_{north} + F^c_{west} + F^c_{south} + F^c_{east} \tag{3.69}$$

$$F^d = F^d_{north} + F^d_{west} + F^d_{south} + F^d_{east} \tag{3.70}$$

At next we consider the parts north, west, and so on, of the integrals F^c *and* F^d*:*

North:

At first we consider the convective term:

$$F^c_{north} = \int_{S_{north}} \rho\phi u \cdot \nu \, dS \approx \underbrace{\int_{S_{north}} \rho u \cdot \nu \, dS}_{=:\dot{m}_{north}} \phi_N, \tag{3.71}$$

and $\dot{m}_{north}$ *represents the flux of the mass through the north face. We can compute this mass flux in the following form:*

$$\dot{m}_{north} = (\rho u_y)_{north} \, \delta x. \tag{3.72}$$

The Equation (3.72) is exact, if the velocity $(u_x)_{north}$ *is constant along the north face. This is satisfied in our case. So we get for the flux approximation:*

$$F^c_{north} \approx \begin{cases} \max\{\dot{m}_{north}, 0\}\phi_{center} + \min\{\dot{m}_{north}, 0\}\phi_{north} & for\ UDS^5, \\ \dot{m}_{north}(1 - \lambda_{north})\phi_{north} + \dot{m}_{north}\lambda_{north}\phi_{north} & for\ CDS^6. \end{cases} \tag{3.73}$$

Now we consider the diffusion term:

$$\begin{aligned} F^d_{north} &= \int_{S_{north}} \Gamma \nabla\phi \cdot \nu \, dS \approx \left(\Gamma \frac{\partial \phi}{\partial y}\right)_{north} \delta x \\ &= \frac{\Gamma \delta x}{y_{north} - y_{center}} (\phi_{north} - \phi_{center}). \end{aligned} \tag{3.74}$$

Note, that $y_{north} = \frac{y_{i+1}+y_i}{2}$ *and* $y_{center} = \frac{y_{i-1}+y_i}{2}$.

For the other faces of the control volume, we only present the solutions.

[5] UDS means the upwind differences scheme. We will explain it later.

[6] CDS means the central differences scheme. We will explain it later.

West:
Convective term:

$$F^c_{west} \approx \begin{cases} \max\{\dot{m}_{west},0\}\phi_{center} + \min\{\dot{m}_{west},0\}\phi_{west} & for\ UDS, \\ \dot{m}_{west}(1-\lambda_{west})\phi_{west} + \dot{m}_{west}\lambda_{west}\phi_{west} & for\ CDS. \end{cases}$$

and $\dot{m}_{west}$ is given by

$$\dot{m}_{west} = -\left(\rho u_x\right)_{west}\delta y.$$

Diffusion term:

$$F^d_{west} \approx \frac{\Gamma\delta y}{x_{center}-x_{west}}(\phi_{center}-\phi_{west}),$$

with $x_{west} = \frac{x_{i-1}+x_i}{2}$ and $x_{center} = \frac{x_{i+1}+x_i}{2}$.

South:
Convective term:

$$F^c_{south} \approx \begin{cases} \max\{\dot{m}_{south},0\}\phi_{center} + \min\{\dot{m}_{north},0\}\phi_{south} & for\ UDS, \\ \dot{m}_{south}(1-\lambda_{south})\phi_{south} + \dot{m}_{south}\lambda_{south}\phi_{south} & for\ CDS. \end{cases},$$

and $\dot{m}_{south}$ is given by

$$\dot{m}_{south} \approx -\left(\rho u_y\right)_{south}\delta x.$$

Diffusion term:

$$F^d_{south} \approx \frac{\Gamma\delta x}{y_{center}-y_{south}}(\phi_{center}-\phi_{south}),$$

with $y_{south} = \frac{y_{i-1}+y_i}{2}$ and $y_{center} = \frac{y_{i+1}+y_i}{2}$.

East:
Convective term:

$$F^c_{east} \approx \begin{cases} \max\{\dot{m}_{east},0\}\phi_{center} + \min\{\dot{m}_{east},0\}\phi_{east} & for\ UDS, \\ \dot{m}_{east}(1-\lambda_{east})\phi_{east} + \dot{m}_{east}\lambda_{east}\phi_{east} & for\ CDS. \end{cases},$$

and $\dot{m}_{east}$ is given by

$$\dot{m}_{east} = \left(\rho u_x\right)_{east}\delta y.$$

Diffusion term:

$$F^d_{east} \approx \frac{\Gamma\delta y}{x_{east}-x_{center}}(\phi_{east}-\phi_{center}),$$

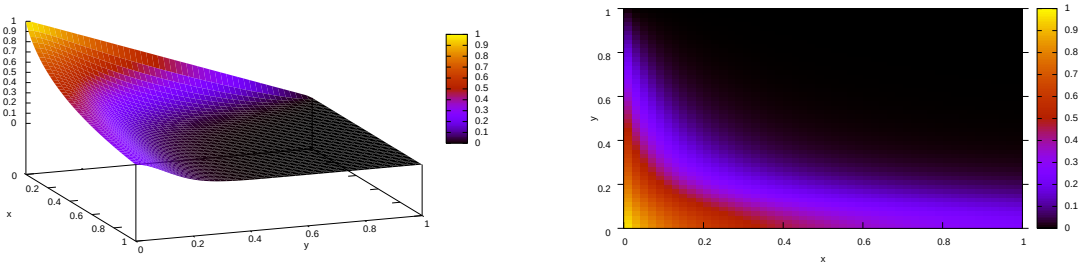
with $x_{east} = \frac{x_{i+1}+x_i}{2}$ and $x_{center} = \frac{y_{i-1}+y_i}{2}$.

Now we can compute the coeffencints of A^c and A^d:

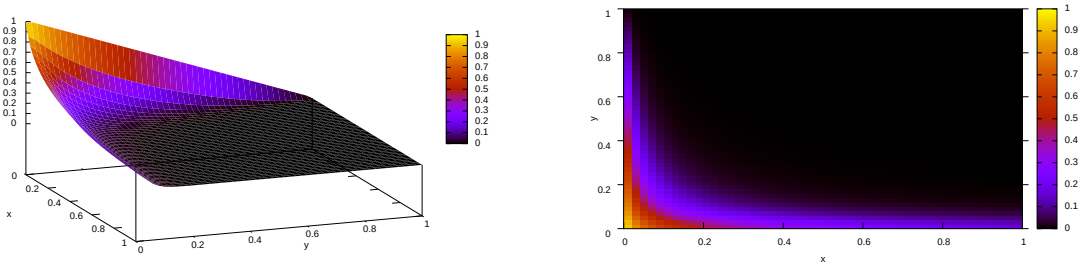
	F^c *UDS*		F^c *UDS*		F^d
A^c_{north}	$= \min(\dot{m}_{north},0)$	A^c_{north}	$= \dot{m}_{north}\lambda_{north}$	A^d_{north}	$= -\frac{\Gamma\delta x}{x_{north}-x_{center}}$
A^c_{west}	$= \min(\dot{m}_{west},0)$	A^c_{west}	$= \dot{m}_{west}\lambda_{west}$	A^d_{west}	$= -\frac{\Gamma\delta y}{x_{center}-x_{south}}$
A^c_{south}	$= \min(\dot{m}_{south},0)$	A^c_{south}	$= \dot{m}_{south}\lambda_{south}$	A^d_{south}	$= -\frac{\Gamma\delta x}{x_{center}-x_{south}}$
A^c_{east}	$= \min(\dot{m}_{east},0)$	A^c_{east}	$= \dot{m}_{east}\lambda_{east}$	A^d_{east}	$= -\frac{\Gamma\delta y}{x_{east}-x_{center}}$

With continuity coditions,

$$\dot{m}_{north} + \dot{m}_{west} + \dot{m}_{south} + \dot{m}_{east} = 0,$$



3.9a: 50×50 cells, $\rho = 1, \Gamma = 0.01$



3.9b: 50×50 cells, $\rho = 1, \Gamma = 0.001$

Figure 3.9: Approximation of the problem (3.68) with the UDC method.

follows the entry of A^c_{centre} *and* A^d_{centre}*:*

$$\begin{aligned} A^c_{centre} &= -(A^c_{north} + A^c_{west} + A^c_{south} + A^c_{east}) \\ A^d_{centre} &= -(A^d_{north} + A^d_{west} + A^d_{south} + A^d_{east}) \end{aligned}$$

Now we define:

$$A_\ell := A^c_\ell + A^d_\ell, \tag{3.75}$$

and ℓ *represents one of the indices* $north,\ west,\ south,\ east.$
The boundary conditions change the coeffencients at the cells next to the boundary in the following:

Bouandary faces:

North	$A_{north} = 0$	
West	$A_{west} = 0$	$Q = Q - A_{west} \cdot \phi(0, y)$
South	$A_{center} = A_{center} + A_{south}$	$A_{south} = 0$
East	$A_{center} = A_{center} + A_{east}$	$A_{north} = A_{north} + \frac{u_{y,e}\delta x}{2\delta y} A_{east}$
	$A_{south} = A_{south} - \frac{u_{y,e}\delta x}{2\delta y} A_{east}$	$A_{east} = 0$

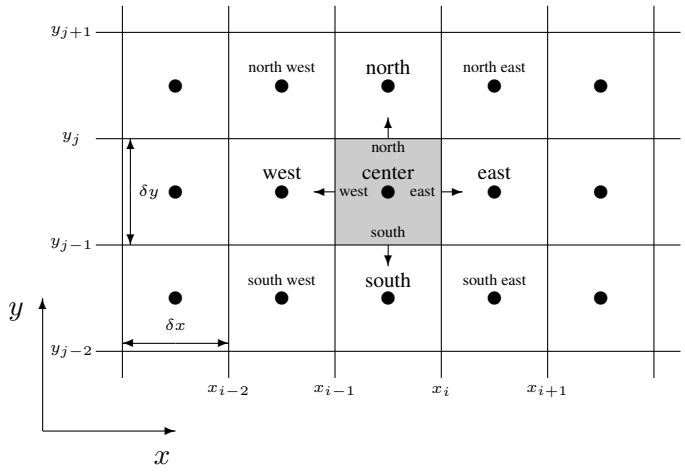


Figure 3.10: 2D tessellation with quadratic cells.

Bouandary vertexes:

North West	$A_{north} = 0$	$A_{west} = 0$
	$Q = Q - A_{west} \cdot \phi(0, y)$	
South West	$A_{center} = A_{center} + A_{south}$	$A_{west} = 0$
	$A_{south} = 0$	$Q = Q - A_{west} \cdot \phi(0, y)$
South East	$A_{north} = A_{north} + \frac{u_{y,e}\delta x}{\delta y} A_{east}$	$A_{center} = A_{center} + A_{south}$ $+ (1 - \frac{u_{y,e}\delta x}{\delta y}) A_{east}$
	$A_{south} = 0$	$A_{east} = 0$
North East	$A_{center} = A_{center} + A_{east}$	$A_{north} = 0$
	$A_{south} = A_{south} - \frac{u_{y,e}\delta x}{\delta y} A_{east}$	$A_{east} = 0.$

Now we can build the system of linear equations. By solving this system, we get the approximative solution of the problem (3.68) with the given boundary conditions. In the Figure 3.9 the solution of the problem is visualized with the UDC method on an equidistance grid with 50×50 *cells and the parameter* $\Gamma = 0.01$ *and* $\Gamma = 0.001$. *The realization in done in Fortran.*

Approximation of the surface integrals

Here we consider different approaches to approximate the surfaces integrals. In the following S is the surface of the control volume T. The surface of the CV is decomposable into a finite number of plane sub-faces, i.e. $S = \bigcup_{\ell \in \mathbb{S}1} S_\ell$ and the normal vector ν_ℓ on the sub-face S_ℓ is constant, where $\mathbb{S}$ represent the set of subsurface indexes.
With this follows, that the integral $\int_{S_\ell} F \, dS$ is writable in the following way:

$$\int_S f \, dS = \sum_{\ell \in \mathbb{S}} \int_{S_\ell} f \, dS, \tag{3.76}$$

where f is one integrand of the considered problem, e.g. the convection term of the Example 3.2.12 ($\rho\phi u \cdot \nu$). In the following we assume, that we have only a 2D case and the cells are all quadratic, because in this case it is demonstrative. For the 3D case or more complicated cell geometries it works on the same principle, but it is only more technical.

In the Figure 3.10, we show a schematic picture in order to illustrate the tessellation in the control volumes. To compute the Equation (3.76) exactly, we need the the integrand f everywhere on the surface S or on each subsurface S_ℓ. But this information is missing, we only

know the integrand on the center of the control volume. So we must approximate the integral. The simplest approximation is midpoint rule:

$$F_\ell = \int_{S_\ell} f \, dS = \bar{f}_\ell \delta S_\ell \approx f_\ell \delta S_\ell, \tag{3.77}$$

where $\bar{f}_\ell$ represents the mean-value of f_ℓ on the surface S_ℓ, and δS_ℓ stands for the length of S_ℓ. This is the simplest second order method. If the value of the integrand is unknown on the point ℓ, we must consider the other approximations for the integral. For more details about the surface integral look on the book of Ferziger and Perić [Fre97], Stoer [Sto99] or Schwarz [Sch97].

Approximation of the volume integrals

Now we consider methods to approximate the volume integral over the CV T, because in the transport term it is not possible to compute all terms with the surface integrals, e.g. the right hand side requires the volume integral. We consider an equation of the following form:

$$Q = \int_T q \, d\Omega = \bar{q} \Delta T \approx q_{center} \Delta T, \tag{3.78}$$

where $\bar{q}$ represents the mean-value of q in the CV T, q_{center} represents the function value in the middle of CV cell T, and ΔT represents the volume of the cell T. For a constant or linear q this approximation is exact. In the other cases the approximation is of second order. For more details see to the same reference books like the section before about *Approximation of the surface integrals*.

Interpolation Practices

In this section we present two methods for the interpolation of the integrals.

Upwind Interpolation

This method is also called **upwind differences scheme** the shortcut is **(UDS)**. The method approximates the first derivative of the unknown ϕ with backward or forward differences dependent on the velocity u. In detail this means exemplary in the 2D case for ϕ_N:

$$\phi_N := \begin{cases} \phi_{center} & if \quad (u \cdot \nu)_{north} > 0 \\ \phi_{north} & if \quad (u \cdot \nu)_{north} < 0. \end{cases} \tag{3.79}$$

This choice of the ϕ_N assures, that there never accure oscillations in the solution similar to the view in the Example (3.2.4). Considering the Taylor series, it is clear, that this is a first order scheme. For the other boundary of a cell, it goes on the same way.

Linear Interpolation

This interpolation idea is to use a similar approach to the central differences in the finite difference method, this implies the acronym **(CDS)** for **central differences scheme**. In detail this means exemplary in the 2D case for ϕ_N:

$$\phi_N = \phi_{north} \lambda_{north} + \phi_{center}(1 - \lambda_{north}), \tag{3.80}$$

where λ_{north} is the linear interpolation coefficient and it is define as:

$$\lambda_{north} := \frac{y_N - y_{center}}{y_{north} - y_{center}}, \tag{3.81}$$

and y_N is the y-coordinate of the variable ϕ_{north}, if we have quadratic cells and the ϕ are cell centered, as shown in the Figure 3.10. In the same way, it is possible for the other boundaries of a cell.

Remark 3.2.13
Here we have only represented two methods for the interpolation. For other methods or a more detailed consideration look on the book of Freziger and Perić [Fre97]. Other methods are: quadratic upwind interpolation (QUIK), higher-order schemes, linear upwind scheme (LUDS) and others.

Boundary conditions

In this subsection we consider the realization of the different **boundary conditions** with the finite element method. We consider the Dirichlet and the Neumann boundary conditions. Now we restrict us only on the 2D case with quadratic cells.

Dirichlet boundary condition:
In the following we consider the boundary cell T, which contacts the north boundary, and the function value on the north boundary is given by $\phi = f(x)$. On an interior cell we can formulate the problem in the following way:

$$A_{north}\phi_{north} + A_{west}\phi_{west} + A_{south}\phi_{south} + A_{east}\phi_{east} + A_{center}\phi_{center} = Q. \tag{3.82}$$

But on the north boundary cell T the value of the term ϕ_{north} is known. By this and by the Equation (3.82) follow:

$$A_{west}\phi_{west} + A_{south}\phi_{south} + A_{east}\phi_{east} + A_{center}\phi_{center} = Q - A_{north}\underbrace{\phi_{north}}_{=f}. \tag{3.83}$$

The Equation (3.83) realizes the boundary conditions. For the other kinds of boundary cells it works on the same way.

Neumann boundary condition:
In the following we consider the boundary cell T, which contacts the west boundary, and the derivative on the west boundary is given by $\nabla\phi = g(x, y)$. This implies, that in the discrete version

$$\begin{pmatrix} \frac{\phi_{center}-\phi_{west}}{\delta x} \\ \frac{\phi_{north}-\phi_{south}}{2\delta y} \end{pmatrix} = \begin{pmatrix} g_x(x,y) \\ g_y(x,y) \end{pmatrix}, \tag{3.84}$$

and this implies

$$\frac{\phi_{center} - \phi_{west}}{\delta x} + \frac{\phi_{north} - \phi_{south}}{2\delta y} = \underbrace{g_x(x,y) + g_y(x,y)}_{:=G(x,y)}. \tag{3.85}$$

This equation is equivalent to:

$$\phi_{west} = \left[\left(-G(x,y) + \frac{\phi_{north} - \phi_{south}}{2\delta y}\right)(\delta x) + \phi_{center}\right]. \tag{3.86}$$

The Equations (3.82) and (3.86) imply:

$$\left(A_{north} + A_{west}\frac{\delta x}{2\delta y}\right)\phi_{north} \tag{3.87}$$

$$+\left(A_{south} - A_{west}\frac{\delta x}{2\delta y}\right)\phi_{south} \quad +(A_{center} + A_{west})\phi_{center} \tag{3.88}$$

$$= Q + A_{west}G(x,y)(\delta x) \tag{3.89}$$

For the other kinds of boundary cells it works on the same way.

Discretisation of the Navier-Stokes equations with the finite volumes method

In the following we discretize the Navier-Stokes equations with the finite volume method. At first we write the Navier-Stokes equations (3.21) in the integral notion:

$$\int_T \frac{\partial}{\partial t}(\rho u_\ell)\,d\Omega + \int_T (u\cdot\nabla)(\rho u_\ell)\,d\Omega + \int_T \frac{\partial p}{\partial x_\ell}\,d\Omega = \int_T \mu\Delta u_\ell\,d\Omega + \int_T \rho f_\ell\,d\Omega.$$

We have also seen this form in the Example 3.2.11. At next we consider each part of this integral separatly and apply the Gauss'theorem, like in the Example 3.2.11, and give some remarks to the terms.

Diffusion term

This is the $(\mu\Delta u))_\ell$ part of the Equation (3.21) in the integral notion and by the application of the Gauss'theorem follows:

$$\int_T \mu\Delta u_\ell\,d\Omega = \int_T \mu\Big(\underbrace{\frac{\partial^2 u_\ell}{\partial x^2}}_{:=\frac{\partial}{\partial x}\psi_x} + \underbrace{\frac{\partial^2 u_\ell}{\partial y^2}}_{:=\frac{\partial}{\partial y}\psi_y}\Big)\,d\Omega = \underbrace{\int_T \mu\frac{\partial}{\partial x}\psi_x\,d\Omega}_{\substack{:=\int_S \mu\psi_x\,dS \\ =\int_S \mu\frac{\partial u_\ell}{\partial x}\,dS}} + \underbrace{\int_T \mu\frac{\partial}{\partial y}\psi_y\,d\Omega}_{\substack{:=\int_S \mu\psi_y\,dS \\ =\int_S \mu\frac{\partial u_\ell}{\partial y}\,dS}}$$

$$= \int_S \mu(\nabla u_\ell)\cdot\nu\,dS. \tag{3.90}$$

We can decompose the last term in the following way:

$$\int_S \mu(\nabla u_\ell)\cdot\nu\,dS = \int_{S_{north}} \mu\frac{\partial u_\ell}{\partial y}\,dS - \int_{S_{west}} \mu\frac{\partial u_\ell}{\partial x}\,dS - \int_{S_{south}} \mu\frac{\partial u_\ell}{\partial y}\,dS + \int_{S_{east}} \mu\frac{\partial u_\ell}{\partial x}\,dS \tag{3.91}$$

One possible discretisation is:

$$\int_{S_{north}} \mu\frac{\partial u_\ell}{\partial y}\,dS \approx \mu\left[\frac{\partial u_\ell}{\partial y}\right]\delta S_{north}, \tag{3.92}$$

where $\left[\frac{\partial u_\ell}{\partial y}\right]$ represents the discrete derivative and δS_{north} represents the length of the interval. And u_ℓ represents the velocity in x or y direction. For the other boundaries it goes on the same way.

Convective term

The conservation term is the $((u\cdot\nabla)(\rho u))_\ell$ part of the Equation (3.21) in the integral notion and the application of the Gauss'theorem follows:

$$\int_T (u\cdot\nabla)(\rho u_\ell)\,d\Omega = \int_T \left(\frac{\partial(\rho u_\ell u_x)}{\partial x} + \frac{\partial(\rho u_\ell u_y)}{\partial y}\right)\,d\Omega = \int_S \rho u_\ell u\cdot\nu\,dS \tag{3.93}$$

We can decompose the last term of the Equation (3.93) in the following way:

$$\begin{aligned}\int_S \rho u_\ell u \cdot \nu \, dS = \quad & \int_{S_{north}} \rho u_\ell u_{north} \, dS - \int_{S_{west}} \rho u_\ell u_{west} \, dS \\ & - \int_{S_{south}} \rho u_\ell u_{south} \, dS + \int_{S_{east}} \rho u_\ell u_{east} \, dS \end{aligned} \tag{3.94}$$

One possible discretisation is:

$$\int_{S_{north}} \rho u_\ell u_{north} \, dS = \rho u_\ell u_{north} \delta S_{north} \tag{3.95}$$

Pressure term
The pressure term is the $(\nabla p)_\ell$ part of the Equation (3.21) in the integral notion and by the application of the Gauss'theorem follows:

$$\int_T \frac{\partial p}{\partial x_\ell} \, d\Omega = \int_{S_\ell} p \, dS \tag{3.96}$$

We can decompose the last term of the Equation (3.96) in the following way:

$$\begin{aligned}\int_{S_\ell} p \, dS = \quad & \int_{S_{north}} p_{north} \, dS - \int_{S_{south}} p_{south} \, dS \quad if \; \ell = x \\ \int_{S_\ell} p \, dS = \quad & \int_{S_{east}} p_{east} \, dS - \int_{S_{west}} p_{west} \, dS \quad if \; \ell = y \end{aligned} \tag{3.97}$$

One possible discretisation is:

$$\int_{S_{north}} p_{north} \, dS = p_{north} \delta S_{north}, \tag{3.98}$$

where δS_{north} is the length of the interval.

Remark 3.2.14 (Derivative approximation)
The approximation of the derivative for the diffusion term is possible as in the Example 3.2.12 or a method of the Section 3.2.1.

Remark 3.2.15 (Value approximation)
The approximation of the values on the boundaries are possible with the UDS or CDS method as shown in the Example 3.2.12.

Remark 3.2.16 (Right hand side)
The approximation of the right hand side is straight forward, because this term is explicitly known. We must only compute the integral over the the CV or approximate it with a numerical quadrature.

Remark 3.2.17 (Staggered grid)
In the finite volume method a staggered grid is common to reduce oscillations. This is realized to a similar way like the finite differences.

Time discretisation
Here we present no special method. Two possible methods are the *explicit* or *implicit Euler-method*, also possible there are high order methods.
A more detailed consideration is given by the section about **finite differences**, the method there is an explicit method in the velocity and implicit in the pressure. If we change the spatial discretisation with the finite differences by finite volumes, this time discretisation is also possible. Other methods are presented in the book of Freziger and Perić [Fre97].

An Algorithm to compute the Navier-Stokes equations is realizable in a similar way like the Algorithm 3.2.8. We must change the approximation of the momentum and the continuity equation from finite differences to finite volumes.
More details to this are given in the book of Freziger and Perić [Fre97].

Chapter 4

Automatic differentiation

In the following we use the conventional shortcut AD for automatic differentiation.
The idea of AD is to compute the exact derivative in a computer program, without differentiating the program code by hand. The principle of AD is to compute the derivative of an expression, that is created by elementary functions, by combining the derivative of the elementary functions with the chain rule.
The structure of this chapter is abutted on the lecture notes from Fischer [Fis05] and on my seminar [Lei06]. Now we define two notations, which we need to define the AD in a simple and mathematical correct form.

Definition 4.0.18 (Independent variable)
Let $\Omega \subset \mathbb{R}^n$, $f \in C^1(\Omega)$, $x := (x_1, \ldots, x_n) \in \Omega$. Then a variable x_i is called an **independent variable***, if $\frac{\partial f}{\partial x_i}$ is the desired derivative.*

Definition 4.0.19 (Elementary function)
A function f_i is called an **elementary function** *if*

$$f_i := \begin{cases} y \circ z & \circ & \in \{ +, -, *, /, **, \ldots \} \\ g(y) & g(\cdot) & \in \{sin(), cos(), exp(), \ldots\} \\ g(y,z) & g(\cdot,\cdot) & \in \{ atan2(), \ mod(), \ \ldots \} \end{cases}$$

with $i \in \mathbb{N}$, and $y, z \in \{x_1, \ldots, x_n, f_1, \ldots, f_{i-1}, a_1, \ldots, a_\ell\}$.
$x_1, \ldots, x_n$ are independent variables and $a_1, \ldots, a_\ell$ are constants. Then y, z are called **member of** *f_i.*

It is necessary to distinguish between two kinds of AD, the forward mode and the reverse mode. In the next section we give an introduction to the forward mode.

4.1 Forward mode

Now we define the forward mode of AD in a theoretical form and then we explain it with an example and derive the correct algorithm.

Definition 4.1.1 (Forward mode)
The **forward mode** *is a kind of AD, which computes mutually the evaluation and the derivative of each elementary function.*

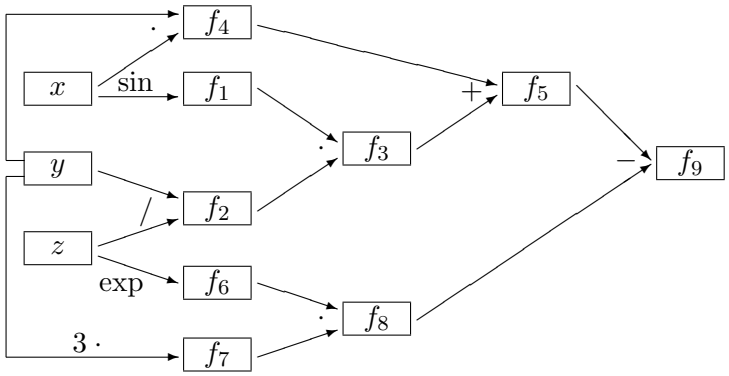


Figure 4.1: Function dependence graph

The Figure 4.1 shows a function dependence graph, of the function

$$f(x,y,z) := \sin x\frac{y}{z} + xy - e^z 3y, \tag{4.1}$$

to illustrate the dependence of the independent variables and the elementary functions of the function $f(x,y,z)$.

Example 4.1.2
We consider the function (4.1):

$$f(x,y,z) = \underbrace{\underbrace{\underbrace{\underbrace{sin(x)}_{=:f_1}\cdot\underbrace{(y/z)}_{=:f_2}}_{=:f_3} + \underbrace{x\cdot y}_{=:f_4}}_{=:f_5} - \underbrace{\underbrace{\exp(z)}_{=:f_6}\cdot\underbrace{(3\cdot y)}_{=:f_7}}_{=:f_8}}_{=:f_9}$$

and differentiate this one by using the forward mode.

$$\begin{aligned}
f_1 &= \sin(x), & \nabla f_1 &= \begin{pmatrix}\cos(x) & 0 & 0\end{pmatrix}^T\\
f_2 &= y/z, & \nabla f_2 &= \begin{pmatrix}0 & 1/z & -y/(z^2)\end{pmatrix}^T\\
f_3 &= f_1\cdot f_2, & \nabla f_3 &= \nabla f_1\cdot f_2 + \nabla f_2\cdot f_1\\
f_4 &= x\cdot y, & \nabla f_4 &= \begin{pmatrix}z & 0 & x\end{pmatrix}^T\\
f_5 &= f_3 + f_4, & \nabla f_5 &= \nabla f_3 + \nabla f_4\\
f_6 &= \exp(z), & \nabla f_6 &= \begin{pmatrix}0 & 0 & exp(z)\end{pmatrix}^T\\
f_7 &= 3\cdot y, & \nabla f_7 &= \begin{pmatrix}0 & 3 & 0\end{pmatrix}^T\\
f_8 &= f_6\cdot f_7, & \nabla f_8 &= \nabla f_6\cdot f_7 + \nabla f_7\cdot f_6\\
f_9 &= f_5 - f_8, & \nabla f_9 &= \nabla f_5 - \nabla f_8
\end{aligned}$$

The analytical solution is: $\nabla f(x,y,z) = \begin{pmatrix} \cos(x)\cdot(y/x)+y \\ \sin(x)/z + x - 3\cdot\exp(z) \\ (-\sin(x)\cdot y)/z^2 - \exp(z)\cdot(3\cdot y) \end{pmatrix}$

If we apply recursively the ∇f_i *for* $i = 1,\ldots,8$ *in* f_9 *it is trivial to see, that the solution of the forward mode is correct.*

Algorithm 4.1.3 (Forward mode)
Let $a_1,\ldots,a_\ell$ *be constants,* $x_1,\ldots,x_m$ *be the independent variables, and* $f_1,\ldots,f_n$ *beelementary functions for* $\ell,m,n \in \mathbb{N}$.
Now we write $g(y,z)$*, if we mean the expression of the elementary function* f_i *where* $i \in \{1,\ldots m\}$*. If the expression of the elementary function is a unitary expression, then* z *is empty, else* $y,z \in \{a_1,\ldots,a_\ell,x_1,\ldots,x_m,f_1,\ldots,f_{i-1}\}$*. Let* f_n *be the function, whose derivative is in demand. The calculation specification to compute the derivative of* f_n *is:*

for $i = 1,\ldots,n$

$f_i = g(y,z)$
$\nabla f_i = (\frac{\partial}{\partial y}g(y,z)\frac{\partial y}{\partial x_1} + \frac{\partial}{\partial z}g(y,z)\frac{\partial z}{\partial x_1},\ldots,\frac{\partial}{\partial y}g(y,z)\frac{\partial y}{\partial x_m} + \frac{\partial}{\partial z}g(y,z)\frac{\partial z}{\partial x_m})^T$

end

Remark 4.1.4 (Computational complexity)

1. *In the forward mode we can estimate the computation complexity for a elementary function*
$$f_k : \mathbb{R}^d \to \mathbb{R},$$
where $d = 1$*, if the elementary function is an unitary operation or function, else* $d = 2$*. The complexity is bounded with* $\max(6,\ 3n+4)$*, where* n *represents the dimension of the gradient vector, this estimation follows from the Table 4.1, which lists the computuation complexity of the elementary functions.*
For a given elementary function we get the exact computation complexity from the Table 4.1.

2. *For a function*
$$f : \mathbb{R}^{m\times n} \to \mathbb{R},$$
where n *represents the number of the independent variables,* m *is the number of the variables, whose derivative is not searched, and* $\ell \in \mathbb{N}$ *is the number of the elementary function to compute the function* f*, we can estimate the computation complexity in the following way* $\ell \cdot \max(6, 3n+4)$.

3. *It is clear, that this estimations are very coarse, but a priori it is not possible to give an better approximation, but, if the code is given, it is possible to analyze the code and give a determination of the exact computation complexity or a better approximation.*

Operator	Value f_k	derivative f'_k	Complexity
$+$	$f_i + f_j$	$f'_i + f'_j$	$n+1$
$+$	$f_i + a$	f_i	1
$-$	$f_i - f_j$	$f'_i - f'_j$	$n+1$
$-$	$f_i - a$	f'_i	1
$-$	$a - f_i$	$-f'_i$	$n+1$
$\cdot$	$f_i \cdot f_j$	$f'_i \cdot f_j + f_i \cdot f'_j$	$3n+1$
$\cdot$	$a \cdot f_i$	$a \cdot f'_i$	$n+1$
$/$	f_i/f_j	$(f'_i - f_k f'_j)/f_j$	$3n+1$
$/$	f_i/a	f'_i/a	$n+1$
$/$	a/f_j	$(-f_k \cdot f'_j)/f_j$	$n+3$
$\sqrt{}$	$\sqrt{f_i}$	$f'_i/(2f_k)$	$n+2$
$\hat{}$	${f_i}^{f_j}$	$f'_i(f_k f_j/f_i) + (\ln(f_i) f_k) f'_j$	$3n+4$
$\hat{}$	${f_i}^{a}$	$f'_i(f_k a/f_i)$	$n+3$
$\hat{}$	a^{f_j}	$(\ln(a) f_k) f'_j$	$n+3$

Function	Value f_k	derivative f'_k	Complexity
exp	$\exp f_i$	$f'_i f_k$	$n+1$
ln	$\ln f_i$	f'_i/f_i	$n+1$
sin	$\sin f_i$	$f'_i \cos f_i$	$n+2$
cos	$\cos f_i$	$f'_i(-\sin f_i)$	$n+3$
tan	$\tan f_i$	$f'_i/\cos^2 f_i$	$n+3$
arcsin	$\arcsin f_i$	$f'_i/\sqrt{1-f_i^2}$	$n+4$
arccos	$\arccos f_i$	$f'_i/(-\sqrt{1-f_i^2})$	$n+5$
arctan	$\arctan f_i$	$f'_i/(1+f_i^2)$	$n+3$
sinh	$\sinh f_i$	$f'_i \cosh f_i$	$n+2$
cosh	$\cosh f_i$	$f'_i \sinh f_i$	$n+2$
tanh	$\tanh f_i$	$f'_i/\cosh^2 f_i$	$n+2$

Table 4.1: Computation complexity of elementary functions in the forward mode

Remark 4.1.5 (Storage complexity)
In the forward mode, we can estimate the storage complexity for AD-types in the following way:

ℓ	*is the number of AD variables*
n	*is the dimension of the derivative*
c	*is a constant for some control parameter in a AD type*
S	*is the storage size of the values, it is dependent on the precision*
S_{all}	*is the maximum of the storage size for all AD-types, in the running program.*

$$S_{all} \leq \ell * ((n+1) * S + c)$$

Now we explain the reverse mode.

4.2 Reverse mode

This section is structured like the section before, firstly we define the reverse mode, then an example follows and at last, the algorithm and some remarks are given.

Definition 4.2.1 (Reverse mode)
The **reverse mode** *is a kind of AD, which computes the value of each elementary function, directly, and logs the elementary function with its value. The derivative will be computed at a given point of the program. The computation of the derivative starts with the last elementary function and finishes with the first one.*

Example 4.2.2
We consider the function (4.1).

$$f(x,y,z) := \underbrace{\underbrace{\underbrace{\underbrace{sin(x)}_{=:f_1}\cdot\underbrace{(y/z)}_{=:f_2}}_{=:f_3}+\underbrace{x\cdot y}_{=:f_4}}_{=:f_5}-\underbrace{\underbrace{\exp(z)}_{=:f_6}\cdot\underbrace{(3\cdot y)}_{=:f_7}}_{=:f_8}}_{=:f_9}$$

$$\begin{aligned} f_9 &= f_5 - f_8, & f_8 &= f_6 \cdot f_7, & f_7 &= 3 \cdot y, \\ f_6 &= \exp(z), & f_5 &= f_3 + f_4, & f_4 &= x \cdot y, \\ f_3 &= f_1 \cdot f_2, & f_2 &= y/z, & f_1 &= \sin(x), \end{aligned}$$

$$\begin{aligned}
\frac{\partial f_9}{\partial f_9} &= 1 \\
\frac{\partial f_9}{\partial f_8} &= -1 \\
\frac{\partial f_9}{\partial f_7} &= \frac{\partial f_9}{\partial f_8}\frac{\partial f_8}{\partial f_7} && = (-1)\cdot f_6 \\
\frac{\partial f_9}{\partial f_6} &= \frac{\partial f_9}{\partial f_8}\frac{\partial f_8}{\partial f_6} && = (-1)\cdot f_7 \\
\frac{\partial f_9}{\partial f_5} &= 1 \\
\frac{\partial f_9}{\partial f_4} &= \frac{\partial f_9}{\partial f_5}\frac{\partial f_5}{\partial f_4} && = 1\cdot 1 \\
\frac{\partial f_9}{\partial f_3} &= \frac{\partial f_9}{\partial f_5}\frac{\partial f_5}{\partial f_3} && = 1\cdot 1 \\
\frac{\partial f_9}{\partial f_2} &= \frac{\partial f_9}{\partial f_3}\frac{\partial f_3}{\partial f_2} && = 1\cdot f_1 \\
\frac{\partial f_9}{\partial f_1} &= \frac{\partial f_9}{\partial f_3}\frac{\partial f_3}{\partial f_1} && = 1\cdot f_2 \\
\frac{\partial f_9}{\partial z} &= \frac{\partial f_9}{\partial f_2}\frac{\partial f_2}{\partial z}+\frac{\partial f_9}{\partial f_6}\frac{\partial f_6}{\partial z} && = f_1\cdot(-y/z^2)+(-f_7)\cdot\exp(z) \\
\frac{\partial f_9}{\partial y} &= \frac{\partial f_9}{\partial f_2}\frac{\partial f_2}{\partial y}+\frac{\partial f_9}{\partial f_4}\frac{\partial f_4}{\partial y}+\frac{\partial f_9}{\partial f_7}\frac{\partial f_7}{\partial y} && = f_1\cdot(1/z)+1\cdot x+(-f_6)\cdot 3 \\
\frac{\partial f_9}{\partial x} &= \frac{\partial f_9}{\partial f_1}\frac{\partial f_1}{\partial x}+\frac{\partial f_9}{\partial f_4}\frac{\partial f_4}{\partial x} && = f_2\cdot\cos(x)+1\cdot y
\end{aligned}$$

The analytical solution is: $\nabla f(x,y,z) = \begin{pmatrix} \cos(x)\cdot(y/x)+y \\ \sin(x)/z + x - 3\cdot\exp(z) \\ (-\sin(x)\cdot y)/z^2 - \exp(z)\cdot(3\cdot y) \end{pmatrix}$
It is trivial to see, that the solution of the reverse mode is correct.

Now we look at a technical algorithm to compute the reverse mode.

Algorithm 4.2.3 (Reverse mode)
Let $x_1, \ldots, x_n$ be the independent variables, and $f_1, \ldots, f_m$ be elementary functions for $n, m \in \mathbb{N}$. And $\mathcal{M}_i$ is the set of all indices of elementary functions, whereof f_i is a member. η_i is the number of elements in $\mathcal{M}_i$. $\mathcal{M}_{x_i}$ is the set of all indices of elementary functions, whereof x_i is a member. η_{x_i} is the number of elements in $\mathcal{M}_{x_i}$.

$$\mathcal{M}_i := \{\ell \in \{m-1, \ldots, i+1\} \mid f_i \; member \; of \; f_\ell\}, \;\; \eta_i := \sharp\mathcal{M}_i$$

$$\mathcal{M}_{x_i} := \{\ell \in \{m, \ldots, 1\} \mid x_i \; member \; of \; f_\ell\}, \;\; \eta_{x_i} := \sharp\mathcal{M}_{x_i}$$

Let f_m be the function, whose derivative is in demand.
The calculation's specification to compute the derivative of f_m is :

$$\begin{aligned}
\frac{\partial f_m}{\partial f_m} &= 1 \\
\frac{\partial f_m}{\partial f_{m-1}} &= \frac{\partial f_m}{\partial f_{m-1}} \\
\frac{\partial f_m}{\partial f_{m-2}} &= \begin{cases} \frac{\partial f_m}{\partial f_{m-1}} \frac{\partial f_{m-1}}{\partial f_{m-2}} & if \; f_{m-2} \; a \; member \; of \; f_{m-1} \\ \frac{\partial f_m}{\partial f_{m-2}} & else \end{cases} \\
&\vdots \\
\frac{\partial f_m}{\partial f_i} &= \sum_{j=1}^{\eta_i} \frac{\partial f_m}{\partial f_{\ell_j}} \frac{\partial f_{\ell_j}}{\partial f_i} \quad for \; \ell_j \in \mathcal{M}_i \; and \; \ell_i \neq \ell_j \forall i \neq j \\
&\vdots \\
\frac{\partial f_m}{\partial f_1} &= \sum_{j=1}^{\eta_1} \frac{\partial f_m}{\partial f_{\ell_j}} \frac{\partial f_{\ell_j}}{\partial f_1} \quad for \; \ell_j \in \mathcal{M}_1 \; and \; \ell_i \neq \ell_j \forall i \neq j \\
\\
\frac{\partial f_m}{\partial x_n} &= \begin{cases} \frac{\partial f_m}{\partial x_n} & if \; \mathcal{M}_{x_n} = \emptyset \\ \sum_{j=1}^{\eta_{x_n}} \frac{\partial f_m}{\partial f_{\ell_j}} \frac{\partial f_{\ell_j}}{\partial x_n} \quad for \; \ell_j \in \mathcal{M}_{x_n} \; and \; \ell_i \neq \ell_j \forall i \neq j & else \end{cases} \\
&\vdots \\
\frac{\partial f_m}{\partial x_1} &= \begin{cases} \frac{\partial f_m}{\partial x_1} & if \; \mathcal{M}_{x_1} = \emptyset \\ \sum_{j=1}^{\eta_{x_1}} \frac{\partial f_m}{\partial f_{\ell_j}} \frac{\partial f_{\ell_j}}{\partial x_1} \quad for \; \ell_j \in \mathcal{M}_{x_1} \; and \; \ell_i \neq \ell_j \forall i \neq j & else \end{cases}
\end{aligned}$$

$(\frac{\partial f_m}{\partial x_1}, \ldots, \frac{\partial f_m}{\partial x_n})$ *is the searched solution.*

Remark 4.2.4 (Computational complexity)

1. *For an elementary function we can estimate the computation complexity with the Table 4.2, which lists the operations and their computation complexity. The computation complexity is bounded by* 9.

Operation	Value f_k	derivative $\bar{f}_i$	derivative $\bar{f}_j$	Complexity
$+$	$f_i + f_j$	$\bar{f}_i += \bar{f}_k$	$\bar{f}_j += \bar{f}_k$	3
$+$	$f_i + a$	$\bar{f}_i += \bar{f}_k$	-	2
$-$	$f_i - f_j$	$\bar{f}_i += \bar{f}_k$	$\bar{f}_j -= \bar{f}_k$	3
$-$	$f_i - a$	$\bar{f}_i += \bar{f}_k$	-	2
$-$	$a - f_i$	$\bar{f}_i -= \bar{f}_k$	-	2
$\cdot$	$f_i \cdot f_j$	$\bar{f}_i += \bar{f}_k \cdot f_j$	$\bar{f}_j += \bar{f}_k \cdot f_i$	5
$\cdot$	$a \cdot f_i$	$\bar{f}_i += \bar{f}_k \cdot a$	-	3
$\cdot$	$f_i \cdot a$	$\bar{f}_i += \bar{f}_k \cdot a$	-	3
$/$	f_i / f_j	$\bar{f}_i += \bar{f}_k / f_j$	$\bar{f}_j -= \bar{f}_k \cdot f_k / f_j$	7
$/$	f_i / a	$\bar{f}_i += \bar{f}_k / a$	-	3
$/$	a / f_i	$\bar{f}_i -= \bar{f}_k \cdot f_k / f_i$	-	4
$\sqrt{}$	$\sqrt{f_i}$	$\bar{f}_i += \bar{f}_k / (2 f_k)$	-	4
$\hat{}$	${f_i}^{f_j}$	$\bar{f}_i += \bar{f}_k \cdot f_k f_j / f_i$	$\bar{f}_i += \bar{f}_k \cdot \ln f_i \cdot f_k$	9
$\hat{}$	${f_i}^{a}$	$\bar{f}_i += \bar{f}_k \cdot f_k a / f_i$	-	5
$\hat{}$	a^{f_i}	$\bar{f}_i += \bar{f}_k \cdot \ln a \cdot f_k$	-	5

Function	Value f_k	derivative $\bar{f}_i$	Complexity
exp	$\exp f_i$	$\bar{f}_i += \bar{f}_k f_k$	3
ln	$\ln f_i$	$\bar{f}_i += \bar{f}_k / f_k$	3
sin	$\sin f_i$	$\bar{f}_i += \bar{f}_k \cos f_i$	4
cos	$\cos f_i$	$\bar{f}_i -= \bar{f}_k \sin f_i$	4
tan	$\tan f_i$	$\bar{f}_i += \bar{f}_k (1 + f_k^2)$	5
arcsin	$\arcsin f_i$	$\bar{f}_i += \bar{f}_k / \sqrt{1 - f_i^2}$	6
arccos	$\arccos f_i$	$\bar{f}_i -= \bar{f}_k / \sqrt{1 - f_i^2}$	6
arctan	$\arctan f_i$	$\bar{f}_i += \bar{f}_k / (1 - f_i^2)$	5
sinh	$\sinh f_i$	$\bar{f}_i += \bar{f}_k \cosh f_i$	4
cosh	$\cosh f_i$	$\bar{f}_i += \bar{f}_k \sinh f_i$	4
tanh	$\tanh f_i$	$\bar{f}_i += \bar{f}_k / \cosh^2 f_i$	5

Table 4.2: Computation complexity of elementary functions in the reverse mode

2. *For a function*

$$f : \mathbb{R}^{m \times n} \to \mathbb{R},$$

where n represents the number of the independent variables, m is the number of the variables whose derivative is not searched, and $\ell \in \mathbb{N}$ is the number of the elementary function to compute the function f. We can estimate the computation complexity in the following way $9 \cdot \ell$.

3. *A problem is: that, if we have two functions f, g, whose derivative is searched, and they have some equal temporary steps (elementary functions), we* **must** *use the evaluation for the derivative of each function. I.e., if ℓ_1 is the number of elementary functions of f and ℓ_2 is the one of g, then we must do $\ell_1 + \ell_2$ evaluations.*

Remark 4.2.5 (Storage complexity)
In the reverse mode, we can estimate the storage complexity for AD-types in the following way:

n	*is the number of elementary functions of each path*
n_{max}	*is the number of elementary functions*
p	*is the number of derivative paths*
c	*is a constant for some control parameter in a AD-type*
S	*is the storage size of the values, it is dependent on the precision*
S_{all}	*is the maximum of the storage size for all AD-types, in the running program.*

$$S_{all} \leq n_{max} \cdot S + \sum_{i=1}^{p} n_i(S + c)$$

4.3 Operator overloading

Operator overloading is a kind of implementation of the AD, which is a library of the computer language.

4.3.1 Functionality of operator overloading

The idea of the operator overloading is to create a new data type and define for this type the operators, like `+,-,*,/,...` and also the mathematical functions like `sin, cos,...`and now it is possible to work with this type similar like a `double`.
Now we show an example of the application of operator overloading on a simple program. In the next chapter we give a more detailed overview about an implementation of operator overloading AD tool in Fortran.

Example 4.3.1 (Operator overloading)
We consider the function (4.1).

Listing 4.1: AD operator overloading sourcecode f.

```
1  program example
2
3    use AD
4
5    type(forward) :: x,y,z
6    type(forward) :: f
7    real(kind(1.d0)) :: a, init_v, init_d(3)
8
9    a =  3.
10
11   init_d    =  0.
12   init_d(1) =  1.
13   init_v    =  1.1
14   call initAD(x,  init_v, init_d, .true.)
15   init_d    =  0.
16   init_d(2) =  1.
17   init_v    =  2.1
18   call initAD(y,  init_v, init_d, .true.)
19   init_d    =  0.
20   init_d(3) =  1.
21   init_v    = -0.1
22   call initAD(z, init_v, init_d, .true.)
23
24   f = sin(x) * (y/z) + x*y - exp(z) * a*y
25
26 end program example
```

Listing 4.2: Original source code.

```
1  program example
2
3    real(kind(1.d0)) :: x,y,z
4    real(kind(1.d0)) :: f
5    real(kind(1.d0)) :: a
6
7    a =  3.
8
9    x =  1.1
10   y =  2.1
11   z = -0.1
12
13   f = sin(x) * (y/z) + x*y - exp(z) * a*y
14
15 end program example
```

4.3.2 Advantages and disadvantages

Here we give an overview about the advantages and the disadvantages of the operator overloading method. At first we show the advantages.

Advantages

The main advantage of the operator overloading method is, that the source code has a clear form. This allows, that we can change and develop the program directly with AD, i.e. it is possible to develop a new program with AD in the same way as without AD, there is only one difference we must use for all variables, which have an influence on the end value, the AD-type.

Now we show the disadvantages.

Disadvantages

A big disadvantage of the operator overloading method is, that it is not available for each programming language, because the program language must support the operator overloading. This is the reason, why it is not possible to use this method on Fortran 77 and C, but it is possible in Fortran 90/95 and C++. Another problem is a performance's damage, because it is not possible, that the compiler optimizes the overloaded operators so good. It is also not trivial to use operator overloading on a complete and big program, because it is not simple to find the dependence on the necessary and unnecessary code.

4.4 Source transformation

Source transformation is a kind of implementation of AD, which is an independent program.

4.4.1 Functionality of source transformation

This program works like a preprocessor, it reads the source of the program, searches all relevant program lines and writes new lines in the program, which computes the derivative.
In the following we show some simple examples for the source transformation, which are created by the AD transformation tool **tapenade**[1].

[1] Tapenade is AD tool for source transformation developed by the INRIA.

Example 4.4.1 (Source transformation for the + operator)
Here we show the source transformation for the + operator, the Listing 4.3 is the original source code and the Listing 4.4 is the transformed source code.

Listing 4.3: Original sourcecode from a + subroutine.

```
subroutine add(f, x, y)
    real, intent(out) :: f
    real, intent(in)  :: x, y
    f = x+y
end subroutine add
```

Listing 4.4: AD transformed sourcecode from a + subroutine.

```
!        Generated by TAPENADE     (INRIA, Tropics team)
!  Tapenade - Version 2.2 (r1239) - Wed 28 Jun 2006 04:59:55 PM CEST
!  Differentiation of add in forward (tangent) mode:
!   variations  of output variables: f
!   with respect to input variables: x y
SUBROUTINE ADD_D(f, fd, x, xd, y, yd)
  IMPLICIT NONE
  REAL, INTENT(OUT) :: f
  REAL, INTENT(OUT) :: fd
  REAL, INTENT(IN) :: x
  REAL, INTENT(IN) :: xd
  REAL, INTENT(IN) :: y
  REAL, INTENT(IN) :: yd
  fd = xd + yd
  f = x + y
END SUBROUTINE ADD_D
```

Example 4.4.2 (Source transformation for the $**$ operator)
Here we show the source transformation for the $**$ operator, the Listing 4.5 is the original source code and the Listing 4.6 is the transformed source code.

Listing 4.5: Original sourcecode from a *sin* subroutine.

```
subroutine pow(f, x, y)
    real, intent(out) :: f
    real, intent(in)  :: x, y
    f = x**y
end subroutine pow
```

Listing 4.6: AD transformed sourcecode from a *sin* subroutine.

```
!        Generated by TAPENADE     (INRIA, Tropics team)
!  Tapenade - Version 2.2 (r1239) - Wed 28 Jun 2006 04:59:55 PM CEST
!  Differentiation of pow in forward (tangent) mode:
!   variations  of output variables: f
!   with respect to input variables: x y
SUBROUTINE POW_D(f, fd, x, xd, y, yd)
  IMPLICIT NONE
  REAL, INTENT(OUT) :: f
  REAL, INTENT(OUT) :: fd
  REAL, INTENT(IN) :: x
  REAL, INTENT(IN) :: xd
  REAL, INTENT(IN) :: y
  REAL, INTENT(IN) :: yd
  IF (yd .EQ. 0.0 .OR. x .LE. 0.0) THEN
    IF (xd .EQ. 0.0) THEN
      fd = 0.0
    ELSE
      fd = y*x**(y-1)*xd
    END IF
  ELSE IF (xd .EQ. 0.0) THEN
    fd = LOG(x)*x**y*yd
  ELSE
    fd = x**y*(LOG(x)*yd+y*xd/x)
  END IF
  f = x**y
END SUBROUTINE POW_D
```

Example 4.4.3 (Source transformation for the *sin*-function)
Here we show the source transformation for the sin-function, the Listing 4.7 is the original source code and the Listing 4.8 is the transformed source code.

Listing 4.7: Original sourcecode from a *sin* subroutine.

```
1  subroutine sin_(f, x)
2      real, intent(out) :: f
3      real, intent(in)  :: x
4      f = sin(x)
5  end subroutine sin_
```

Listing 4.8: AD transformed sourcecode from a *sin* subroutine.

```
1  !        Generated by TAPENADE     (INRIA, Tropics team)
2  !  Tapenade - Version 2.2 (r1239) - Wed 28 Jun 2006 04:59:55 PM CEST
3  !  Differentiation of sin_ in forward (tangent) mode:
4  !   variations  of output variables: f
5  !   with respect to input variables: x
6  SUBROUTINE SIN_D(f, fd, x, xd)
7    IMPLICIT NONE
8    REAL, INTENT(OUT) :: f
9    REAL, INTENT(OUT) :: fd
10   REAL, INTENT(IN) :: x
11   REAL, INTENT(IN) :: xd
12   INTRINSIC SIN
13   fd = xd*COS(x)
14   f = SIN(x)
15 END SUBROUTINE SIN_D
```

Example 4.4.4 (Source transformation for the *sqrt*-function)
Here we show the source transformation for the sqrt-function, the Listing 4.9 is the original source code and the Listing 4.10 is the transformed source code.

Listing 4.9: Original sourcecode from a *sqrt* subroutine.

```
1  subroutine sqrt_(f, x)
2      real, intent(out) :: f
3      real, intent(in)  :: x
4      f = sqrt(x)
5  end subroutine sqrt_
```

Listing 4.10: AD transformed sourcecode from a *sqrt* subroutine.

```
1  !        Generated by TAPENADE     (INRIA, Tropics team)
2  !  Tapenade - Version 2.2 (r1239) - Wed 28 Jun 2006 04:59:55 PM CEST
3  !  Differentiation of sqrt_ in forward (tangent) mode:
4  !   variations  of output variables: f
5  !   with respect to input variables: x
6  SUBROUTINE SQRT_D(f, fd, x, xd)
7    IMPLICIT NONE
8    REAL, INTENT(OUT) :: f
9    REAL, INTENT(OUT) :: fd
10   REAL, INTENT(IN) :: x
11   REAL, INTENT(IN) :: xd
12   INTRINSIC SQRT
13   IF (xd .EQ. 0.0 .OR. x .EQ. 0.0) THEN
14     fd = 0.0
15   ELSE
16     fd = xd/(2.0*SQRT(x))
17   END IF
18   f = SQRT(x)
19 END SUBROUTINE SQRT_D
```

4.4.2 Advantages and disadvantages

In this subsection we discuss the advantages and disadvantages of the source transformation.

Advantages

A big advantage of the source transformation is, that in finished code it is not so easy to find each dependence between the variables, if the code is big or not so good readable, and by a source transformation tool this is done automatically. This concept is also possible for each programming language.

Disadvantages

A disadvantage of the source transformation is, that the transformed source code is not good readable, because the tool inserts many lines and new variables. This is the reason why a transformed code is not so good for continuous developments, and new developments are not so easy realizable with source transformation. Another problem is, that there it is possible to get problems with variable names.

4.5 Complexity comparison of different approaches on two functions

In this section we analyze the time and the storage complexity of different approaches of AD on two functions, dependent of their dimensions. We also compare time and storage complexity of AD with the program without AD, and the numerical differentiation (finite differences).
The functions for this tests are chosen, such that we use most of the elementary functions, in order to get an idea to choose the right AD approach in more complex problems.
We have done the experiment on computers with the following configuration: AMD Athlon 64 3500+ processor with 2GB RAM and the operating system is the linux distribution ubuntu.
The programming language is Fortran 90/95, and we have used the Intel Fortran compiler *ifort* without any optimizations.
It is important to say, that the results are dependent on the AD tool, that is used for each AD approach, but the characteristically behavior is independent of it. In our case we have used a self written operator overloading tool[2], because we did not find a finished tool. For the source transformation we have used tapenade.
At first we consider a function, where the preimage is the $\mathbb{R}^n_+$, and the image is the $\mathbb{R}$.

4.5.1 Function f from the $\mathbb{R}^n_+$ to the $\mathbb{R}$

Here we consider a function, which maps from $\mathbb{R}^n_+$ to $\mathbb{R}$.

$$f(x) := \frac{\sin\left(\prod_{i=1}^{n}\sqrt{x_i+1}\right)}{\exp\left(-\sqrt{\sum_{i=1}^{n}\cos^2(x_i)}\right)} \tag{4.2}$$

[2]We explain this tool in the Chapter 5.

It is clear, that this function f is in C, the space of continuously functions. And it is also easy to see, that f is continuous differentiable for all

$$x \in \mathbb{R} \setminus \left\{ x_i = n\pi + \frac{\pi}{2} : \ n \in \mathbb{Z}, \ for\ all\ i = 1, \ldots, n \right\}.$$

We have chosen the values of the vector x at each program start with same random numbers, and we have checked, that the derivative exists for this values. And we have solved the Equation (4.2) 100,000 times.
We analyze the complexity for dimensions $n = 1, \ldots, 200$.
The basis of the program, to solve the Equation (4.2), is the same for all approaches. We only have to change the necessary sequences.

Time complexity

Now we interpret the results of the time complexity of the Equation (4.2), which are visualized in the Figure 4.2.
The first finding is, that operator overloading is in Fortran significant slower than source transformation, but the numerical differentiation is slowest. The next is, that the reverse mode of the operator overloading is worse than the forward mode of the operator overloading also in high dimensions.
An interesting result in the source transformation is that the forward mode is better than the reverse mode until about dimension 20. It is good to see, that the reverse mode is only a constant slower than the program code without computing a derivative, and this result holds also in high dimensions.

Storage complexity

The storage complexity depends on the dimension n of the Equation (4.2) as shown in the Figure 4.3. The storage complexity of the both approaches for the forward mode increases by a moderate factor. The storage complexity of the operator overloading reverse mode increases very strongly dependent on the dimension. The interesting point is, that the storage complexity of the source transformed reverse mode is nearly constant. The reason for this observation is, that this function is *simple*, so it is possible, that the transformation tool can transform the program in a way, which does not need the full tree of the function. That the storage complexity of the program without computing derivatives and the finite difference approach are nearly constant, is clear.

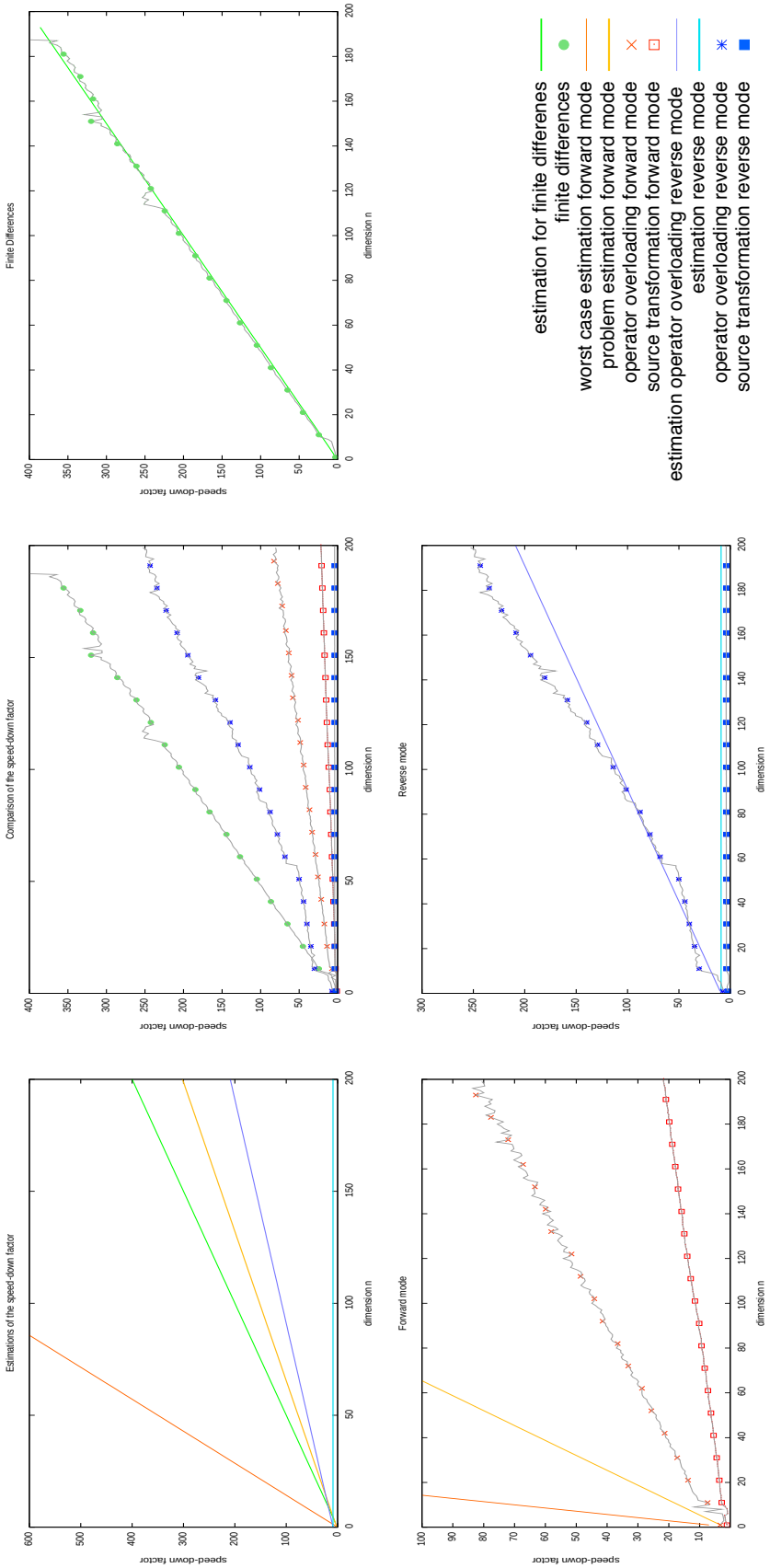


Figure 4.2: Time complexity of the Equation (4.2).

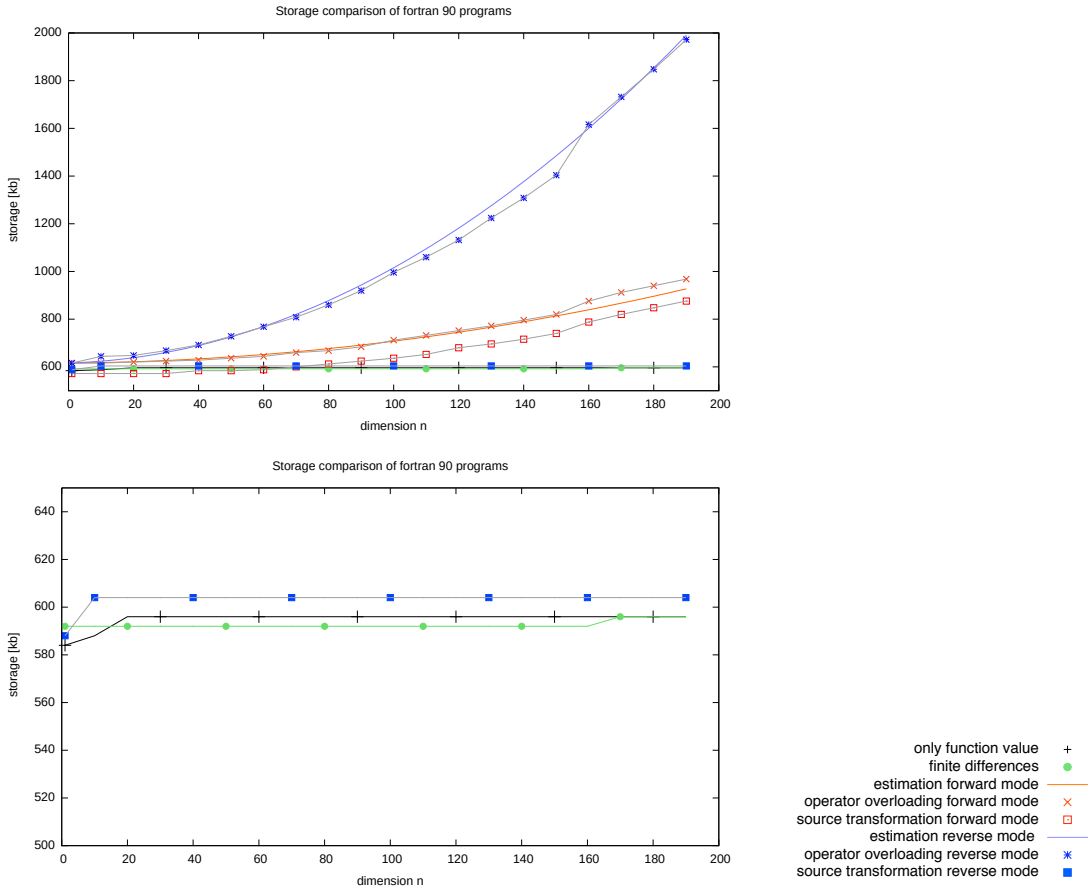


Figure 4.3: Storage complexity of the Equation (4.2).

4.5.2 Function f from the $\mathbb{R}$ to the $\mathbb{R}^n$

Here we consider a function, which maps from $\mathbb{R}$ to $\mathbb{R}^n$.

$$f(x) := \left(\frac{\sin(x \cdot i)}{\exp(x+i)} + (n \cdot x)^i \cdot \exp(x \cdot i) - i \cdot x + \cos\left(\frac{x \cdot x}{n}\right) \right)_{i=1}^{n} \tag{4.3}$$

It is clear, that this function f is in C^1, the space of continuously differentiable functions. The values of x we have choosen at each program start with the same random number. And we solve the Equation (4.3) 100,000 times.
We analyze the complexity for dimensions $n = 1, \ldots, 200$.
The basis of the program, to solve the Equation (4.3), is the same for all approaches. We only have to change the necessary sequences.

Time complexity

The Figure 4.4 shows the time complexity of the different approaches of AD to solve the Equation (4.3). A good result is, that the forward mode with source transformation is only a little bit slower than the program without cthe omputation of derivatives.

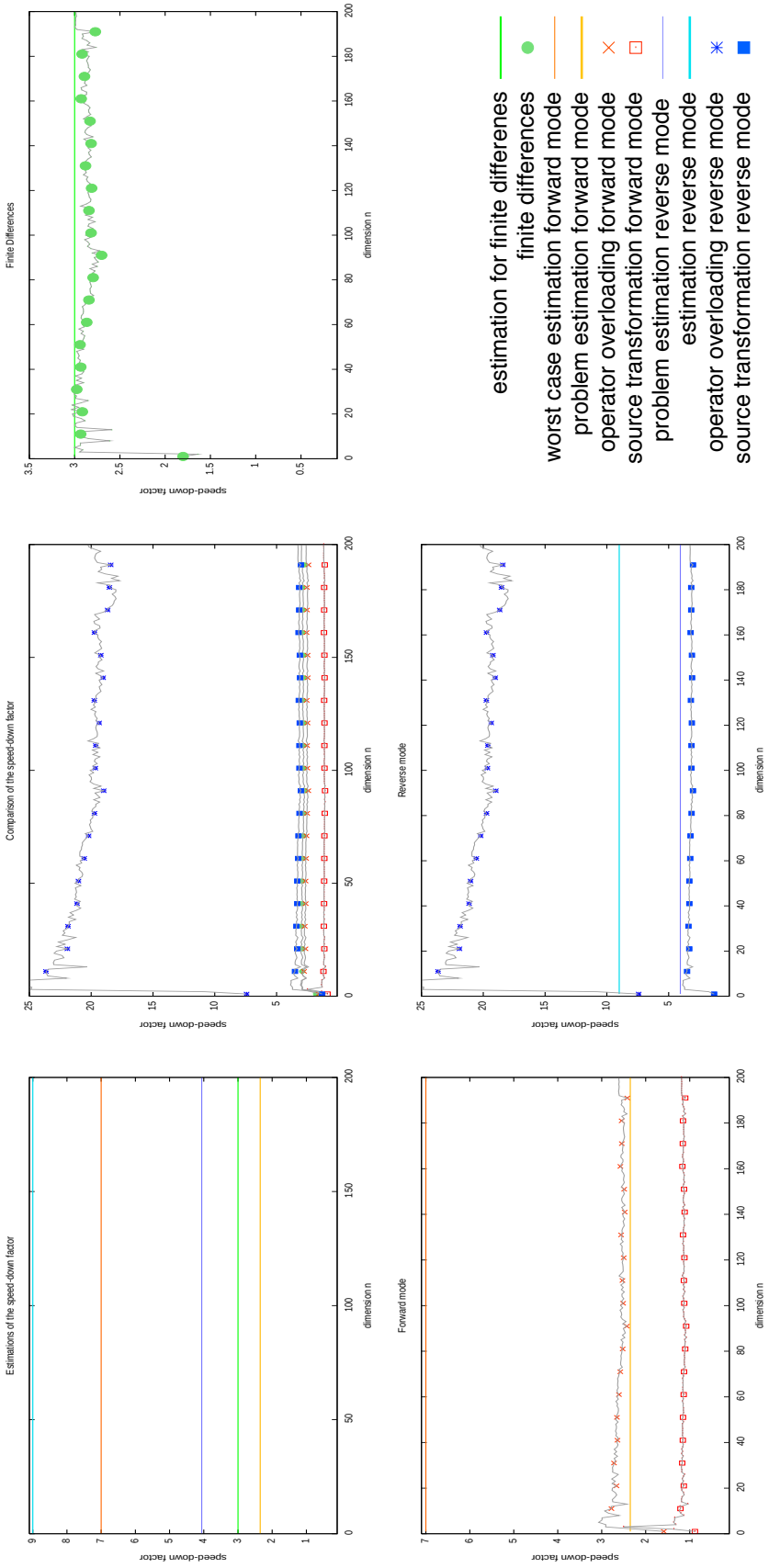


Figure 4.4: Time complexity of the Equation (4.3).

Storage complexity

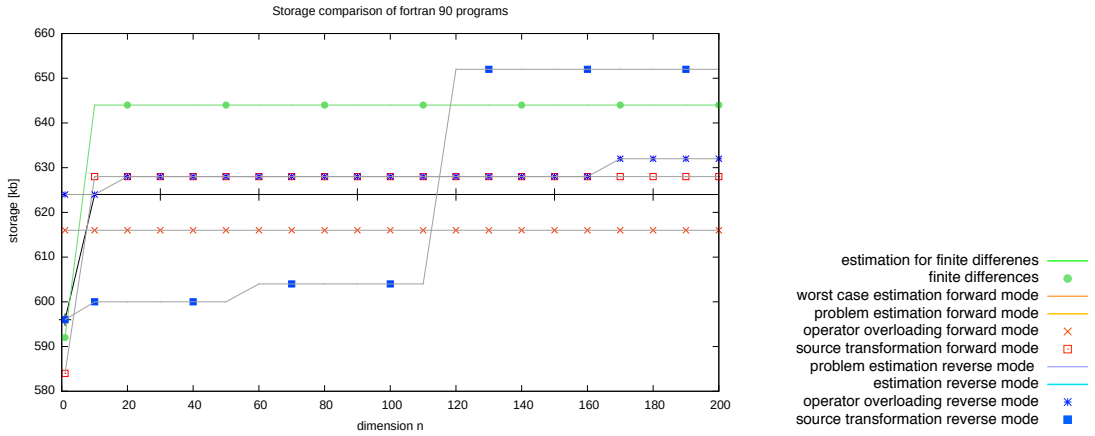


Figure 4.5: Storage complexity of the Equation (4.3).

We see in the Figure 4.5, that the storage complexity by solving the Equation (4.3) is not significant dependent on the chosen approach.
But this result depends on the fact, that the Equation (4.3) is a very simple function, and in bigger realistic problems this storage complexity is not so clear.

4.6 AD is not a silver bullet

In this section we show, that AD is not applieable on each problem, where the derivative is defined.

4.6.1 The problem

Now we explain and define the problem, on which we try to apply AD. We consider the Laplace-problem in two dimensions, on $\Omega = [0,1]^2$, with Dirichlet boundary conditions, for a given right side, $f(x,y) \in C^2$.

$$\begin{aligned} -\Delta u(x,y) &= f(x,y), \quad (x,y) \in \Omega \\ u(x,y) &= 0, \quad (x,y) \in \delta\Omega \end{aligned} \tag{4.4}$$

Now we discretisize this equation with finite differences, on the discrete domain

$$\Omega_h := \{x_i := ih : i = 0, \ldots, N+1\} \times \{y_i := ih : i = 0, \ldots, N+1\}, \tag{4.5}$$

so we get the following linear system of equations, to solve (4.4) with finite differences:

$$A_N x_N = f_N, \tag{4.6}$$

Then A_N from (4.6) has the following structure

$$A_N = \frac{1}{h^2} \begin{pmatrix} T_N & -I_N & 0 & & \cdots & & 0 \\ -I_N & T_N & -I_N & 0 & \cdots & & 0 \\ 0 & -I_N & T_N & -I_N & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \cdots & \vdots \\ 0 & & \cdots & & 0 & -I_N & T_N \end{pmatrix} \in \mathbb{R}^{N^2 \times N^2}, \tag{4.7}$$

where $h := \frac{1}{N+1}$ is the stepsize and N the degree of freedoms, in each dimension, $I_N \in \mathbb{R}^{N\times N}$ is the identity matrix and the matrix T_N from (4.7) has the following structure

$$T_N = \begin{pmatrix} 4 & -1 & 0 & & \cdots & & 0 \\ -1 & 4 & -1 & 0 & \cdots & & 0 \\ 0 & -1 & 4 & -1 & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \cdots & \vdots \\ 0 & & \cdots & & 0 & -1 & 4 \end{pmatrix} \in \mathbb{R}^{N\times N}, \tag{4.8}$$

The right hand side vector f_N is given by discretisation of

$$f(x,y) := 2y(1-y) + 2x(1-x), \tag{4.9}$$

in the following way

$$\begin{array}{rcl} f_N & := & \left(f_{(1,1)}, f_{(2,1)}, \ldots, f_{(N,1)}, f_{(1,2)}, \ldots, f_{(N,N)}\right)^T \\ & = & \left(f(x_1,y_1), \ldots, f(x_N,y_1), f(x_1,y_2), \ldots, f(x_N,y_N)\right)^T \end{array}. \tag{4.10}$$

The vector u_N is indicated on the same way, $u_{(i,j)}$ is the approximated solution on the point (x_i, y_j).
And this linear system of equations we solve with the Gauß-Seidel-Algorithm.

4.6.2 Application

By using the Gauß-Seidel-Algorithm to solve the equation, we get the following formula to compute $u_{(i,j)}$ in each iteration step.

$$u_{(i,j)} = \frac{1}{4}\left(h^2 f_{(i,j)} + u_{(i-1,j)} + u_{(i+1,j)} + u_{(i,j-1)} + u_{(i,j+1)}\right) \tag{4.11}$$

If for the index $i \pm 1 \in \{0, N+1\}$ or $j \pm 1 \in \{0, N+1\}$ holds, then we set the value of u zero, because this is the boundary condition.
By applying AD on (4.11), there results

$$\nabla u_{(i,j)} = \frac{1}{4}\left(h^2 \nabla f_{(i,j)} + \nabla u_{(i-1,j)} + \nabla u_{(i+1,j)} + \nabla u_{(i,j-1)} + \nabla u_{(i,j+1)}\right), \tag{4.12}$$

but this is the discrete solution of

$$-\Delta(\nabla u) = \nabla f, \tag{4.13}$$

with this follows, that we need boundary conditions for (4.13) and in the normal way we do not have this, if we have dirichlet boundary conditions by (4.4).
That is the reason, why we can not apply AD on this problem.

Chapter 5

An implementation of an operator overloading AD tool

In this chapter we present the implementation of an operator overloading AD tool, which is implemented in Fortran. The reason why we present a Fortran tool, is, that for C or C++ there exist many tools but not for Fortran. So it is more interesting to give an overview about a Fortran tool, developed by myself.
In this chapter we don't give an explanation of the principle functioning of the AD. The idea and the basics of this are given in the Chapter 4, for details look there. Here we explain the realization of AD in the programming language Fortran.
For details about programming language Fortran look in the Book [Aki03].

5.1 Forward mode

We start with the forward mode, because this implementation is more demonstrative and so it is later easier to understand the implementation of the reverse mode. It is clear, that we do not show the full source code, because this are many lines of code. Any parts of this are nearly the same, only for different operators and functions.

5.1.1 Main parts

In the Listing 5.1 we describe the main part of the forward mode operator overloading AD tool. Here we have only reduced some includes in the source code. But this parts are explained in the next subsections.
The forward mode operator overloading tool is realized in a `module`. This is similar to a `class` in C++.

Listing 5.1: AD forward mode module.

```
1  module AD
2      implicit none
3      integer, parameter           :: ADPrecision = kind(1.d0)
4      integer, parameter           :: Dimen       = 2
5      real(ADPrecision), parameter :: NAN = transfer(2140000000,1.0)
6      real(ADPrecision), parameter :: ADTol       = 1.E-20
7
8      type, public :: forward
9          private
10         real(ADPrecision)                    :: value
11         real(ADPrecision), dimension(Dimen) :: derivative
12         logical                              :: init
13     end type
14 !
15 ! includes files for operators and function interfaces
16 !
17 contains
18
19     function AD_forward_derivative(arg, num) result(derivative_)
20         type(forward), intent(inout) :: arg
21         integer, intent(in)          :: num
22         real(ADPrecision)            :: derivative_
23         derivative_ = arg%derivative(num)
24     end function AD_forward_derivative
25
26     function valueAD(arg) result(value)
27         type(forward), intent(in) :: arg
28         real(ADprecision)         :: value
29         value           = arg%value
30     end function valueAD
31
32     function getAD(arg) result(value)
33         type(forward), intent(in) :: arg
34         real(ADprecision)         :: value(Dimen+1)
35         value(1)        = arg%value
36         value(2:Dimen+1) = arg%derivative
37     end function getAD
38
39     subroutine initAD(arg, value, derivative_, activ)
40         type(forward),     intent(inout) :: arg
41         real(ADprecision), intent(in)    :: value
42         real(ADprecision), intent(in)    :: derivative_(Dimen)
43         logical,           intent(in)    :: activ
44         arg%value      = value
45         arg%derivative = derivative_
46         arg%init       = activ
47     end subroutine initAD
48 !
49 ! includes files for operators and function implementation
50 !
51 end module AD
```

In the lines 2 until 6 are the definitions of some global variables, which describe the precision, and some tolerances for the whole tool. In the 3rd line we define the dimension of the derivative fixed, because so we can work later without dynamic memory allocation, and it isn't necessary to check the dimension to runtime, this points are important for the performance.
In the lines 8 until 13 we define our new data type *type(forward)*. It contains the value and the derivative vector and a boolean for active. The activity variable enable, that we start the computation of the derivative on an arbitrary point in the program.
The next three lines 14 - 16, represent the list of include directives, for the interface declarations.
The command `contains` in line 17 start the implementation block of the module.
In the lines 19 until 24 the implementation of a function is called `AD_forward_derivative`, which returns the `num`-th entry of the derivative vector of an given AD-variable.
The next function `valueAD`, in the lines 26 until 30, returns the function value of a given AD-variable.

The function `getAD`, in the lines 32 until 37, returns the function value in the first entry of a vector with length `Dimen+1`. In the entries form `2` until `Dimen+1` the derivatives are contained.
In the lines 39 until 47, the subroutine `initAD` is implemented, which sets start values and start derivatives on a given AD-variable. Setting the start values for the derivative is important, if we have results from earlier calculations or, if the AD-variable is an independent variable, then we set the corresponding entry on the value 1.
The three lines 48 - 50, represent the list of `include` directives, for the interface implementation.

5.1.2 Redefinition of a mathematical function

Here we show the redefinition of a mathematical function exemplarly on the sinus function, for other mathematical functions it is analogical.

Listing 5.2: AD forward mode interface sinus.

```
1  !sin
2      interface sin
3          module procedure AD_forward_sin_f
4      end interface
```

The Listing 5.2 is the realization of the interface for the sinus function, with a AD-variable as an argument. In the third line it is given, which procedure implements the sinus function.

Listing 5.3: AD forward mode implementation sinus.

```
1  !sin
2      function AD_forward_sin_f(arg) result(this)
3          type(forward), intent(in) :: arg
4          type(forward)             :: this
5          this%value          = sin(arg%value)
6          this%init           = .false.
7          this%derivative     = 0
8          if(arg.init)then
9              this%init       = .true.
10             this%derivative = arg%derivative*cos(arg%value)
11         end if
12     end function AD_forward_sin_f
```

The Listing 5.3 is the realization of the implementation of the sinus function including the derivative.
In the line 5 the normal function value of the sinus is computed.
The lines 6 and 7 set the `init` of the result to false and their `derivative` to 0. If the `init` of the argument is true, we set the `init` of the result on true and compute the `derivative` of the result, by using the chain-rule, in the lines 9 and 10.
In a similar way, we must do this for the other mathematical functions like `cos`, `tan`.

5.1.3 Redefinition of an operator

Here we show the redefinition of an operator, exemplary on the $\star$ (multiplication) operator, for the other operators and the different kinds of left-hand-side and right-hand-side arguments it is analogical.
We consider two cases for the arguments, at first both arguments AD-variables, and second on the left-hand-side an AD-variable and on the right-hand-side a real-variable.

Listing 5.4: AD forward mode interface multiplication.

```
1 !mul
2     interface operator (*)
3         module procedure AD_forward_mul_ff
4     end interface
5     interface operator (*)
6         module procedure AD_forward_mul_rf
7     end interface
8 !...
```

The Listing 5.4 shows the implementation of the interface declarations for the pure AD-variable case in line 2 until 4 and then real-variable and AD-variable case in the line 5 until 7.

Listing 5.5: AD forward mode implementation multiplication.

```
1  !mul
2      function AD_forward_mul_ff(lhs, rhs) result(this)
3          type(forward), intent(in) :: lhs, rhs
4          type(forward)             :: this
5          this%value      = lhs%value * rhs%value
6          this%derivative = 0.
7          this%init       = .false.
8          if(lhs%init.or.rhs%init)then
9              this%init       = .true.
10             this%derivative = lhs%derivative*rhs%value + &
11                               rhs%derivative*lhs%value
12         end if
13     end function AD_forward_mul_ff
14
15     function AD_forward_mul_fr(lhs, rhs) result(this)
16         type(forward),     intent(in) :: lhs
17         real(ADPrecision), intent(in) :: rhs
18         type(forward)                 :: this
19         this%value      = lhs%value * rhs
20         this%derivative = 0.
21         this%init       = .false.
22         if(lhs%init)then
23             this%init       = .true.
24             this%derivative = lhs%derivative*rhs
25         end if
26     end function AD_forward_mul_fr
27 !...
```

In the Listing 5.5 we show the implementation of the both overloading variables, between the lines 2 until 13 for the pure AD-variable case, and in the lines 15 until 26 for the real-variable and AD-variable case.
The first block of both functions is very similar and analogical to the implementation of the overloading sinus which is given in the Listing 5.3. The main difference is the computation of the derivative, in first case it is the product-rule, and in the second case it is only a smultiplication of the derivative by a factor.

Remark 5.1.1
Now it is easy to see, that the implementation of an operator overloading tool for the **forward mode** *is not so complicated. The main work is desk work.*

5.2 Reverse mode

Now we start with the reverse mode, this is not so straight-forward like the forward mode. We start with main part of the reverse mode module.
As in the forward mode we do not show the full code, but only the structure and some exemplary parts.

5.2.1 Main parts

The reverse mode needs more internal structure for the evaluation of the function value and the derivative. So we have devided the module in more parts, in order to explain the program in a coherent way.
The main parts are diveded in six subparts.

Module

In this section we explain the Listing 5.6, which contains the structure and type declarations for the operator overloading reverse mode module.
The lines 2 until 14 are a cutting of the full list with operator-code's definitions. For each mathematical elementary function (in terms of the Chapter 4) we need an unique identification number for the computation of the derivative.
In the lines 15 until 18, we define some other constants for the module.
At next we define the new type `Element`, in the lines 20 until 31. An `Element` in the reverse mode is an entry in a list, and this list is called trace. To compute the derivative with the reverse mode the following informations are needed: the left and right hand side argument, if these do not exist this pointers are `NULL`, these variables are defined in the lines 23 and 24. We need also the value, the derivative, information about the dimension and the operator code. These variables are defined in the lines 27 until 30. The other variables are necessary for the structure of this tool. In particular, the `before` variable is necessary for evaluating the trace, and the variables `trace` and `tmp` are needed for a correct deleting of the trace.
In the lines 33 until 36 we define the new type `reverse`, this type is necessary, because of the reverse mode we need a list of all temporary results, so we have not to copy the variables we can only change the pointer destination.
Next, both variables, defined in line 38 and 39, are help variables for the list.
Then three lines follow, which represent the including of the interfaces. And the commands after the `contains` represent the main subroutines of this module, which we explain in the next sections.

Listing 5.6: AD reverse mode module.

```
1  module reverseAD
2      integer, parameter :: Assi      = 0
3      integer, parameter :: Ind       = 1
4      integer, parameter :: Const     = 2
5      integer, parameter :: Add       = 3
6      integer, parameter :: Sub       = 4
7      integer, parameter :: Mul       = 5
8      integer, parameter :: Div       = 6
9      integer, parameter :: Pow       = 7
10     integer, parameter :: ADSin     = 8
11     integer, parameter :: ADCos     = 9
12     integer, parameter :: ADTan     = 10
13     integer, parameter :: ADExp     = 11
14     !...
15     integer, parameter :: TraceNumber      = 1
16     integer, parameter :: TraceDimension   = 1
17     real, parameter    :: nan    = 1
18     real, parameter    :: ADTol = 0.1
19
20     type :: Element
21         private
22         type(Element), pointer :: before            => null()
23         type(Element), pointer :: lhs               => null()
24         type(Element), pointer :: rhs               => null()
25         logical, dimension(TraceDimension) :: trace = .false.
26         integer :: tmp                              =  0
27         real    :: value                            =  0
28         real    :: derivative                       =  0
29         integer :: derivativeDim                    = -1
30         integer :: operatorCode                     = -1
31     end type
32
33     type :: Reverse
34         private
35         type(Element), pointer :: storage => null()
36     end type
37
38     type(Element), pointer :: last => null()
39     type(Element), target  :: constDummy
40 !
41 ! includes files for operators and function interfaces
42 !
43 contains
44 !
45 !newIndependent
46 !
47 !newReverse
48 !
49 !reverseSweep
50 !
51 !deleteCheck
52 !
53 !deleteTrace
54 !
55 ! includes files for operators and function implementation
56 !
57 end module reverseAD
```

New independent

In the Listing 5.7 the subroutine is shown, which is necessary to create a new independent (in the sense of Chapter 4) variable.

In the first line we have the header of the subroutine and the delivery variables, this are the `reverse` type, function value, trace and the derivative dimension.
In the lines 8 until 11 we allocate the `Element` of the argument, and check if the allocation is successful.
Then we link the pointer `this` on this `Element`.
Then we set the private variables of this `Element` on the input parameters, the operator code

to `Ind` for an independent variable, and hang the `Element` in the trace list.

Listing 5.7: AD reverse mode new independent.

```
1  subroutine newIndependent(arg,value,trace,derivativeDim)
2      type(Reverse), target,intent(inout)          :: arg
3      real, intent(in)                             :: value
4      logical, dimension(TraceNumber), intent(in) :: trace
5      integer, intent(in)                          :: derivativeDim
6      integer                                      :: err
7      type(Element), pointer                       :: this
8      allocate(arg%storage,stat=err )
9      if(err/=0)then
10         stop
11     end if
12     this => arg%storage
13     this%value          =  value
14     this%trace          =  trace
15     this%tmp            =  1
16     this%derivativeDim  =  derivativeDim
17     this%operatorCode   =  Ind
18     this%before         => last
19     last                => this
20 end subroutine newIndependent
```

New reverse

The Listing 5.8 shows the creation of a new reverse element, which is not an independent variable.

Listing 5.8: AD reverse mode new reverse.

```
1  subroutine newReverse(arg)
2      type(Reverse), target,intent(inout) :: arg
3      integer :: err
4      type(Element), pointer              :: this
5      allocate(arg%storage,stat=err )
6      if(err/=0)then
7          stop
8      end if
9      this => arg%storage
10     this%before         => last
11     last                => this
12 end subroutine newReverse
```

At first the subroutine allocates the `Element`. And then this element is hanged in the list of the trace.

Reverse sweep

The *reverse sweep* is the most important function by the reverse mode, because this routine evaluates the derivative.
The source of this subroutine is very long. This is the reason, why we only show the structure. And we show the full implementation on some exemplary points. Elisions are marked with "`!...`", the calculus rules are given in the Table 4.2.

Listing 5.9: AD reverse mode reverse sweep.

```
1  function reverseSweep(arg, trace) result(traceVector)
2      type(Reverse), intent(in)       :: arg
3      integer, intent(in)             :: trace
4      type(Element), pointer          :: this, lhs, rhs
5      real, dimension(TraceDimension) :: traceVector
6      logical                         :: flag
7      this => arg%storage
8      flag = .false.
9      if(associated(this))then
10         this%derivative = 1
11       do while(associated(this))
12         if(this%trace(trace).eqv..true.)then
13             this%trace(trace) = flag
14             lhs => this%lhs
15             rhs => this%rhs
16             select case (this%operatorCode)
17                 case(Assi); lhs%derivative = this%derivative
18                 case(Ind)   !none
19                 case(Const) !none
20                 case(Add)   !...
21                 case(Sub)   !...
22                 case(Mul)
23                     lhs%derivative = lhs%derivative + this%derivative &
24                                                     * rhs%value
25                     rhs%derivative = rhs%derivative + this%derivative &
26                                                     * lhs%value
27                 case(Div)  !...
28                 case(Pow)  !...
29                 case(ADSin)
30                     lhs%derivative = lhs%derivative + &
31                                      this%derivative * cos(lhs%value)
32                 case(ADCos)     !...
33                 case(ADTan)     !...
34                 case(ADExp)     !...
35                 case(ADLog)     !...
36                 case(ADLog10)   !...
37                 case(ADAsin)    !...
38                 case(ADAcos)    !...
39                 case(ADAtan)    !...
40                 case(ADAtan2)   !...
41                 case(ADSinh)    !...
42                 case(ADCosh)    !...
43                 case(ADTanh)    !...
44                 case(ADAbs)     !...
45                 case(ADSqrt)    !...
46                 case(ADCeil)    !...
47                 case(ADFloor)   !...
48                 case(ADMod)     !...
49                 case(ADModulo)  !...
50                 case(ADAint)    !...
51                 case(ADNint)    !...
52             end select
53         else
54         end if
55         this => this%before
56       end do
57     end if
58     this => arg%storage
59     if(associated(this))then
60       do while(associated(this))
61         if((this%operatorCode==Ind))then
62             traceVector(this%derivativeDim) = this%derivative
63         end if
64         this%derivative = 0
65         this => this%before
66       end do
67     end if
68 end function reverseSweep
```

The input arguments of the `reverseSweep` are the arguments, whose derivative is in interest. And furthermore we get the trace of this derivative.

We pass the trace by beginning at the storage point of the argument in the trace, with a loop. In this loop we decide for each entry with the operator code and the trace variable, in which case we consider this entry. If the trace variable is not true we ignore this entry and goto the

next one. Otherwise we decide dependent on the operator code, how we compute the derivative for the left and the right-hand argument. For the $\star$ operator (multiplication) and for the sinus function it is shown, in the Listing 5.9, in the lines 22 until 26 and 29 until 31.
After the case command we link the actual trace entry to his precursor, and, if this one is NULL, the loop stops.
Then we pass the trace a second times, and look, if the operator code is `Ind`, an independent variable, we copy the value in the result vector on the entry of the derivation dimension of the independent variable.

Delete check

The function `deleteCheck` in the Listing 5.10 is necessary for the memory efficiency. But we must be careful because of the deception, which variable is deleteable and which we need in a later point to claim a clear structuring of the data.
This function only decides above candidates, which are given from the function `deleteTrace`, which is presented in the Listing 5.11. We explain this function in the next section.

Listing 5.10: AD reverse mode delete check.

```
1   function deleteCheck(trace, deleteable) result(this)
2       logical, dimension(TraceNumber), intent(in) :: trace
3       integer, intent(in)                         :: deleteable
4       logical                                     :: this
5       integer :: i
6       if(deleteable/=0)then
7           this = .false.
8           return
9       end if
10      do i=1,TraceNumber
11          if(trace(i).eqv..true.)then
12              this = .false.
13              return
14          end if
15      end do
16      this = .true.
17  end function deleteCheck
```

This function gets two arguments, the deleteable variable and the trace variable, a vector of logicals. The result is a logical variable which says delete or not.
At first, we check, wheather this variable is deleteable, that means, if it is an temporary variable, or not.
Then we check in a loop over the trace dimension, wheather the variable is needed by another trace for evaluating their derivative.

Delete trace

The problem of this subroutine is, we must check **each** entry in the trace list, wheather it is deleteable, this is done by the function `deleteCheck`, in the Listing 5.10.
We pass the trace into several steps: the first step is to delete each deleteable element until we find the first not deleteable element. This is then the new last element of the trace. On this point we skip this loop and restart with the new last element. The second loop deletes also each deleteable element.
The real deleting step follows in three steps, firstly to mark the element with a pointer, secondly to hang it out of the list, the last is to deallocate the element.

Now we have explained the structure of the reverse mode operator overloading tool, but this

structure is in the basic method similar to the evaluation part of a source transformation tool, also, if it is possible, that we get sometimes a direct formula, by using a smart algorithm. At next, we start to explain the overloading for an operator and a mathematical function.

Listing 5.11: AD reverse mode delete trace.

```
1  subroutine deleteTrace()
2      type(Element), pointer :: this, delete, old
3      logical, dimension(TraceNumber) :: trace = .false.
4      this => last
5      old  => last
6      do while(associated(this))
7          if(deleteCheck(this%trace,this%tmp) )then
8              delete      => this
9              old         => this%before
10             last        => this%before
11             this        => this%before
12             deallocate(delete)
13         else
14             this           => this%before
15             exit
16         end if
17     end do
18     do while(associated(this))
19         if(deleteCheck(this%trace,this%tmp))then
20             delete      => this
21             old%before => this%before
22             deallocate(delete)
23         else
24             old         => this
25         end if
26         this           => this%before
27     end do
28 end subroutine deleteTrace
```

5.2.2 Redefinition of a mathematical function

Here we explain the redefinition of the sinus function. We start with definition of the interface for the sinus function and then we come to the implementation.

Listing 5.12: AD reverse mode interface sinus.

```
1 !sin
2     interface sin
3         module procedure AD_reverse_sin_r
4     end interface
```

In the Listing 5.12 the definition of the interface for the sinus function is given. This interface definition is straight forward defined. We are more interested in the implementation, which we consider in the following.

Listing 5.13: AD reverse mode implementation sinus.

```
1  !sin
2      function AD_reverse_sin_r(arg) result(this)
3          type(Reverse), intent(in) :: arg
4          type(Reverse)             :: this
5          type(Element),pointer     :: tthis, targ
6          call newReverse(this)
7          tthis              => this%storage
8          targ               => arg%storage
9          tthis%value        =  sin(targ%value)
10         tthis%lhs          => targ
11         tthis%trace        =  targ%trace
12         tthis%operatorCode =  ADSin
13     end function AD_reverse_sin_r
```

The function, which is shown in the Listing 5.13, gets a reverse element as an argument and returns also a reverse element.
We start with the creation of the return object, in line 6.
Then we link the element in the trace of the argument and the return object. This is done in the lines 7 and 8.
In the following we evaluate the value, set the trace, the left hand side argument and the operator code.

5.2.3 Redefine of an operator

The last description of the reverse mode operator overloading tool is the redefinition of the $\star$ (multiplication) operator. Similarly to the other redefinitions, we start with the interfaces and then we consider the implementations.

Listing 5.14: AD reverse mode interface multiplication.

```
1  !mul
2      interface operator (*)
3          module procedure AD_reverse_mul_r_r
4      end interface
5      interface operator (*)
6          module procedure AD_reverse_mul_r_real
7      end interface
8  !...
```

The Listing 5.14 shows only the interface definition for the multiplication for the two cases left and right hand side are reverse elements and left hand side is areverse element and right hand side is a real variable.

Listing 5.15: AD reverse mode implementation multiplication.

```
1  !mul
2      function AD_reverse_mul_r_r(lhs, rhs)result(this)
3          type(Reverse), intent(in) :: lhs
4          type(Reverse), intent(in) :: rhs
5          type(Reverse)             :: this
6          type(Element),pointer     :: tthis, tlhs, trhs
7          call newReverse(this)
8          tthis               => this%storage
9          tlhs                => lhs%storage
10         trhs                => rhs%storage
11         tthis%value         =  tlhs%value * trhs%value
12         tthis%lhs           => tlhs
13         tthis%rhs           => trhs
14         tthis%trace         =  tlhs%trace .and. trhs%trace
15         tthis%operatorCode  =  Mul
16     end function AD_reverse_mul_r_r
17
18     function AD_reverse_mul_r_real(lhs, rhs) result(this)
19         type(Reverse), intent(in) :: lhs
20         real, intent(in)          :: rhs
21         type(Reverse)             :: this, rhs2
22         type(Element),pointer     :: tthis, tlhs, trhs
23         call newReverse(rhs2)
24         call newReverse(this)
25         trhs                => rhs2%storage
26         trhs%value          =  rhs
27         trhs%operatorCode   =  Const
28         tthis               => this%storage
29         tlhs                => lhs%storage
30         tthis%value         =  tlhs%value * rhs
31         tthis%lhs           => tlhs
32         tthis%rhs           => trhs
33         tthis%trace         =  tlhs%trace
34         tthis%operatorCode  =  Mul
35     end function AD_reverse_mul_r_real
```

The realization of the operator overloading is demonstrated with two functions, for the both kinds of the different arguments. But the proceeding is very similar.
In both cases we create a return object, see lines 7 and 24.
In the first case, we set the left and right hand arguments, the trace and the operator code, and evaluate the value of the multiplication.
In the second case we do this also, but we must create an object for the real variable to storage the value of this variable before, we set the operator code of this variable on `Const` for a constant variable.

Remark 5.2.1
By the overloading of the assumption operator we must be careful, because we must change sometimes a variable from a temporary variable to a not temporary variable, or in the other direction.

Remark 5.2.2
It is clear, that not each part of an operator overloading tool is detailed explained. But we have considered the most important point of the implementation.
It is also clear, that this both implementations of operator overloading tools are not the best possible implementations, in point of efficiency and simplicity. It is considered as a proof of concept, for this in the programming language Fortran 90/95 and as an experimental version.

Chapter 6

Application of AD on flow simulation

In this chapter we present the applications of AD in fluid dynamics.

6.1 2D Example Caffa

The program *Caffa* is a program from Freziger and Perić, which solves the Navier-Stokes equations with a finite volume method and is written in Fortran. The name Caffa stands for *Computer Aided Fluid Flow Analysis*.

6.1.1 Program structure

Now we explain the structure of the program *Caffa*, which we use in the following for the application of AD.
We show a very simplified algorithm of Caffa.

Algorithm 6.1.1 (Caffa)

```
read input
initialisation
do i=1,numberGrid                                  //grid loop
    extrapolate from the coarse to the fine grid
    do t=0,maxT                                    //time loop
        set boundary conditions
        move grid
        do j=1,maxIt                               //outer iterations loop
            compute momentum components
            compute pressure correction
            compute transport equation
        end
        save results
    end
end
```

6.1.2 Variable inflow

Here we present a simple problem, to test the justification of the application of AD in the fluid dynamics. For this test we use the program Caffa in a modified version. That means, that we have excluded all parts of the program, which we have not required. Especially in this case we exclude the part of the moving grid.
Now we explain the configuration of the problem, in the Figure 6.1 the geometry is shown.
Here we consider a Naca0015 profile in a channel with non-slip boundary conditions on the

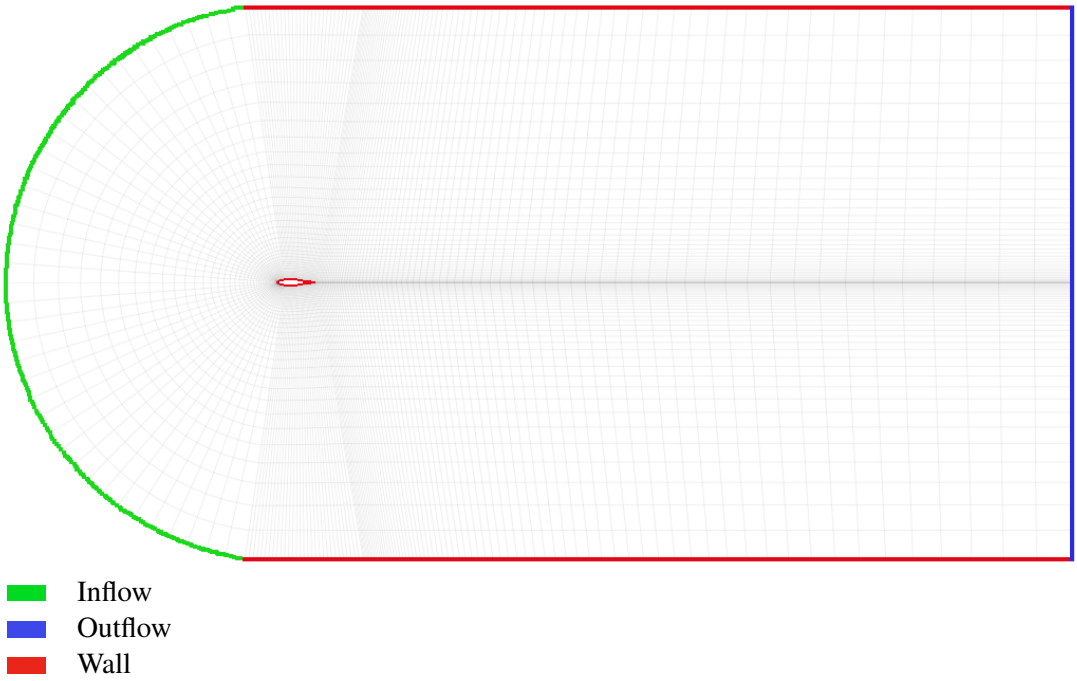


Figure 6.1: Geometry of the example

boundary of the channel and on the profile. On the left side we have inflow boundary conditions and the right side is sedated with outflow conditions.
The Figure 6.2 visualizes the inflow profile dependent on the inflow parameter.
The inflow profile is given by the following formula:

$$u(x, y, c) := -c * \left(\frac{0.68 - |y|}{0.68} \right)^2 \cdot x,$$

where x, y are the coordinates of the grid, $x \in [-7, -1]$ and $y \in [-0.68, 0.68]$, the variable c is the inflow parameter. The inflow parameter c is the parameter, respect to which we differentiate the output parameter. The output parameters are the force in the x direction and the force in the y direction on the airfoil.
The Figure 6.3 shows the developing of the forces in x direction acting on the plane over the full simulation time. After about 440 seconds we can monitor a huge oscillating in the derivatives, the reason for this is, we get a noise in the force values on the basis of summation of rounding errors over the time, and this strengthen each other in the derivative.
The Figure 6.4 represents the extract of the Figure 6.3, in which the noise is not significant.
The Figure 6.5 evinces the force and their derivation dependent on the inflow parameter c, in

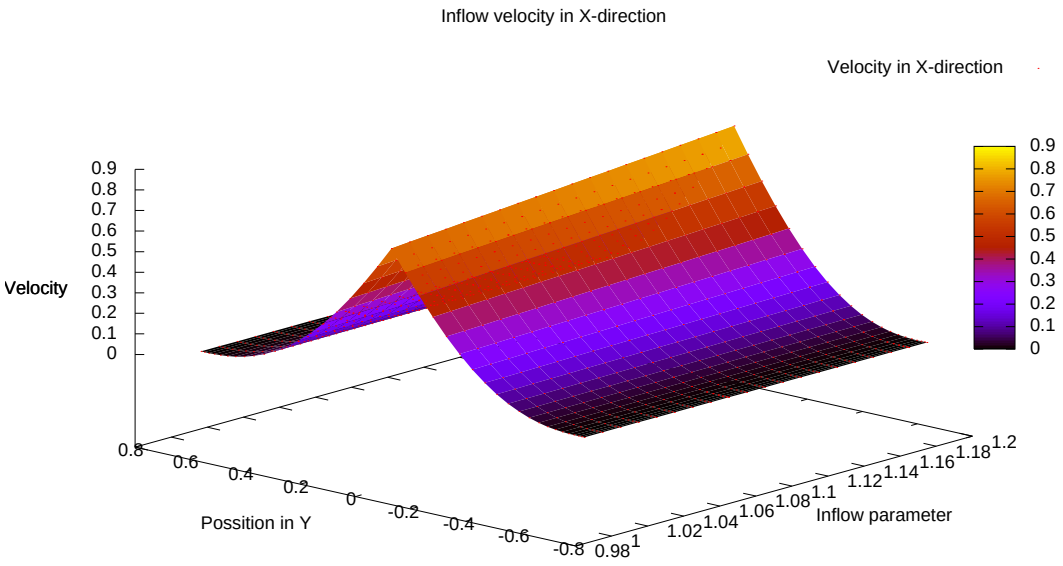


Figure 6.2: Inflow profile

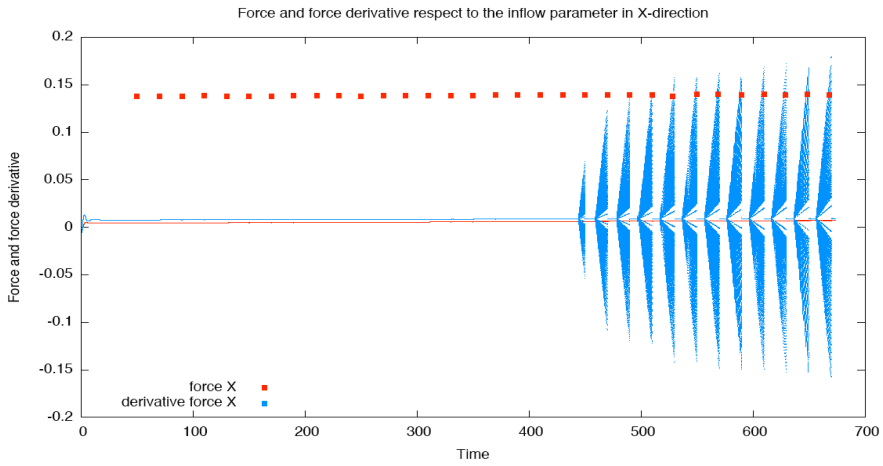


Figure 6.3: Force and derivative on the foil in x direction over the time

the same time interval, which is demonstrated in Figure 6.4.

The Figure 6.6 shows the validation of the derivatives, which are computed with AD, with finite differences.

For the forces and their derivatives in y direction we have the same results, like the forces and their derivatives in x direction.

The conclusion of this example is, it is possible to apply AD to the simulation of fluid dynamics. The Figure 6.6 shows, that the values of the derivative, computed by AD, are in the same order of magnitude like the computed finite differences. A disadvantage is, that the computation of the derivative is very sensitive respecting to numerical noise in the values.

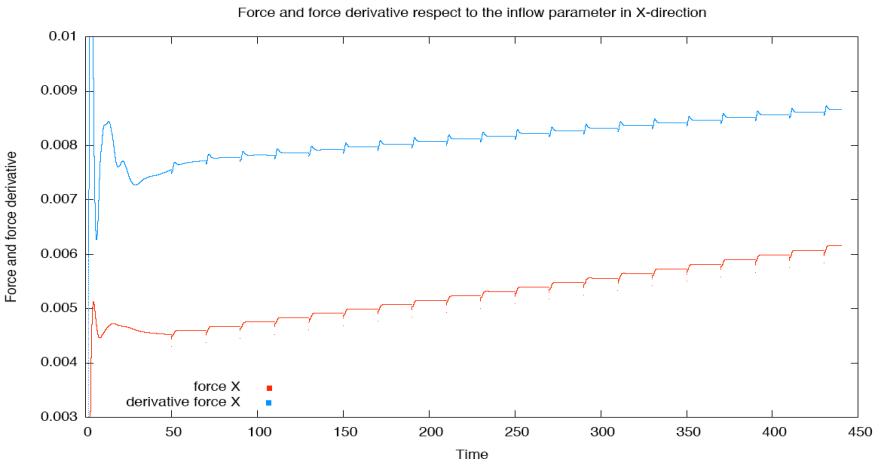


Figure 6.4: Force and derivative on the foil in x direction over the time

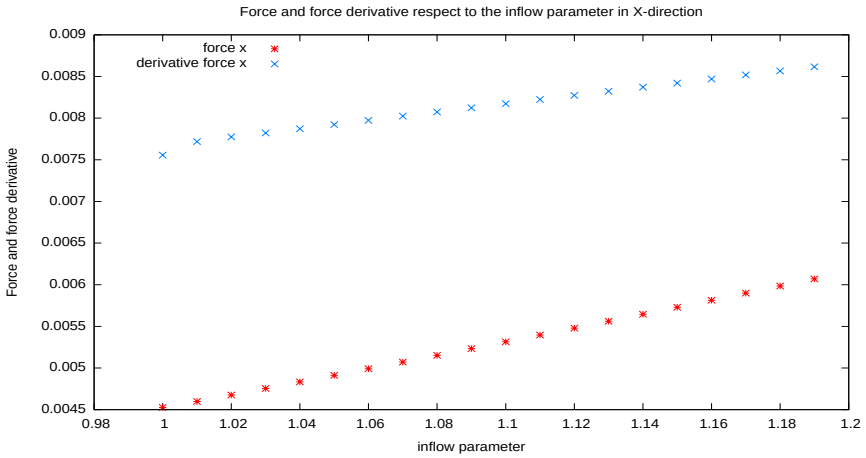


Figure 6.5: Force and derivative on the foil in x direction respect to the inflow parameter

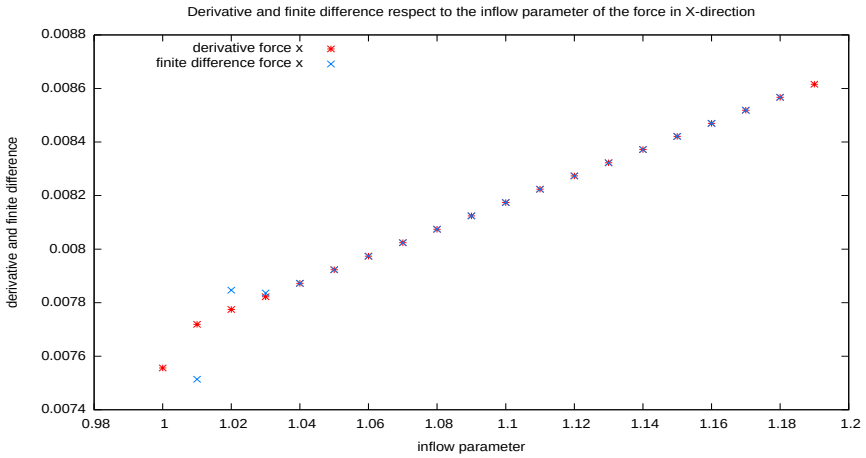


Figure 6.6: Derivative and FD on the foil in x direction respect to the inflow parameter

6.1.3 Angle of attack

In this subsection, we consider the angle of attack of an Eppler 420 profile. The geometry and the boundary conditions are same as in the section before but the profil has changed.
The Figure 6.7 illustrate the angle of attack. We have a look for two topics here. At first we

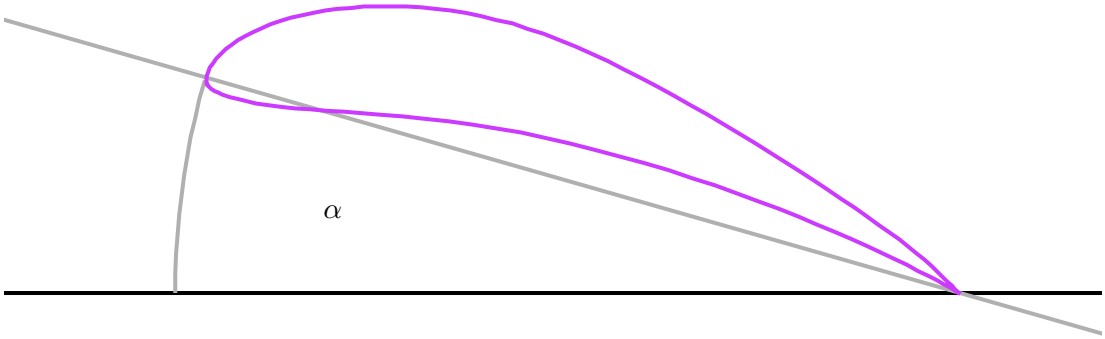


Figure 6.7: Angle of attack

compute the value, first and second derivatives of the forces in y direction, then we verify the derivatives by comparison with the finite differences. We evaluate the force and their first and second derivative for a angle of attacks between 0 and -0.69 radian, in -0.01 steps, this is equivalent to an angle of 0 until about -39.5 degree in -0.57 degree steps.
At next we maximize the force in y direction with the Newton-Algorithm. From the engineer science in fluid dynamics it is known, that, for a given velocity and a given profile, the forces in y direction respect to the angle of attack have only one maximum.
So we search the zero-point of the first derivative with the Newton-Algorithm.

Verification of first and second derivatives

Here we represent the values of the forces in y direction and their first and second derivatives, in dependence of the angle of attack.
The Figure 6.8 shows us the forces on the airfoil in y direction with respect to the angle of attack α, we see, that the developing of the values are relative smooth.
The first picture of the Figure 6.9 shows us the comparison of the first derivative of the forces in y direction with respect to the angle of attack. The conclusion of this picture is, that the range of the derivatives, computed with AD, and of the finite differences is the same, but we see also, that the finite differences have some outlier and the AD has an smoothing effect.
The second picture presents the comparison of the 2nd finite differences, finite differences of the first derivative, computed with AD, and the 2nd derivative, computed with AD. The informations are clear: we see as in the first picture the smoothing effect of AD. And we see, that the values tend to result in the same range, but the 2nd finite differences have begun to oscillate and the finite differences of the first derivative have some outlier.
This graphics show demonstratively, why it is for some application necessary to use derivatives, computed with AD, instead of finite differences.
Now we explain the structure of the program for the optimization.

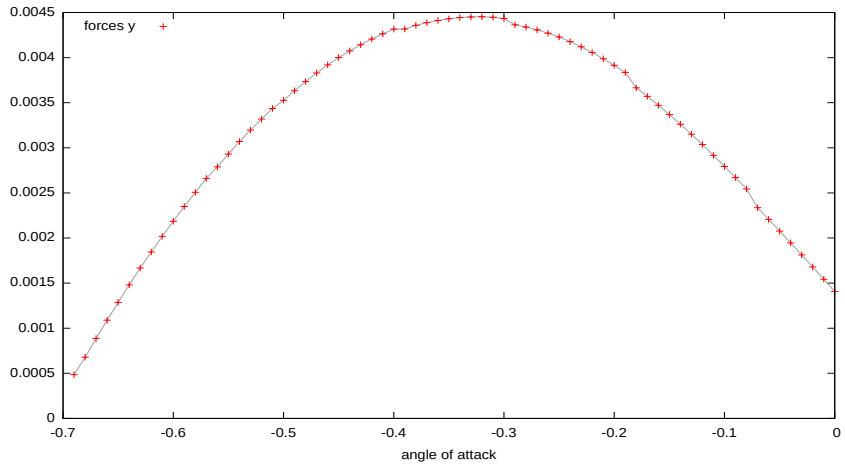


Figure 6.8: Curve of the forces in y direction on the airfoil.

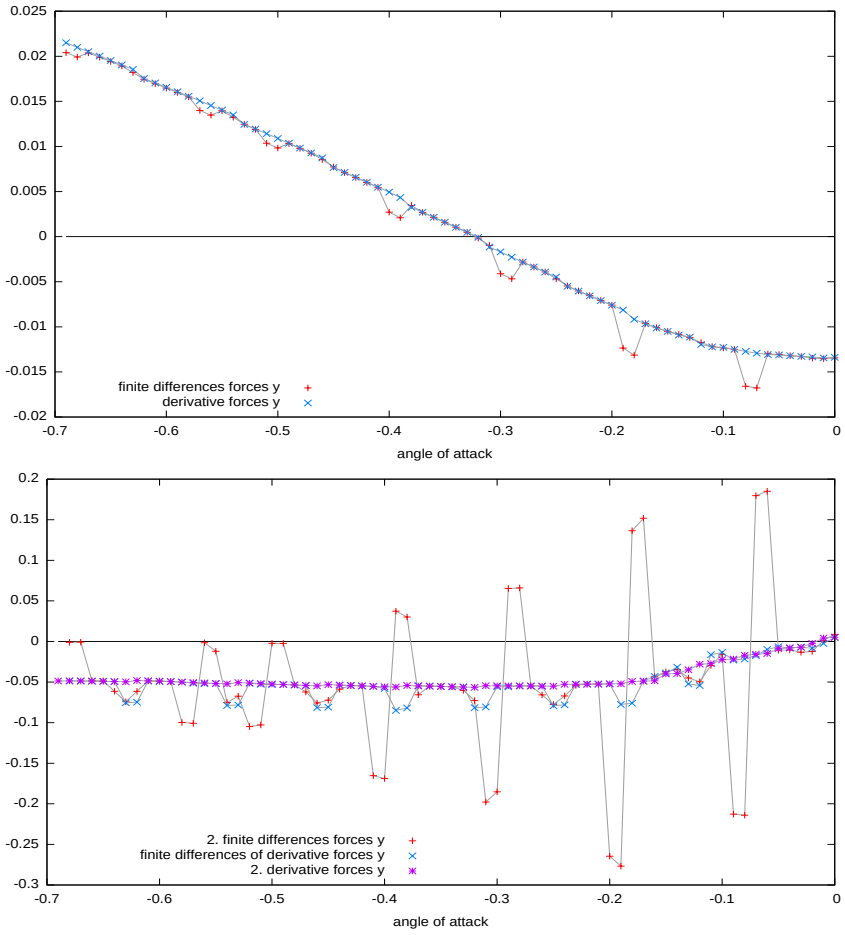


Figure 6.9: Comparison of finite differences and derivatives.

Maximization of the forces in y direction

Now we give the algorithm for this especial optimization (maximization) problem.

Algorithm 6.1.2 (Angle of attack maximization)
1. choose start angle of attack α
2. choose abort parameter tol
3. do until ($f'_y < tol$*)*
3.1 create the grid
3.2 compute forces in y *direction (caffa 2 derivative version)*
3.3 compute new angle of attack
end

Here we give some remarks to the program.

Remark 6.1.3

(i) By the Newton-Algorithm, we have constrained the maximal change of the angle of attack in each step, on 0.1 radian. The reason for this limitation is, to avoid a grid, which is too much deformed, so that the program diverges.

(ii) The start point for the Newton algorithm is chosen -0.03 radian. This we have done, because the second derivative is for 0 very small but positive and the first derivative is negative. A reason for this is, it is possible, that the second derivative is not hundred percently converges. And the Newton-Algorithm converged only locally.

In the Table 6.1 we have presented the values of the angle of attack, the forces in y direction and their first and second derivative. To show the convergent of the Newton-algorithm.
The Figure 6.10 shows the force points on the developing of the force respect to the iterations. The conclusion of this numerical experiment is, that the application of AD for numerical optimization in flow simulation is possible and works well.

Iteration	angle of attack α	Force f_y	1. derivative f'_y	2. derivative f''_y
0.	-0.03000000	0.001811820383	-0.0132745781000000000	-0.007028305677
1.	-0.13000000	0.003151422923	-0.0111443324400000000	-0.03487816619
2.	-0.23000000	0.004119586926	-0.0060333775440000000	-0.05273238525
3.	-0.33000000	0.004451043681	0.0004825580166000000	-0.05614455376
4.	-0.32140508	0.004453181174	-0.0000011305598590000	-0.05640617265
5.	-0.32142512	0.004453217306	0.0000000844061688300	-0.05640560206
6.	-0.32142362	0.004453240372	0.0000000755028276400	-0.05640569324
7.	-0.32142229	0.004453258784	0.0000000471279242200	-0.05640570905
8.	-0.32142145	0.004453271488	0.0000000375825516900	-0.05640571972
9.	-0.32142078	0.004453278911	0.0000000361535010000	-0.05640576372
10.	-0.32142014	0.004453288589	0.0000000065792592790	-0.05640570186
11.	-0.32142003	0.004453291663	0.0000000237156276200	-0.05640573354
12.	-0.32141961	0.004453297321	0.0000000014507328120	-0.05640568955
13.	-0.32141958	0.004453299546	0.0000000101950427000	-0.05640569150
14.	-0.32141940	0.004453303001	-0.0000000010719954670	-0.05640565649
15.	-0.32141942	0.004453304748	0.0000000029741928190	-0.05640564385
16.	-0.32141937	0.004453304989	0.0000000070559291260	-0.05640565943
17.	-0.32141924	0.004453303753	0.0000000118916028200	-0.05640570519
18.	-0.32141903	0.004453305336	-0.0000000029493869330	-0.05640568673
19.	-0.32141908	0.004453306561	-0.0000000023858489500	-0.05640566599
20.	-0.32141912	0.004453307574	-0.0000000023514392310	-0.05640564774
21.	-0.32141917	0.004453304047	0.0000000183921768400	-0.05640572960
22.	-0.32141884	0.004453307763	-0.0000000160892771500	-0.05640565956

Table 6.1: History of the force and their derivatives in y direction respect to the iterations

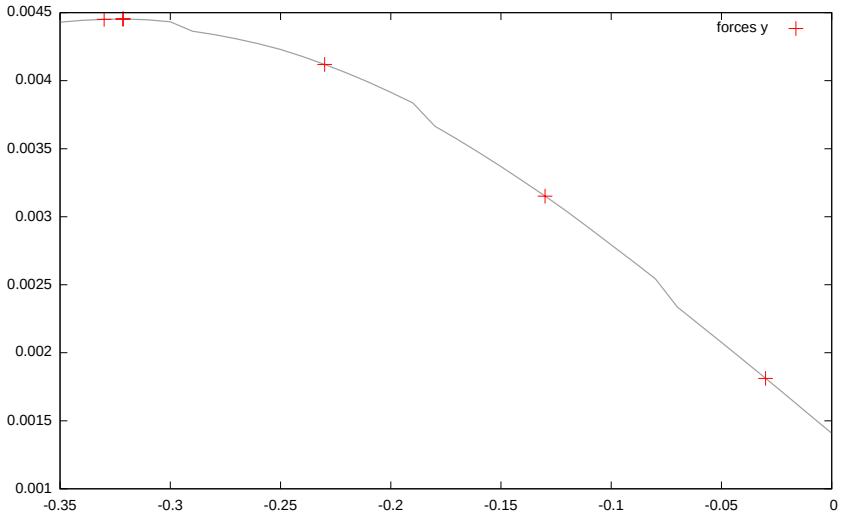


Figure 6.10: History of the force in y direction respect to the iterations

6.2 3D Example Comet

Here we consider the simulation of the Navier-Stokes equations in 3D with the commercial software *Comet*.
The program *Comet* is a program from CD-Adapco, which solves the Navier-Stokes equations, by a finite volume method and is written in Fortran, C and C++.
At first, we explain the structure of the program. Here we do not give any details but the basic structure, then we show the application of AD on Comet.

6.2.1 Program structure

It is clear, that we can not give all details about the program, because it is a commercial code. But we explain the structure to use the program as an user, and show the point for the preparation of AD on the code. And then we explain the validation of the results.

- Description of a problem to solve it with Comet.
  In principle there exists more than one method to describe a problem to simulate it with Comet, but here we consider only the simplest method, with the shell program `Cometpp`. We can decompose this point in three sub-points:
  - Geometry description:
    The description of the simulation geometry is possible in many ways: with mesh creators, CAD programs and command line program `Cometpp`.
  - Problem description:
    Here we define different regions to set the boundary conditions, material properties, maximal number of iterations, convergence criterial and number and size of the time steps, if we have a non static problem. All this things we do with commands in `Cometpp`, it is also possible to do this with a graphic user interface.
  - Usercoding:
    It is possible to expand the program with usercoding. This usercoding uses defined interfaces of the program
- Link the executable program:
  After finishing the preprocess we must link the executable program for this problem. The linking and compilation of the usercoding is done with the script `lcomet`. The script `lcomet` needs the following informations: name of the executable program, precision (i.e. single or double precision), does usercoding exist or not, and information about serial or parallel computing is desired.
- Execute the program:
  Start the compiled program and input the name of the problem.
- Visualization of the results:
  The results are visualizable with the program `Cometpp` or by explicitly saving in files with another program for example `gnuplot`.

Now we explain the ansatz of the AD on comet. The first step is to prepare the usercodes and the interfaces. This means, that we have to define the quantity, which should be differentiated and we have to prepare the interfaces in such a way, that the independent variable is on all important points in the program available. We explain this by the following example:

Example 6.2.1
Now we prefer to get the derivatives of the forces on an airfoil with respect to the inflow velocity.
The computation of the forces over the airfoil is possible in the usercoding file post.f, with a loop over the boundary elements of the airfoil. The initialization of the inflow velocity is possible in two ways:

- *In the problem definition with the script language of* `Cometpp`*, but here the problem for each configuration is, we must use the preprocess and we do not get the information about the inflow changing in the program, i.e. we can not differentiate the program in a easy way.*
- *Set the inflow velocity in the usercoding, this is possible in the file userbc.f. Now we must ensure, that the factor which has influence on the inflow velocity is available from the main program until in this subroutine because the availability is necessary for the AD-tool.*

At next we can use the AD-tool, e.g. *tapenade* on the source code. If we want, that this tool does not differentiate or consider all subroutines, we must out-command the calls for this subroutines. After this we must cancel the out-command, and align the output.

6.2.2 Moving grid

Here we explain an example with a moving grid, i.e. we consider the forces on the airfoil with respect to the angle of attack. In the Figure 6.12 the domain is shown. The domain has two

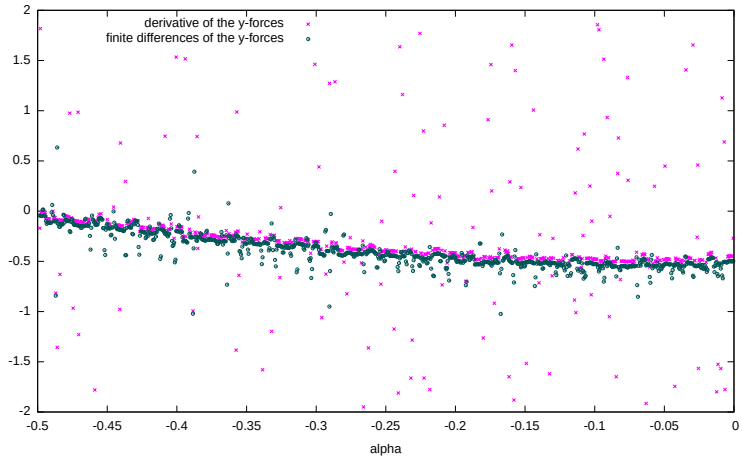
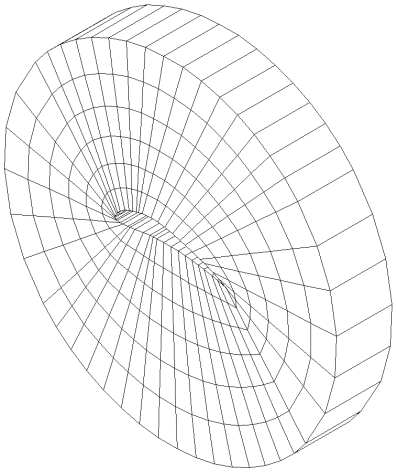
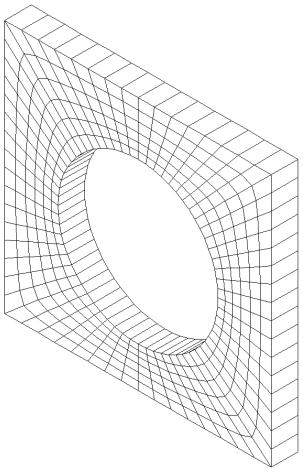


Figure 6.11: Derivatives and finite differenes of the Example 6.2.2

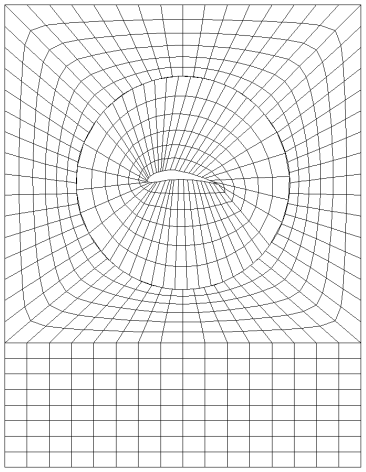
parts the fixed part is shown in the picture b.) and the moved part shown in the picture a.).
The picture c.) shows the topview of the domain in the pictures d.) and e.) we see a 3D view of the domain.
The picture 6.12 f.) shows the boundaries of the domain: we have inflow boundary conditions on the left side, outflow boundary conditions on the right side, and we have slip boundary conditions on the top and the bottom and we have non slip boundary conditions in the middle on the airfoil. One problem is, that it is non trivial to get the derivatives of the grid depending



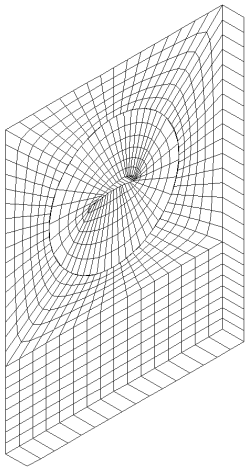
a.) Moving part



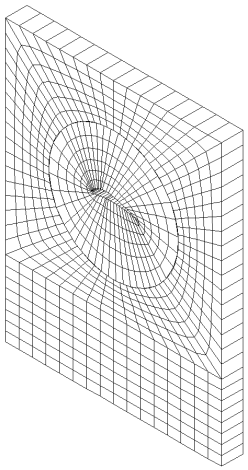
b.) Fixed part



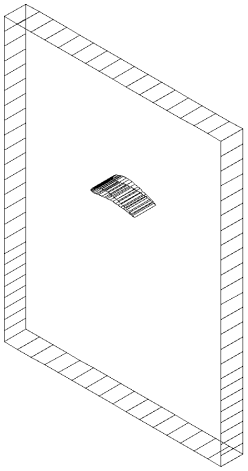
c.) topview of the domain



d.) 3D view of the domain



e.) 3D view of the domain



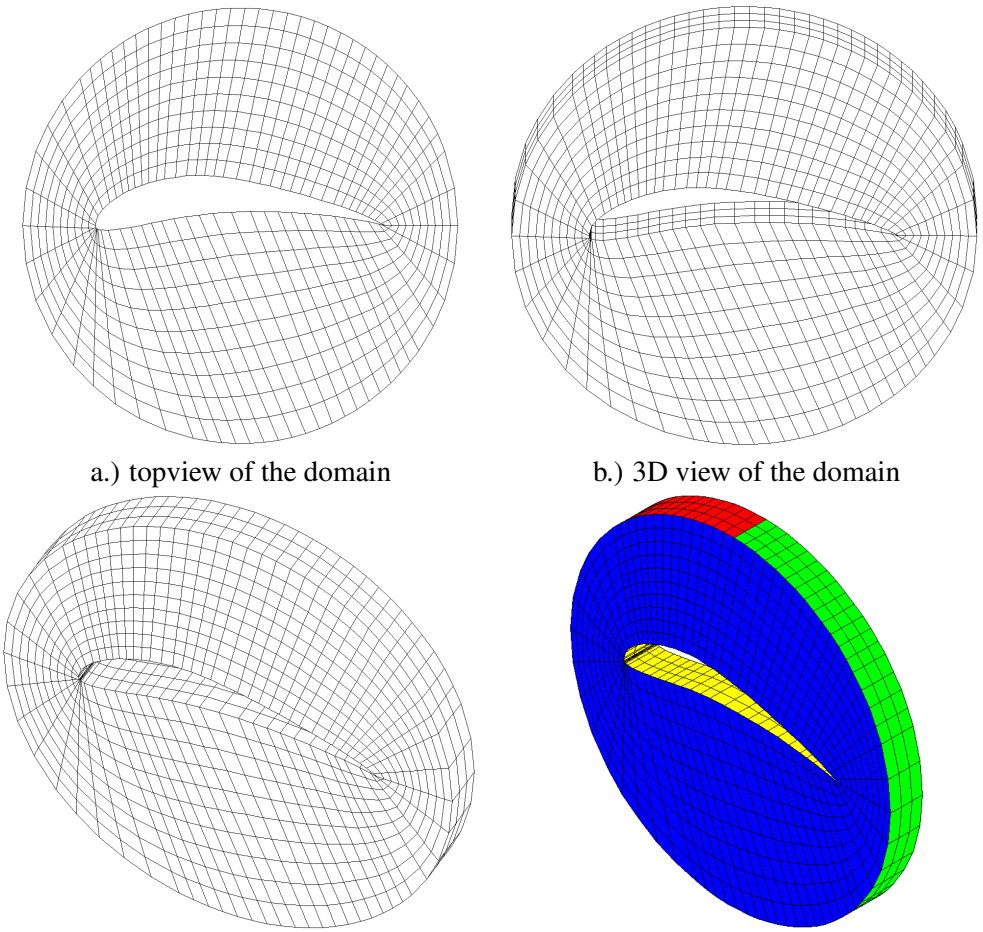
f.) Boundaries of the domain

Figure 6.12: Domain for the comet example 6.2.2

on the angle of attack.
That is the reason why we have initialized the values of the grid's derivative with respect to the angle of attack with finite differences. That is the reason why we have got the strong oscillations on the derivatives of the forces on the airfoil. This is shown in the Figure 6.11.
The conclusion of this example is, that very probably it is possible to apply the AD on the simulation of the 3D case of the Navier-Stokes equations with a moved grid.
At next we show an 3D example, with which it has worked well.

6.2.3 Variable inflow

In this example we choose another approach to avoid the problems of the moving grid example. Here we explain an example with a variable inflow, i.e. we consider the forces on an airfoil for different inflow velocities, different with respect to the amount of the velocity and the direction of the velocity.
At first we explain the geometry of this example and then we present and explain the results of this example.
In the Figure 6.13 the geometry of this example is shown. We consider an Eppler 420 profile



a.) topview of the domain b.) 3D view of the domain

c.) 3D view of the domain d.) 3D view of the domain

Figure 6.13: Domain for the comet example 6.2.3

and as the fluid we consider the air. The boundary conditions are nonslip boundary condition on the airfoil (yellow), we *cut* the circle into two regions. On the left side there are inflow conditions (red) and on the right side there are outflow conditions (green) and the plane faces are symmetry boundary conditions (blue).
Now we explain the both cases.

Variable velocity amount

Here we consider an inflow, the directions u_y and u_z are zero and the velocity in x direction u_x is the variable inflow. In our explicit case $u_x \in [0.1, 0.2]$ and is given by the function $u_x(\alpha) = 0.1 + \alpha$ and $\alpha \in [0, 0.1]$.
Our considered quantity is the forces over the airfoil in x and y direction. For this quantity we consider the derivative respect to α.
We have chosen a stepsize of 0.0001 for α.
The Figure 6.14 shows the process of the forces on the airfoil respect to α, and the associated derivatives and finite differences. Now we explain the results, which are shown detailed in the Figure 6.14. In 6.14 a) we see the process of the derivative of the forces in x direction respect to α, and 6.14 b) shows the same for the forces in y direction. We see, that the derivatives, computed with AD, have the same range as the finite differences. For the most values of α the derivative values, computed with AD, are covered of the FD values. And we see a small dispersion of the FD values. Only for values of α between about 0.005 and 0.025, we see a stronger deviation.
In 6.14 c) the derivatives of the forces in x and y direction are shown respect to α. In 6.14 d) the finite differences of the forces in x and y direction are shown respect to α.
In 6.14 e) the forces in x and y direction are shown respect to α.

Remark 6.2.2
This simple example has shown, that the application of AD on a simulation of the 3D Navier-Stokes equations is possible and the results make sense.

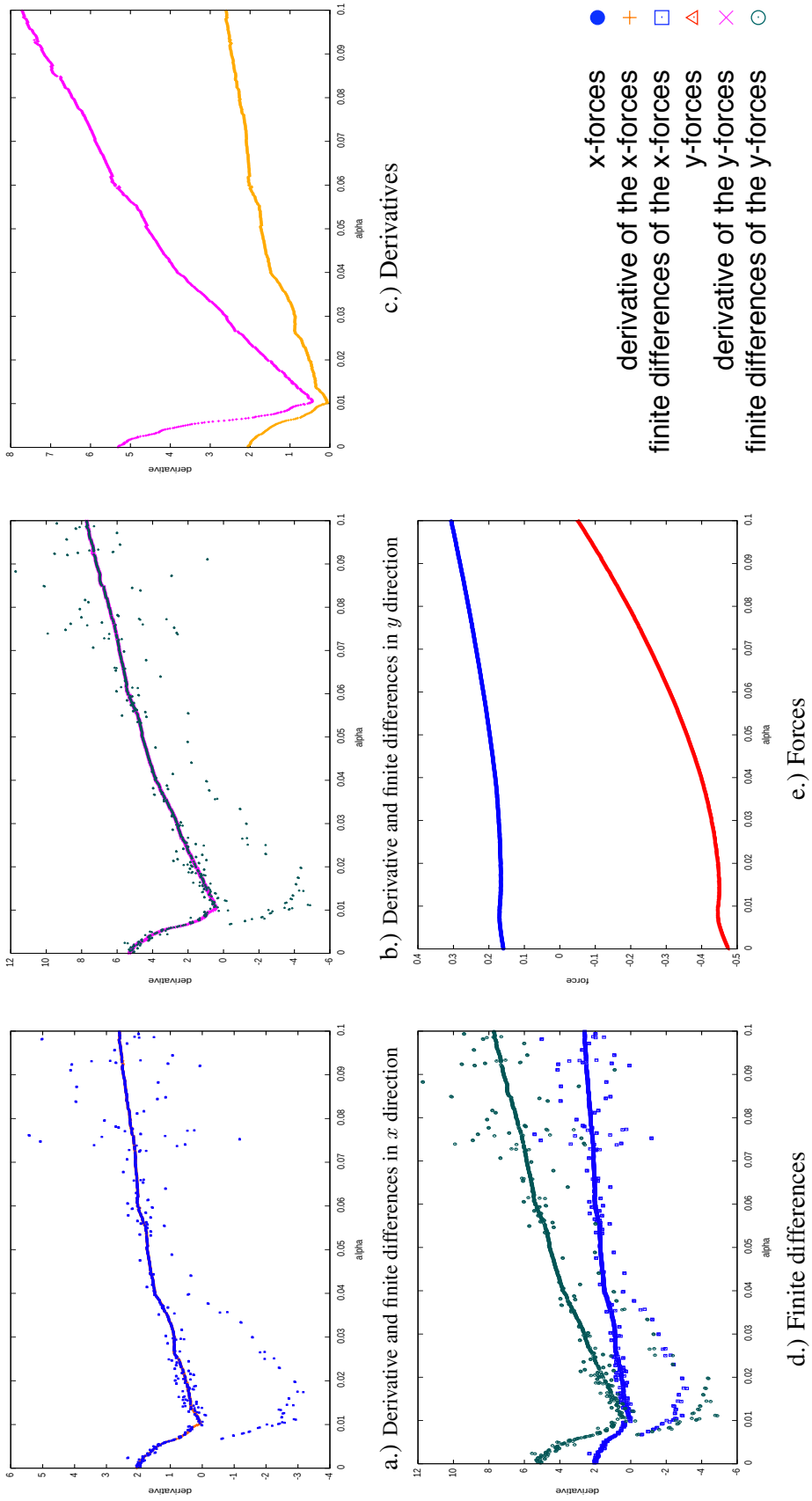


Figure 6.14: Forces, derivatives and finite differences of example 6.2.3

Variable velocity direction

Here we consider an inflow, which is constant with respect to the absolute amount of the velocity (i.e. $\|u\| = const$), and the direction is variable in the x and y direction and constant respect to the z direction.
The input velocity is given with:

$$u = \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} = \begin{pmatrix} 0.15\cos(\alpha) \\ 0.15\sin(\alpha) \\ 0 \end{pmatrix}. \tag{6.1}$$

Our considered quantity is the forces over the airfoil in x and y direction. For this quantity, we consider the derivative with respect to α.
In the Figure 6.15 we see the process of the forces on the airplane for a inflow velocity, which turns. The direction of the turn inflow velocity is given by the Equation (6.1) and α is between 0 and 0.1, the stepsize is 0.0001.
In 6.15 a) and b) we give a comparison of the derivatives, computed with AD, and the finite differences of the forces in x and y direction with respect to the angle α. The graphic 6.15 c) shows the curves of derivatives of the forces in x and y direction respect to α and the same is shown for the finite differences in the graphic d).
The forces in x and y direction respect to α is shown in 6.15 e).

In the Figure 6.16 we show the same problem like in the Figure 6.15. But the range of α and the stepsize of α is changed now $\alpha \in [-0.4, 0.75]$ and the stepsize is 0.0005.
In the Figure 6.16 e.) we see, that the flow is stalled at the point $\alpha \approx 0.53$.
At the stall point we see in the derivatives and in the finite differences a strong discontinuity. This is shown in the Figures 6.16 a-d.). We have a small discontinuity in the derivatives and in the finite differences on $\alpha \approx -0.04$. For $0.53 < \alpha < 0.75$ and $-0.4 < \alpha < -0.04$ the derivatives and the finite differences are not so smooth, the reason of this is a small oscillations in the forces. For $\alpha \in [-\pi, -0.4)$ and $\alpha \in (0.76, \pi)$ we have not shown the forces and the derivatives, because in this ranges the forces are discontinuous and the values of the derivatives are not defined.

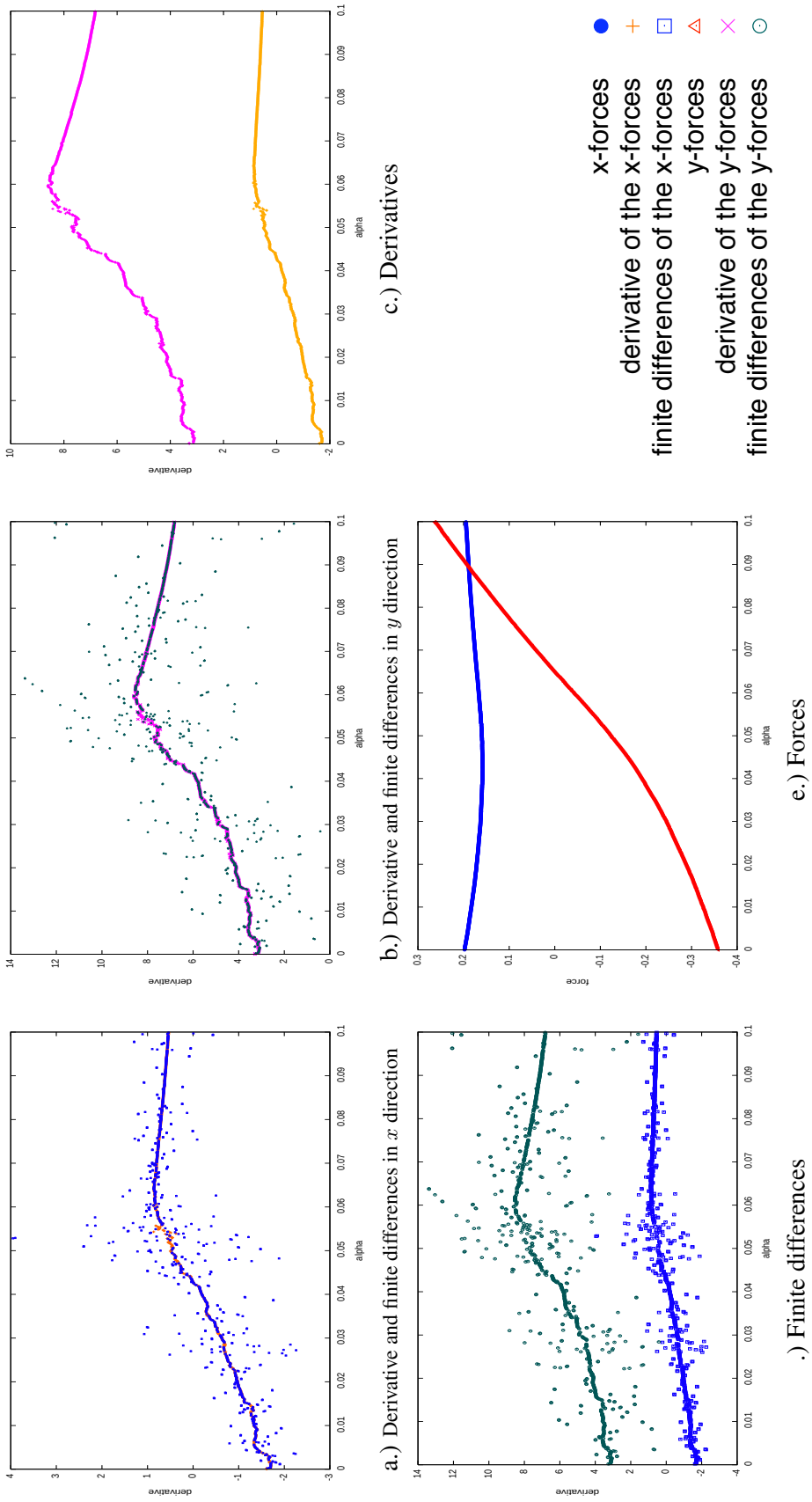


Figure 6.15: Forces, derivatives and finite differences of example 6.2.3

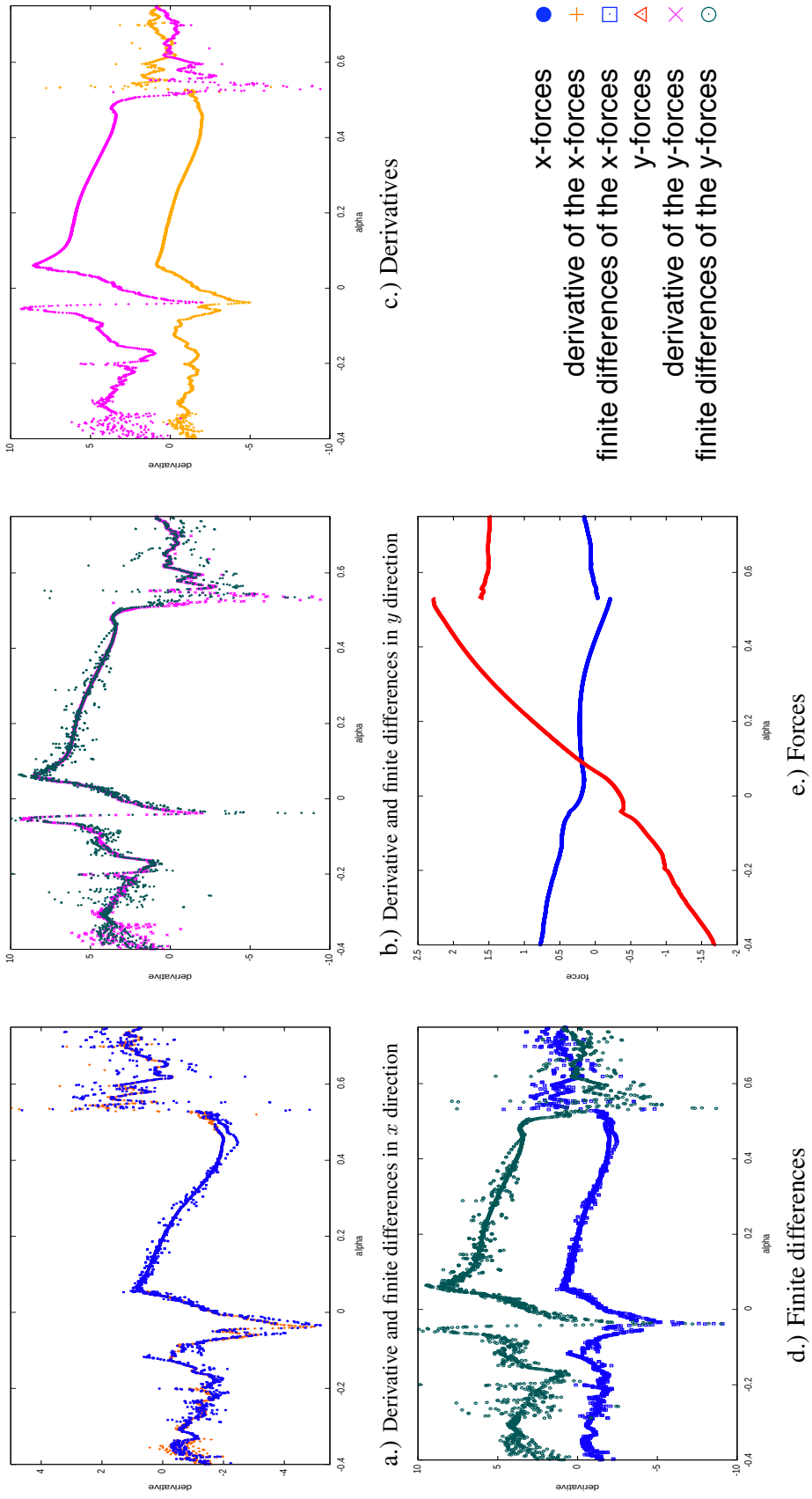


Figure 6.16: Forces, derivatives and finite differences of example 6.2.3

6.2.4 Conclusion for the 3D case

Now we give a conclusion to the numerical experiments in the 3D case.

- In the Subsection 6.2.3 we have seen, that the application of AD on 3D flow simulation is possible, and make sense. And is a good ansatz, which should be advanced.
- Here we have used the AD only to check, if it is possible to use. The next step is to use the AD for the optimization in the 3D case.
- In the Subsection 6.2.3 we have seen, that the application of AD on 3D flow simulation with a moving grid is not trivial. And it not so easy to get the initial derivative for the grid, if the moving is realized in a separate program or tool, which is controlled by a script language.
- A problem is, that the application of AD on the 3D fluid flow simulation software Comet is only possible, if the source code is available and this is normally not given, another idea is, you get the finished program, which is already prepared with AD, but here the problem is, that the application of AD is in the most cases a special version for a fixed problem.

Chapter 7

Results, problems and outlook

This chapter shows the results, the problems and the outlooks of this diploma thesis.

7.1 Results

Now we recall the results of this diploma thesis. We decompose this section in subpoints and then we make a conclusion of all this points.

7.1.1 Basics

Here we present the results of the chapters about the numerical optimization and the Navier-Stokes equations.

Numerical optimization

In this chapter we have explained the most important theoretical basics about optimization and numerical optimization. And we have presented some algorithms for the numerical optimization. And we have given a small comparison of the different approaches for three standard problems.
The issue of this comparison is, that we should normally use optimization algorithms, which use informations about the function, so that we get normally a better efficiency.

Navier-Stokes equations

This chapter has two parts. The first is the part about the derivation of the Navier-Stokes equations, there we give a short derivation of the Navier-Stokes equations by starting with the physical rules and finishing by a mathematical formulation. The second part presents two numerical approaches to compute an approximative solution of the Navier-Stokes equations. We have presented the finite differences method and the finite volume method, we have started by the basics of these methods, have shown some problems of these methods and the application on the Navier-Stokes equations.
The issue of this chapter is, that the simulation of the Navier-Stokes equations is a non trivial problem and the efficient optimization in the simulations is important.

7.1.2 Automatic differentiation

This is one of the main topics of this diploma-thesis. Here we have introduced the automatic differentiation in a formal correct way. Then we have presented different kinds of AD and have compared this kinds with two test examples. The result of this comparison is, that the runtime of a source transformed code in Fortran 90/95 is more efficient than the operator overloading. The answer to the choice of forward or backward mode is not so easy to say, because this is dependent on the problem.
The second part is the presentation of a realization of an operator overloading tool in Fortran 90/95. The message in this chapter is an instruction to realize the operator overloading in a programming language.

7.1.3 Application

Here we have applied the knowledge of the chapter before, i.e. we have applied the AD on the simulation of the Navier-Stokes equations with the target optimization. We have separated this point in two subpoints the 2D case and the 3D case.

2D case

On this point we have considered some different examples. At first we have done a simple example to check, wheather the application of AD on the flow simulation is possible. We have verified this by the consideration of the changing of the forces on a airfoil with respect to the inflow velocity. Then we have compared the finite differences with the values of the automatic differentiation. This example has shown, that the application make sense in the 2D case.
The next example, the optimization of the angle of attack, is a little bit more complex. Here we have considered a constant inflow and have changed the angle of attack, i.e. the grid has moved. An issue of this example is, that a moving grid is not so easy for the application of AD, because, if we change the grid *fast*, we get convergence for the values, but not for the derivatives.
But this example has also shown, that the computation of derivatives with AD has a regularization effect compared with the finite differences. A possible explanation for this effect is, that a small error in the values can imply a big error in the finite differences, but a small error on the values does not imply a big error in the derivatives computed with AD.
The last result of this example is, that the use of optimization algorithms with informations about the derivative are more efficient than others, also in the simulation of fluid flow.

3D case

In this case, we are limited to the application of AD and the validation on the simulation, because this is a non trivial problem.
We have considered two different kinds of problems the first one with a moving grid the second one without a moving grid. The results of the example with moving grid are not so fine because the initialization of geometry variables with a starting derivation is not so easy because the moving of the grid is realized in a separate tool. By initializing the grid geometry with finite differences of the grid geometry, we have got derivatives for the forces over the airfoil in the same range as the finite differences of the forces over the airfoil, but the oscillating of the AD values are very big and larger than the oscillating of the finite differences.

But in the example with a fixed grid and a variable inflow, we have seen, that the application of AD on the 3D simulation of fluid flow is possible. On this example we have seen the regularization effect of the automatic differentiation like in the 2D case.

7.1.4 Conclusion of the results

The main conclusion over all points of precedent results of this work is:
The application of AD in flow simulation for optimization is general possible and useful. But the application is not trivial. The application of the automatic differentiation produces a lot of work for each problem at the moment, but here there exists already an idea to eliminate this problem, more information about this we give in the section outlook with the hybrid method.
We have answered many basic questions, but there exists still many work to make optimization with AD in the flow simulation ready for the practical application.

7.2 Problems

In this section we discuss the problems, which exist by the application of AD on a mathematical problem.

7.2.1 Programming problems

Here we show problems, which come out by the application of AD on programs.

Fortran 77

A problem by this programming languages are, that there exist only one method to apply AD on a program, namely the source transformation. The problem by applying this method is, that the transformed program code is not easy to change after the application of AD, because the transformed code is not easy humanly readable, if the considered problem is a non trivial problem.

C/C++

One problem is the dynamic memory allocation by the programing language C/C++. This can be a problem, if we use an operator overloading tool with the following structure of the `ADType`:

```
...
class ADType
{
 public:
   double value;
   double *derivative;
...
```

And then in the program we use also dynamic memory allocation in the form of `**x`. And after the allocation of the memory for `**x`, we change the size of `*derivative` and then we give the memory of `**x` free. If we do so, it is possible, if the realization of the function to give the memory free it is not good realized, that we get a memory leak.

Existing programs

A big problem is the application of AD on existing programs, which are not designed for the application, of AD. This is the case, if more than one programming language is mixed via an interfaces or an similar way.
Another problem is, that the convergence criteria must be fulfilled also for the AD values.
But an important problem is the initialization of the values by reading files or user input, because all this points must be changed by hand, if we need also initialization for the automatic differentiation.

7.2.2 Mathematical problems

One of the main problems of the application of AD on a problem is, that it is not already clear, that the derivative is well defined. And it is normally not easy to check this, so it is necessary to make a validation of the results with finite differences. But this is also not a warranty, that the results are every times well defined, because it is possible, that in a not checked case, we need e.g. the derivative of $\sqrt{x}$ for the value $x = 0$ and this is not defined. The reason, why it is not easy to check is, that AD is normally not used on small problems.

7.3 Outlook

In this section we present and discuss ideas, which are becoming during the work on this diploma-thesis and especially by the application of AD on finished source code and flow simulation software.

7.3.1 Development of new approaches in AD

Here we present two ideas for the developing of the AD approaches.

Simplifying in source transformation

The idea of this point is combining a source transformation tool with a computer algebra system (e.g. maple, mathematica). The reason of the approach is, that the existing source transformation tools sometimes produce correct derivatives, but the formulas are not simplified. And a computer algebra tool support derivative and also a simplifying functionality. So it is possible to differentiate and simplifying the equation (for reducing the computational complexity) in one step.
To realize this we need a tool, which translates the equations from the programming language in the computer algebra syntax and back. And we also need a functionality to check the dependence, but this is also needed in the normal source transformation.

Hybrid methods

The main focus for this idea is the development of new programs with AD. This approach enable it to separate the development of the algorithm and the application of AD.
In the following we explain the AD hybrid approach on a very simple problem, with the primitivist realization. It is clear, that this kind of realization, with the C preprocessor, is not comfortable enough for a productive work.

At first we must deconstruct the problem in the following parts;

(i) variable declaration,
(ii) variable initialization,
(iii) input,
(iv) output,
(v) algorithm.

In the parts (i)-(iv) there exist normally a difference between program with and without AD, but the part (v) is the important part.

Listing 7.1: AD hybrid main program.

```
1  program hybridAD
2
3  #include"defFile.h"
4
5      !alpha, x1,x2, y1,y2,y3
6  #include"variablesH.f"
7      real(kind(1.d0)) initAlpha,initX1,initX2
8
9  !init variables
10     initAlpha =  2.3
11     initX1    =  3.4
12     initX2    = -1.2
13 #include"initvariablesH.f"
14
15
16  y1 = sin(alpha*x1)
17  y2 = x1*x2+x1**2+sin(x2)
18  y3 = alpha*x1+cos(alpha*x2)
19
20 !output
21     #include"outputH.f"
22
23 end program hybridAD
```

In the Listing 7.1 we show a typical deconstruction of a model problem. In the line 3 we insert the *control* file this file contains only the `#define MACRO 1`, and the word `MACRO` represents, which kind of the program is interested. Line 6 includes the *variable declaration* part, line 13 the *variable initialization* and line 21 the *output* part. It is not necessary to separate the different parts in separate files, but it is clearer. The lines 16-18 are the algorithm and we see, that this part is independent from the case AD or non AD. Now we explain the different parts:

Listing 7.2: AD hybrid variables.

```
#ifdef NONAD
!whithout AD
#undef ADalpha
#undef ADx
    real(kind(1.d0)) alpha
    real(kind(1.d0)) x1,x2
    real(kind(1.d0)) y1,y2,y3
#endif
#ifdef ADalpha
!derivative respect to the variables alpha
#undef ADx
    use AD
    type(forward)    alpha
    type(forward)    y1,y3
    real(kind(1.d0)) y2
    real(kind(1.d0)) x1,x2
    real(kind(1.d0)) init_d(1)
#endif
#ifdef ADx
!derivative respect to the variables x1,x2
    use AD
    type(forward)    x1,x2
    type(forward)    y1,y2,y3
    real(kind(1.d0)) alpha
    real(kind(1.d0)) init
    real(kind(1.d0)) init_d(2)
#endif
```

The Listing 7.2 represent the *variable declaration (i)* part. The realization is very simple. For each possible case the variables are declarated and with the seated `MACRO` we exchange, what is the needed. It is possible to combine this part with a source transformation tool, so that it is enough to say this is a independent and this is a search AD variable and the implicit dependent variables are detected automatically.

Listing 7.3: AD hybrid initialize.

```
#ifdef NONAD
!whithout AD
    alpha =  initAlpha
    x1    =  initX1
    x2    =  initX2
#endif
#ifdef ADalpha
!derivative respect to the variables alpha
    init_d(1) = 1.
    call initAD(alpha,initAlpha,init_d,.true.)
    x1        =  initX1
    x2        =  initX2
#endif
#ifdef ADx
!derivative respect to the variables x1,x2
    alpha =  initAlpha
    init_d(1) = 1.
    init_d(2) = 0.
    call initAD(x1,initX1,init_d,.true.)
    init_d(1) = 0.
    init_d(2) = 1.
    call initAD(x2,initX2,init_d,.true.)
#endif
```

The Listing 7.3 represent the *variable initialization (ii)* part. It is clear, that this part depends directly on the case, which we consider.

Listing 7.4: AD hybrid output.

```
#ifdef NONAD
!whithout AD
    write(*,*) 'y1=',y1,' y2=',y2,' y3=',y3
#endif
#ifdef ADalpha
!derivative respect to the variables alpha
    write(*,*) 'y1=',valueAD(y1),' y2=',y2,' y3=',valueAD(y3)
    write(*,*) 'y1d=',derAD(y1),' y3d=',derAD(y3)
#endif
#ifdef ADx
!derivative respect to the variables x1,x2
    write(*,*) 'y1=',valueAD(y1),' y2=',valueAD(y2),' y3=',valueAD(y3)
    write(*,*) 'y1d=',derAD(y1),' y2d=',derAD(y2),' y3d=',derAD(y3)
#endif
```

The Listing 7.4 represent the *output (iv)* part. We see, that the output changes on the case, because by using AD we are interested normally also on the derivatives.

Listing 7.5: AD hybrid non AD version.

```
program hybridAD
    !alpha, x1,x2, y1,y2,y3
!whithout AD
    real(kind(1.d0)) alpha
    real(kind(1.d0)) x1,x2
    real(kind(1.d0)) y1,y2,y3
    real(kind(1.d0)) initAlpha,initX1,initX2
!init variables
    initAlpha = 2.3
    initX1 = 3.4
    initX2 = -1.2
!whithout AD
    alpha = initAlpha
    x1 = initX1
    x2 = initX2
 y1 = sin(alpha*x1)
 y2 = x1*x2+x1**2+sin(x2)
 y3 = alpha*x1+cos(alpha*x2)
!output
!whithout AD
    write(*,*) 'y1=',y1,' y2=',y2,' y3=',y3
end program hybridAD
```

The Listing 7.5 represent the a compilable program for the non AD case. This is file we get by set the `MACRO` on `NONAD` and use the *gcc* with the following options
`gcc -o tmpFileName -E -P FileName`. The file `tmpFileName` is now directly compilable with a fortran compiler.

7.3.2 Application of AD on flow simulation

Now we give a outlook of the application of AD on flow simulation. It is clear, that in this topic we have not answered all open questions, now we give some outlook about important results.

A general outlook

A general outlook for the application of AD on flow simulation is to expand the problems in 3D for the numerical optimization. In details, that means, we must prepare the code in a form, so that an interface between the simulation tool and the optimization program exist. Also it is necessary to prepare optimization relevant problems for AD.
Another outlook is to eliminate the problems by the moving grid case to enable the consideration of more complex problems.
It is also necessary to make the application of the AD independent of the problem, that means, that we do not need to apply the AD tool for each problem.

Optimal grid generation for AD

In this point we give an idea to avoid the problem of the application of AD on fluid flow simulation with moving grids.
The idea is very simple, but the realization is not so trivial. The idea is to write a grid generation tool for the simulation of the Navier-Stokes equations, which is directly designed for the automatic differentiation. That means, that the data structure is designed for automatic differentiation types. And the moving of the grid is realized, such that we get the informations for the derivative in the moving grid directly.

Bibliography

[Aki03] Ed Akin. *Object-Oriented Programming Via Fortran 90/95*. Cambridge University Press, 01 2003.

[Alt02] Walter Alt. *Nichtlineare Optimierung. Eine Einführung in Theorie, Verfahren und Anwendungen*. Vieweg, 08 2002.

[Cau47] M. Augustin Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes Rendus de l'acadèmie des sciences, Paris*, 1847.

[Fis05] Herbert Fischer. *Algorithmisches Differenzieren*. Lecture notes, 2005.

[Fre97] *Computational Methods for Fluid Dynamics*. Springer, Berlin, 1997.

[Gru04] Roger Grundmann. Cfd lehrbrief. Studienbrief zur Vorlesung Numerische Methoden (CFD) Technische Universität Dresden, 2004.

[Kna00] *Numerik partieller Differentialgleichungen*. Springer, Berlin, 2000.

[Lan74] *Hydrodynamik*. Lehrbuch der theoretischen Physik Band VI. Akademie Verlag, Berlin, 10 1974.

[Lei06] Ralf Leidenberger. Implementierung von algorithmischem differenzieren. Note, 05 2006.

[MG95] Tilman Neunhoeffer Michael Griebel, Thomas Dornseifer. *Numerische Simulation in der Strömungsmechanik*. Vieweg Verlagsgesellschaft, 11 1995.

[PD95] Adreas Hohmann Peter Deufelhard. *Numerische Mathematik :eine algorithmisch orientierte Einführung*. de Gruyter, Berlin, 1995.

[Rie95] Ulrich Rieder. *Operations research I*. University Ulm, 1995.

[Sch97] *Numerische Mathematik*. B. G. Teubner, Stuttgart, 1997.

[Sto99] Josef Stoer. *Numerische Mathematik 1*. Springer, Berlin, 8 edition, 05 1999.

[TB04] Mario Ohlberger Timothy Barth. Finite volume methods: foundation and analysis. *Encylopedia of Computational Mechanics.*, 2004.

[Urb05] Karsten Urban. Numeric i. lecture notes of numeric I, 2004-2005.

[Urb06] Karsten Urban. Numeric ii. lecture notes of numeric II, 2005-2006.

[WF07] Dieter Hoffmann Wilhelm Forst. Optimization - theory and practice. preprint, 2007.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Ulm, den August 30, 2007 _____

(Unterschrift)