Faculty of Science and Technology

Topology Optimization Using the Lattice Boltzmann Method

Master's Thesis in Computational Engineering

by

Angela Hoch

Internal Supervisors

Aksel Hiorth

Knut Erik Teigen Giljarhus

June 15, 2021

“The art of structure is where to put the holes.”

From Robert Le Ricolais, 1894-1977

Abstract

Human design is limited by our ideas of geometries. Topology optimization is a tool to cross that barrier and improve design, especially for applications that are irrevocable once they are in use. This thesis presents a Python implementation of the Lattice Boltzmann Method (LBM) with a porosity model and a separate Python implementation of the adjoint method using the Lagrange multiplier method. The implementation is intended as a basis for topology optimization and is created with the deliberate application of blood flow related topology optimization. The two-dimensional LBM is implemented with a no-slip fullway bounce-back boundary for the closed boundary and solid nodes. For the open boundaries, a non-equilibrium density boundary condition and an equilibrium velocity Boundary Condition (BC) are provided. The porosity model constitutes a hybrid method of the Stokes and Brinkman equations. The implemented test case represents a quadratic domain with a velocity boundary building a flow inlet and a pressure boundary that is the flow outlet. New topologies are created by adjusting the porosity of the nodes, so that the ideal shape of the tube under the specified conditions based on a scalar objective function is obtained, when the LBM is combined with an optimization algorithm as the adjoint method. The adjoint method is applied to a one-dimensional problem, where the geometry of a tube is optimized based on the radius along the axis of the tube. The theory and the implementation procedure are documented in a detailed manner to facilitate the understanding of the methods and provide a foundation for own implementations.

Acknowledgements

This master thesis could never have been completed without good supporters. Firstly, I want to thank my supervisors, Aksel Hiorth and Knut Erik Teigen Giljarhus, for always being supportive, professional, and positive. Both consistently allowed this thesis to be my own work but steered me in the right direction whenever I needed guidance.

Thanks to Yehia for giving me stability in these turbulent times of the pandemic. A special thanks to Hanne, for understanding me with my struggles and for great discussions. I am forever grateful for Chrissy and our friendship that persists regardless of how far we live apart.

Most importantly, none of this could have happened without my family. Thank you to my dad for keeping a neutral perspective when discussing complex decisions with me. Thanks to my mom for reminding me of my achievements because I sometimes forget to appreciate them. And thanks to my brother for understanding me without words.

Finally, I must express my very profound gratitude to our ancestors who created our world with cooperativeness, humbleness, and belief in peace. They laid the foundation of the global network of researchers and nurtured the belief in sharing knowledge which makes science and education what it is today. Moreover, our belief in peace and cooperation made it possible for me to study in this beautiful country that I was not born in.

Contents

List of Figures vii

Code Listings viii

Acronyms ix

List of Symbols xii

1 Introduction 1

1.1 State of the Art . . . 2

1.2 Motivation . . . 4

2 Theory 5

2.1 Lattice Boltzmann Method . . . 5

2.1.1 Macroscopic Moments . . . 9

2.1.2 Closed Boundaries . . . 11

2.1.3 Open Boundaries . . . 12

2.2 Porosity . . . 17

2.3 Topology Optimization . . . 19

2.3.1 Adjoint Method . . . 20

2.3.2 Example . . . 24

3 Implementation 30

3.1 Lattice Boltzmann Method with Porosity Model . . . 30

3.2 Topology Optimization . . . 38

3.2.1 Example 39
3.2.2 Combination with the Lattice Boltzmann Method 46

4 Results **48**

4.1 Lattice Boltzmann Method 48
4.1.1 Computational Setup 48
4.1.2 All Fluid Nodes 50
4.1.3 All Porous Nodes 53
4.1.4 Fluid and Porous Nodes 54
4.2 Topology Optimization 56
4.2.1 Computational Setup 56
4.2.2 Target Pressure with Exponent 3 58
4.2.3 Target Pressure with Exponent 3, Doubled 58
4.2.4 Target Pressure with Exponent 4 59

5 Discussion **63**

5.1 Usage of the Lattice Boltzmann Method 63
5.2 Usage of the Adjoint Method 66
5.3 Blood Flow 66
5.4 Porosity 68
5.5 Unidirectional Inlet Flow 69
5.6 Applicability in Medicine 70

6 Conclusion **71**

Bibliography **73**

Appendices **79**

A User Manual **79**

A.1 Lattice Boltzmann Method 79

A.2 Adjoint Method 80

B Python Code **81**

B.1 Lattice Boltzmann Method 81

B.2 Adjoint Method 96

List of Figures

2.1 Streaming and collision step 6

2.2 D2Q9 velocity set 8

2.3 Bounce-back, no-slip boundary condition 12

2.4 Density boundary 13

2.5 Velocity boundary 16

2.6 Problem definition for adjoint method example 24

3.1 Schematic of one cycle of the LB algorithm 31

3.2 Geometry of the computational domain 31

3.3 Schematic of one cycle of the adjoint method 39

4.1 Porosities of the simulations 50

4.2 Illustration of locations 51

4.3 Velocity profiles with fluid nodes 51

4.4 Pressure drop 52

4.5 Velocity and vorticity of the simulations 53

4.6 Velocity profiles with porous nodes 54

4.7 Velocity profiles with fluid and porous nodes 55

4.8 Velocity profiles in the middle 55

4.9 Velocity profiles at the outlet 56

4.10 Target pressure functions 57

4.11 Pressure along the tube for $-x^3+1$ 58

4.12 Shape of the tube 59

4.13 Pressure along the tube for $-2x^3+2$ 60

4.14 Velocity along the tube 60

4.15 Pressure along the tube for $-x^4+1$ 61

4.16 Objective at the first 50 iterations 62

5.1 Diagonal inlet flow 69

Code Listings

3.1 D2Q9 function 32
3.2 Propagation function 33
3.3 Store bounce-back values 33
3.4 Macroscopic parameter function 34
3.5 Equilibrium distribution function 35
3.6 Open boundaries function 35
3.7 Collision function 37
3.8 Bounce-back collision 37
3.9 Convergence criterion 37
3.10 Domain properties 41
3.11 Objective and convergence 42
3.12 Lagrange multipliers 44
3.13 Sensitivities 45
3.14 Design variable update 46
4.1 Parabolic inlet velocity profile 49
B.1 lbfuns 81
B.2 myLBM 94
B.3 adjointfuns 96
B.4 myAdjoint 100

Acronyms

ATC Adjoint Transpose Convection

BC Boundary Condition

BCs Boundary Conditions

BGK Bhatnagar-Gross-Krook

CFD Computational Fluid Dynamics

DVs Discrete Velocities

DV Discrete Velocity

LB Lattice Boltzmann

LBE Lattice Boltzmann Equation

LBGK Lattice Bhatnagar-Gross-Krook

LBM Lattice Boltzmann Method

List of Symbols

This list describes several symbols that are used within the body of the document.

α Index of the discrete velocity

β Porosity measure

Δt Time step

Δx Distance between nodes

η Factor for the design variable update

κ Design variables

λ Lagrange multipliers

$\mathscr{F}$ Objective function

$\mathscr{L}$ Lagrangian or augmented objective function

ν Kinematic viscosity

Ω_α Collision operator

ρ Density

τ Relaxation time

A $M \times M$ matrix

b Column-vector with the shape M

c_s Factor that relates density to pressure/Speed of sound in the fluid?

$c_\alpha = (c_{\alpha x}, c_{\alpha y})$ Direction of the discrete velocity f_i

d_r Factor to calculate the modified velocity to account for porosity

F Force

f_α Discrete velocity

f_α^{eq} Discrete equilibrium velocity

g Distribution function

g Set of constraints

k Factor for polynomial scaling of the porosity

m Mass

N Number of nodes along the tube

P Number of design variables

p Pressure (LBM), design variables (Adjoint Method)

p_a Adjoint pressure

p_t Target pressure

r Position in space

t Current time

u Column-vector of physical variables with the shape M

u_a Adjoint velocity

$U_r = (U_x, U_y)$ Modified velocity at the position r to account for porosity

$u_r = (u_x, u_y)$ Velocity at the position r

w_α Lattice weights

1. Introduction

Numerical methods and simulation software are essential for the work of engineers. Development can be shortened distinctly by testing prototypes digitally before performing real-world tests. However, most human creations are still based on our ideas of geometries. This is especially critical when the application does not allow realistic testing on prototypes and can hardly be modified after installation as, for instance, in medicine. After suffering a myocardial infarction, also called heart attack, one or more tiny tubes called stents are usually placed in the patient's blood vessel to redistribute plaque and reduce further occlusion (Niccoli and Eitel, 2018). Their curvature is usually determined by the flexibility of the stent and does not necessarily conform to the initial coronary geometry (Zhang et al., 2018), which changes the local wall shear stress distribution in the blood vessel and can result in negative consequences for the patient (LaDisa Jr et al., 2003). Computational Fluid Dynamics (CFD) can predict fluid flow reliably. With the development of new methods to implement stents, it could in combination with optimization algorithms be applied to tailor the curvature of stents to the patient's coronary geometry.

To provide a framework for topology optimization that can for example be applied to stent implementation in human blood vessels, we have chosen to use the LBM. The LBM is attractive, because it is relatively easy to add new physics, thus one can extend the topology algorithm presented in this thesis to study the effect of different fluids and Boundary Conditions (BCs), which subsequently can be used to investigate optimal topology for blood and various substances to manufacture geometries.

1.1 State of the Art

The methodology of all CFD methods consists of

- preprocessing, where the geometry, the mesh, the physical boundaries, and the BC are defined,
- the simulation, where discretized versions of the mathematical equations are solved, and
- postprocessing.

The most common CFD methods discretize with the aid of the finite volume method, the finite element method, or the finite difference method. They numerically solve the conservation equations of macroscopic properties, namely the mass, momentum, and energy conservation. The LBM is another discretization method. It is based on simplified particle dynamics, meaning that the simulation is conducted at particle level. The fluid is modelled by fictive distributions that are moving and interacting across the computational domain. The conservation equations are recovered only indirectly in contrast to conventional discretization methods that are initially built on them. Due to the simulation procedure the LBM is especially simple to implement to study flow in complex geometries. The finite volume method and the finite element method discretize the control volume or the volume of the element, while the finite difference method and the LBM discretize space and time (Mohamad, 2011).

Topology optimization is a method of computation that identifies a favorable topology within a domain given one or more objective functions. The shape can therefore differ from the initially given shape so that any geometry possible within the design space can be obtained. Topology optimization has a wide range of applications in structural design, where conceptual designs as well as detailed shape and sizing optimizations are conducted (Zhu and Gao, 2016; Bendsoe and Sigmund, 2004), in microfluidics (Deng et al., 2018), for electromagnetic waves to optimize microwaves and antennas (Aage et al., 2010) and in dynamics to optimize vibrations and particularly the eigenvalues of a machine or a structure (Bendsoe and Sigmund, 2004). Also, coronary stents are subject to design optimization. The commonly inserted stents are called

balloon-mounted metal mesh permanent scaffolds. They are widened to the desired size by a balloon that is then deflated and removed. Consequently, the geometry of the implemented stent is determined by its flexibility, the surrounding coronary geometry as well as the shape of the inflated balloon. The focus of the conducted optimizations lies on the complex scaffold of the stent that gives stability and widens the blood vessel rather than the curvature of the tube (Ribeiro et al., 2021; James and Waisman, 2016). Current research about the curvature of stents observes their flexibility to enhance general adjustment to the surrounding geometry without deliberately adjusting it to the coronary geometry of individual patients (Saito et al., 2020; Wentzel et al., 2000).

CFD simulations with non-Newtonian fluids, such as blood are performed for several reasons. The main reason is the design of biomedical devices, such as blood pumps, which can also be designed with the aid of topology optimization (Alonso et al., 2019; Romero and Silva, 2017). Besides, there are studies on arterial bypass configurations (Zhang and Liu, 2015; Abraham et al., 2005; Quarteroni and Rozza, 2003) and hemolysis, which indicates blood damage (Alonso and Silva, 2021), for instance.

The LBM is widely used for shape optimization. Kreissl et al. (2011) use an explicit level-set approach to model the fluid-solid interface and determine the geometry of fluidic devices and obstacles that are immersed in flows. De Avila Belbute-Peres et al. (2020) apply a hybrid graph neural network to speed up fluid flow predictions compared to conventional CFD simulations. Obiols-Sales et al. (2020) utilize deep learning algorithms to accelerate the convergence of CFD simulations. The approach shows improvements regarding generalization and the variety of applicable cases like laminar and turbulent flows as well as the possibility to consider convergence constraints. Pingen et al. (2007) and Pingen et al. (2009) utilize the LBM in combination with varying porosity for design optimization with the adjoint method. Later Pingen and Maute (2010) model a non-Newtonian fluid to analyze the 2D dual-pipe problem at different Reynolds numbers. Other studies with the LBM and non-Newtonian fluids have been conducted by Chuanhu et al. (2016) and Vikhansky (2012). Conrad et al. (2015) analyze the accuracy of non-Newtonian Lattice Boltzmann (LB) simulations and describe how to minimize

the error that is induced by the relaxation time.

1.2 Motivation

As it becomes clear in the previous section, topology optimization is subject to research in various fields. Although the use of the adjoint method is widely used in combination with topology optimization, the method is only theoretically described in the available literature. Consequently, inexperienced readers will find it difficult to apply the theoretical approach to their application. The knowledge gap is a detailed description for the implementation of topology optimization using the LBM that is easily understandable even for an inexperienced reader. Only Pingen and Maute (2010) combine topology optimization with the LBM and non-Newtonian fluids. Topology optimization in combination with the LBM to model blood flow for a real-world application has not been conducted so far.

The reader will be provided with the necessary methods and tools to implement topology optimization using the adjoint LBM to examine blood flow related applications. The aim of the study is to facilitate the implementation process to make the available methods more accessible for a wider audience. Therefore, the LBM will be implemented in Python with the required BCs. Thereafter, the porosity model is added. Finally, the adjoint method is described and an example calculation is provided that is supposed to provide the reader with a thorough understanding of how to adjust the calculations to an individual application. As the overarching aim is to provide an easily understandable manual for the implementation, the simplest approaches are preferred throughout this work.

2. Theory

This chapter introduces the main methods applied in this thesis such as the LBM to solve the fluid dynamics, the porosity model and the adjoint method for topology optimization.

2.1 Lattice Boltzmann Method

The LBM can be used for various CFD problems from multiphase flow to heat transfer, flow through porous media and slightly compressible flows (Guo and Shu, 2013; Huang et al., 2015; Evgrafov, 2006). The Lattice Boltzmann Equation (LBE) initially evolved out of the Lattice Gas Cellular Automata with the aim to bypass its main drawbacks, which are statistical noise and the complexity of the collision rule. The LBM is based on kinetic theory of gases, where the time evolution of density distributions on a lattice is simulated instead of individual particles in the lattice gas automata. One of the most used implementations is the Lattice Bhatnagar-Gross-Krook (LBGK) model. This model is regarded as simple and effective despite some new limitations about the single relaxation time for specific applications (Succi, 2001). Kinetic theory is based on statistical mechanics, and it describes the properties of gases on a microscopic level. At a microscopic level, individual molecules are observed and at a macroscopic level, the fluid is viewed as an entity with continuous properties where individual molecules cannot be distinguished. In between lies the mesoscopic scale where molecules are grouped into distributions. On the mesoscopic level, the locations of molecules are not monitored, because they dislimn to the position of the respective distribution. Also, the macroscopic properties of the fluid are not tracked directly but can be computed by the aid of the distributions that describe the density and velocity of the distribution at a certain location. This scale is utilized by the

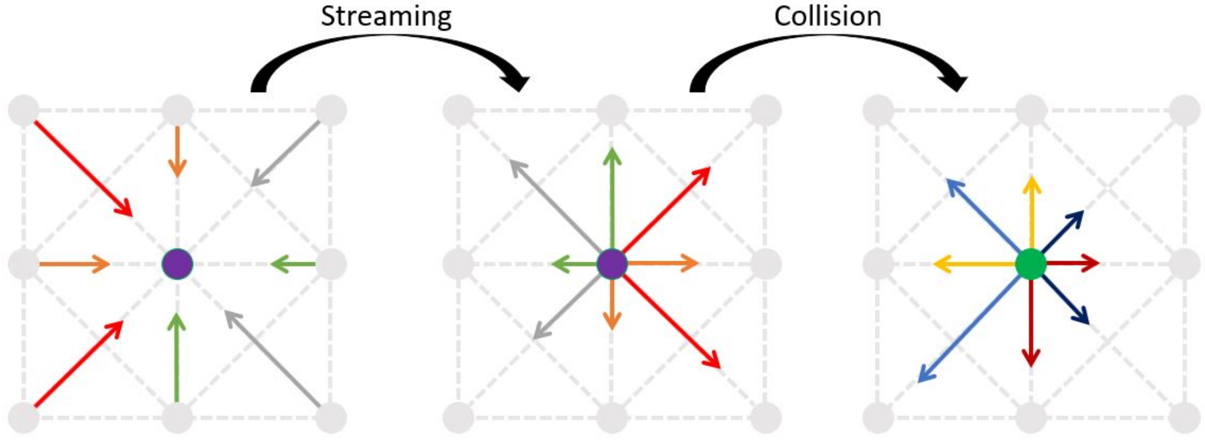


Figure 2.1: Streaming and collision step

LBM, where the fluid is modelled by these fictive particles or distributions that are moving and interacting across the computational domain. The LBM consists of two steps called streaming or propagation and collision or relaxation (Krüger et al., 2017) as visualized in figure 2.1. The collision step describes the redistribution of molecules. During this part of the LBM, molecules can switch from one fictive group to another, while the number of molecules in one distribution remains constant during the streaming step. The equation that describes the redistribution is called collision operator Ω, and obeys the Boltzmann equation (Mohamad, 2011; Mattila, 2010):

$$\frac{\partial f}{\partial t} + u\frac{\partial f}{\partial r} + \frac{F}{m}\frac{\partial f}{\partial u} = \Omega, \tag{2.1}$$

Where the distribution function f is a function of the position r, the velocity u and the time t, and Ω is a function of f. $\frac{F}{m} = \frac{du}{dt}$ according to Newton's second law with the mass m and the force F. The first two terms of equation 2.1 can be viewed as the movement of the particles with the velocity u. The third term represents forces like for example intermolecular forces that appear when two particles collide. Commonly, the simpler Bhatnagar-Gross-Krook (BGK) collision operator is used, which will be introduced in equation 2.3. Before that, the readers attention is directed to the distribution function f and its meaning for the LBM.

The distinctiveness of the LBM lays in the Discrete Velocity (DV) distribution function $f_\alpha(r,t)$ which is also called the particle population. f_α is dependent on space and time and holds the

Discrete Velocities (DVs) of the respective position r at the respective time t. Instead of a continuous distribution function, the velocities are discretized according to a chosen velocity set. Therefore, the DV distribution function describes the velocity in each of the DV directions of the fluid at position r and time t. Each time step during the streaming step the particle population, or element of f, can move along its DV direction to a neighboring node. The number of particles that will indeed move to that node is determined by the collision step, which updates the DV distribution function for the next time step. When choosing the velocity set, few velocities are preferred to minimize the required computational resources such as memory and computing power, while more DVs provide higher accuracy (Krüger et al., 2017). In this work, the velocities are discretized with the D2Q9 velocity set shown in figure 2.2, which is one of the most used velocity sets to solve Navier-Stokes equations in two dimensions. The abbreviation refers to nine DVs in two-dimensional space (Krüger et al., 2017). The table in figure 2.2 shows the explicit form with the indices α, the weights w_α and the direction of the vector by coordinates in x- and y-direction $c_{\alpha x}$ and $c_{\alpha y}$. Figure 2.2 shows one black node in the middle with the DVs indicated as arrows that lead to the neighboring gray nodes. The black numbers indicate the indices α of the vectors and the weights w_α are displayed in red. The indexing α is based on the weight of the respective velocity direction. In the literature, multiple other indexing sequences can be found (Krüger et al., 2017; Pingen et al., 2007; Zou and He, 1997). The weights w_α and the direction of the DVs $c_\alpha = (c_{\alpha x}, c_{\alpha y})$ are required for the computation and will come into focus especially during the implementation process.
Now it is clear that the distribution function f_α contains one particle population for each DV direction at every positions r in the computational domain at the time t. To conduct a simulation, these distributions must stream and collide as shown in figure 2.1. The LBE that evolved out of the Lattice Gas Cellular Automata is obtained when discretizing the Boltzmann equation in velocity space, physical space, and time:

$$f_\alpha(\mathbf{r} + \mathbf{u}_r \Delta t, t + \Delta t) = f_\alpha(\mathbf{r}, t) + \Omega_\alpha(\mathbf{r}, t). \tag{2.2}$$

α	0	1	2	3	4	5	6	7	8
w_α	$\frac{4}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$
$c_{\alpha x}$	0	0	1	0	-1	-1	1	1	-1
$c_{\alpha y}$	0	1	0	-1	0	1	1	-1	-1

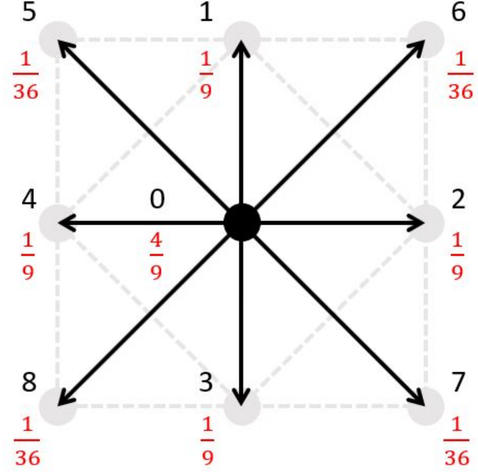


Figure 2.2: D2Q9 velocity set

The discrete-velocity distribution function of the next time step $t + \Delta t$ is obtained by applying the collision operator Ω_α to the particle population of the current time t. The particles represented by $f_\alpha(r,t)$ move to the next node within one time step Δt and with the velocity $\mathbf{u}_r$ (Krüger et al., 2017). The previously mentioned LBGK model refers to the LBM which applies the BGK collision operator:

$$\Omega_\alpha(\mathbf{r},t) = -\frac{f_\alpha(\mathbf{r},t) - f_\alpha^{eq}(\mathbf{r},t)}{\tau}\Delta t, \tag{2.3}$$

Where $f_\alpha^{eq}(\mathbf{r},t)$ is the equilibrium distribution of the DVs and τ is the relaxation time. This collision operator is still the simplest one available for solving Navier-Stokes problems (Krüger et al., 2017) and will be used in this work. The main purpose of the collision operator is to conserve mass and momentum (Krüger et al., 2017). To maintain stability when using the BGK collision operator, all equilibrium populations f_α^{eq} must be positive or zero. The additional stability criterion $\frac{\tau}{\Delta t} > \frac{1}{2}$ follows from the Chapman-Enskog analysis, that is mentioned in more detail in section 2.1.1 (Krüger et al., 2017).

In this work the equilibrium distribution $f_\alpha^{eq}(\mathbf{r},t)$ is determined by the aid of a Taylor series expansion of the Maxwell-Boltzmann equilibrium distribution as described by He and Luo

(1997):

$$f_\alpha^{eq}(\mathbf{r},t) = w_\alpha \rho_r \left[1 + 3(\mathbf{c}_\alpha \mathbf{u}_r) + \frac{9}{2}(\mathbf{c}_\alpha \mathbf{u}_r)^2 - \frac{3}{2}\mathbf{u}_r^2\right], \tag{2.4}$$

Where $\mathbf{u}_r$ is the macroscopic velocity vector, ρ_r is the macroscopic pressure and $\mathbf{c}_\alpha = (c_{\alpha x}, c_{\alpha y})$ is the velocity vector describing the direction of the respective DV. This approach is also used by Pingen et al. (2007) and Pingen et al. (2009) and it is applicable for low Mach number flow conditions. The LBGK equation can be obtained by combining the equations 2.2 and 2.3:

$$f_\alpha(\mathbf{r} + \mathbf{u}_r \Delta t, t + \Delta t) = f_\alpha(\mathbf{r},t) - \frac{f_\alpha(\mathbf{r},t) - f_\alpha^{eq}(\mathbf{r},t)}{\tau}\Delta t. \tag{2.5}$$

The result is then divided into two separate equations for the collision and streaming step, respectively as they will be conducted individually:

$$f_\alpha^*(\mathbf{r},t) = f_\alpha(\mathbf{r},t) - \frac{f_\alpha(\mathbf{r},t) - f_\alpha^{eq}(\mathbf{r},t)}{\tau}\Delta t = (1 - \frac{\Delta t}{\tau}) f_\alpha(\mathbf{r},t) + \frac{\Delta t}{\tau} f_\alpha^{eq}(\mathbf{r},t), \tag{2.6}$$

$$f_\alpha(\mathbf{r} + \mathbf{u}_r \Delta t, t + \Delta t) = f_\alpha^*(\mathbf{r},t). \tag{2.7}$$

For notational clarity, the dependence of f, f^{eq} and Ω on $\mathbf{r}$ and t will not be explicitly written in the following.

2.1.1 Macroscopic Moments

Now it is clear how to determine the equilibrium distribution by the aid of the velocity set suitable to the grid geometry. However, it is still unclear how the non-equilibrium distribution f is obtained so that kinetic theory is preserved. This is described by the Chapman-Enskog

analysis (Chapman, 1939). By the aid of solvability conditions that conserve mass and momentum in combination with, in this case the Taylor expansion and the BGK collision operator, it can be concluded that the LB equation solves the continuity equation and the Navier-Stokes equation (He and Luo, 1997). The Chapman-Enskog analysis is utilized to connect the kinetic theory with the conservation equations and provides the missing piece to calculate the non-equilibrium distribution and the macroscopic moments. However, the LBM does not directly solve the Navier-Stokes equation, but solves the LB equation. Under certain conditions this approximates Navier-Stokes flow. For the D2Q9 velocity set this is achieved by calculating the macroscopic density ρ_r according to equation 2.8 and the macroscopic flow velocity $\mathbf{u}_r = (u_x, u_y)$ by equation 2.9 (Krüger et al., 2017; Zou and He, 1997):

$$\rho_r = \sum_{\alpha=0}^{8} f_\alpha = f_0 + f_1 + f_2 + f_3 + f_4 + f_5 + f_6 + f_7 + f_8, \tag{2.8}$$

$$\begin{aligned} u_x &= \frac{1}{\rho_r} \sum_{\alpha=1}^{8} c_{\alpha x} f_\alpha = \frac{1}{\rho_r} \left[(f_2 + f_6 + f_7) - (f_4 + f_5 + f_8) \right] \\ u_y &= \frac{1}{\rho_r} \sum_{\alpha=1}^{8} c_{\alpha y} f_\alpha = \frac{1}{\rho_r} \left[(f_1 + f_5 + f_6) - (f_3 + f_7 + f_8) \right]. \end{aligned} \tag{2.9}$$

The interested reader is referred to Chapman (1939), Krüger et al. (2017), Mohamad (2011) and Zou and He (1997) for more information and a detailed derivation. When performing the LB simulation, the macroscopic moments must be updated every iteration. Other macroscopic properties like the pressure can be calculated based on the density and velocity. The pressure p is related to the density by the isothermal ideal gas equation of state (Pingen et al., 2007):

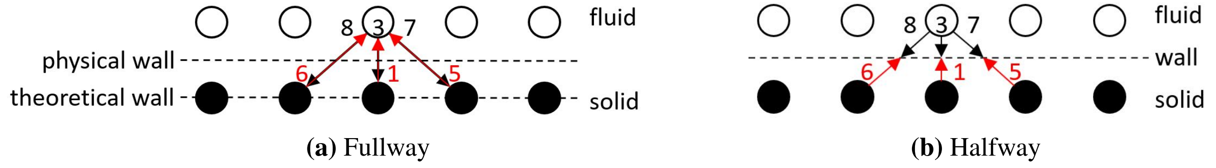
$$p(\mathbf{r},t) = c_s^2 \rho(\mathbf{r},t), \tag{2.10}$$

Where c_s is the lattice speed of sound $c_s = \frac{c}{\sqrt{3}}$ with the lattice speed $c = \frac{\delta x}{\delta t}$.

2.1.2 Closed Boundaries

Describing appropriate BCs is a major challenge in every CFD problem. The boundaries are to represent the interaction between the fluid molecules and the molecules in the solid wall (Succi, 2001). In other cases, the boundaries describe how a force, speed, density, or temperature is applied on the fluid on an open boundary meaning an inlet or outlet where the fluid enters or leaves the computational domain (Krüger et al., 2017). In this work, only a few BCs are applied like a no-slip boundary at solid walls, a velocity condition, and a density condition on the open boundaries. Information about additional BCs can be found in Succi (2001), Krüger et al. (2017), Mohamad (2011) and Guo and Shu (2013).

The no-slip BC generates a fluid velocity of zero at the respective solid surface. The two types of implementations are called on-grid and mid-grid (Succi, 2001) or fullway and halfway (Krüger et al., 2017) bounce-back BC . In both cases the solid wall is aligned with the grid of the lattice. Regarding the fullway bounce-back condition, the theoretical boundary is positioned exactly on the grid line as visualized in figure 2.3a. In that case the velocities at the boundary node are simply reversed. This BC is first-order accurate. In contrast, the halfway bounce-back condition positions the boundary between the last fluid and the first solid nodes as shown in figure 2.3b. This BC is second-order accurate. With both methods the physical boundary lays midway between the last fluid and the first solid node (Krüger et al., 2017). From the perspective of the implementation procedure, the theoretical wall of the fullway bounce-back BC is not aligned with the physical wall, because the distributions are stored at the solid boundary and only reflected at the subsequent time step. However, the physical wall is situated in between the last fluid and the first solid node as for the halfway bounce-back BC. The bounce-back BC enforces the no-slip condition by bouncing the particles back instead of reflecting them forward. Consequently, the fluid does not slip on the wall Krüger et al. (2017). As the fullway bounce-back approach is the easiest way to implement solid walls and as Pingen et al. (2007) applies only this approach for topology optimization, this BC is used in this work.



(a) Fullway **(b)** Halfway

Figure 2.3: Bounce-back, no-slip boundary condition

2.1.3 Open Boundaries

Besides the boundaries representing the interaction between fluid and solid, the open boundaries must be described. There are numerous schemes that can be applied on open boundaries (Yu et al., 2005; Noble et al., 1995; Zou and He, 1997). In the following, the approach of Zou and He (1997) is discussed. The general approach is to calculate the unknown DVs with the known ones. At every boundary node, six DVs are known after streaming and at every corner node, three DVs are known. These and the macroscopic properties that are given at the open boundary are employed to determine the unknown DVs. To do this, the equations 2.8 and 2.9 are used to derive the following expressions:

$$\begin{aligned} \rho_r &= f_0 + f_1 + f_2 + f_3 + f_4 + f_5 + f_6 + f_7 + f_8 \\ \rho_r u_x &= (f_2 + f_6 + f_7) - (f_4 + f_5 + f_8) \\ \rho_r u_y &= (f_1 + f_5 + f_6) - (f_3 + f_7 + f_8), \end{aligned} \tag{2.11}$$

$$\begin{aligned} \rho_r - \rho_r u_x &= f_0 + f_1 + f_3 + 2(f_4 + f_5 + f_8) \\ \rho_r u_x + \rho_r u_y &= (f_1 + f_2 + 2f_6) - (f_3 + f_4 + 2f_8) \\ \rho_r u_x - \rho_r u_y &= (f_2 + f_3 + 2f_7) - (f_1 + f_4 + 2f_5), \end{aligned} \tag{2.12}$$

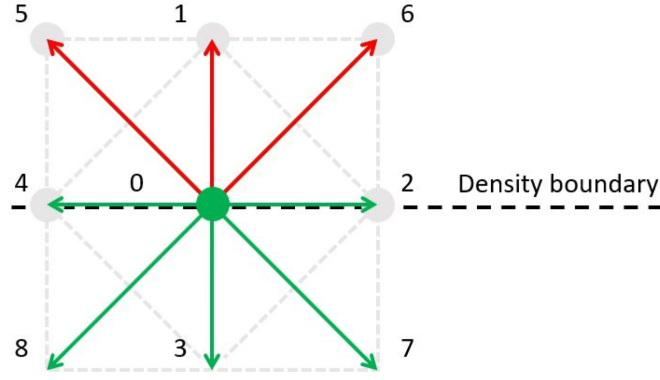


Figure 2.4: Density boundary

$$\begin{aligned}
\rho_r - \rho_r u_y &= f_0 + f_2 + f_4 + 2(f_3 + f_7 + f_8) \\
\rho_r u_x + \rho_r u_y &= (f_1 + f_2 + 2f_6) - (f_3 + f_4 + 2f_8) \\
\rho_r u_y - \rho_r u_x &= (f_1 + f_4 + 2f_5) - (f_2 + f_3 + 2f_7).
\end{aligned} \tag{2.13}$$

The subtractions and additions in equations 2.12 and 2.13 are used to derive the equations for the missing DVs for boundaries along the y- and x-axis, respectively.

Density Boundary Condition

The density boundary is implemented with a non-equilibrium density BC . Due to the direct relation of pressure and density described in equation 2.10, this boundary is also applied when the pressure is prescribed. Figure 2.4 shows an open boundary at the bottom of a domain. The green DVs indicate that they are known after streaming while the red ones are undetermined. Grey dots indicate a neighboring node, and the numbers describe the indices α of the DVs. After the streaming step is conducted, f_0, f_2, f_3, f_4, f_7 and f_8 are known as visualized in figure 2.4. Additionally, $\rho_r = \rho_{out}$ is the density at the outlet, which is known and $\mathbf{u}_{out} = (u_x, u_y) = (0, u_{out})$ is the velocity at the outlet, which is unknown. However, the velocity in x-direction is supposed to be zero ($u_x = 0$) when the fluid flows solely along the y-axis. To determine the missing properties, the equations 2.13 are used as the density boundary lays along the x-axis at the bottom of the domain. $\rho_r - \rho_r u_y$ is reorganized to calculate the unknown velocity along the

y-axis u_y:

$$u_y = 1 - \frac{1}{\rho_{out}} \left[f_0 + f_2 + f_4 + 2(f_3 + f_7 + f_8) \right]. \tag{2.14}$$

Now, only the DVs f_1, f_5 and f_6 remain unknown. The DV f_1 is obtained by applying the bounce-back rule to the non-equilibrium distribution normal to the outlet:

$$f_1 - f_1^{eq} = f_3 - f_3^{eq}. \tag{2.15}$$

Besides, $\rho_r u_x + \rho_r u_y$ and $\rho_r u_y - \rho_r u_x$ in 2.13 are reorganized to obtain the equation for f_6 and f_5 respectively:

$$\begin{aligned} f_1 &= f_1^{eq} + (f_3 - f_3^{eq}) \\ f_5 &= \frac{1}{2}(\rho_r(u_y - u_x) - f_1 + f_2 + f_3 - f_4 + 2f_7) \\ f_6 &= \frac{1}{2}(\rho_r(u_x + u_y) - f_1 - f_2 + f_3 + f_4 + 2f_8). \end{aligned} \tag{2.16}$$

The expressions in equation 2.16 are further simplified by Zou and He (1997). The equilibrium distributions for f_1^{eq} and f_3^{eq} are inserted according to the Taylor expansion in equation 2.4 into:

$$\begin{aligned} f_1 &= f_1^{eq} + (f_3 - f_3^{eq}) \\ &= f_3 + f_1^{eq} - f_3^{eq} \\ &= f_3 + \frac{1}{9}\rho_{out}\left[1 + 3u_y + \frac{9}{2}u_y^2 - \frac{3}{2}u_y^2\right] - \frac{1}{9}\rho_{out}\left[1 - 3u_y + \frac{9}{2}u_y^2 - \frac{3}{2}u_y^2\right] \\ &= f_3 + \frac{1}{3}\rho_{out}u_y + \frac{1}{3}\rho_{out}u_y \\ &= f_3 - \frac{2}{3}\rho_{out}u_y. \end{aligned} \tag{2.17}$$

The result is then inserted into the expressions for f_5 and f_6 in equation 2.16:

$$
\begin{aligned}
f_6 &= \frac{1}{2}(\rho_r(u_x+u_y) - f_1 - f_2 + f_3 + f_4 + 2f_8) \\
&= f_8 - \frac{1}{2}(f_2 - f_4) - \frac{1}{2}(f_1 - f_3) + \frac{1}{2}\rho_{out}u_x + \frac{1}{2}\rho_{out}u_y \\
&= f_8 - \frac{1}{2}(f_2 - f_4) - \frac{1}{2}(f_3 - \frac{2}{3}\rho_{out}u_y - f_3) + \frac{1}{2}\rho_{out}u_x + \frac{1}{2}\rho_{out}u_y \\
&= f_8 - \frac{1}{2}(f_2 - f_4) - \frac{1}{3}\rho_{out}u_y + \frac{1}{2}\rho_{out}u_x + \frac{1}{2}\rho_{out}u_y \\
&= f_8 - \frac{1}{2}(f_2 - f_4) + \frac{1}{6}\rho_{out}u_y + \frac{1}{2}\rho_{out}u_x \\
&= f_8 - \frac{1}{2}(f_2 - f_4) + \frac{1}{6}\rho_{out}u_y,
\end{aligned}
\tag{2.18}
$$

$$
\begin{aligned}
f_5 &= \frac{1}{2}(\rho_r(u_y-u_x) - f_1 + f_2 + f_3 - f_4 + 2f_7) \\
&= f_7 + \frac{1}{2}(f_2 - f_4) + \frac{1}{6}\rho_{out}u_y.
\end{aligned}
\tag{2.19}
$$

As $u_x = 0$ at the pressure boundary, the term $\frac{1}{2}\rho_{out}u_x$ can be neglected when calculating f_5 and f_6. Now, all the unknown DVs for a density boundary at the bottom of the domain can be calculated. f_1, f_5 and f_6 are obtained with equation 2.17, 2.19 and 2.18 respectively. Every iteration, the unknown DVs of the nodes on the density boundary must be calculated accordingly. For density boundaries at other locations like the top, the right or the left, the equations for the unknown DVs can be derived by the same concept.

Velocity Boundary Condition

The other type of open boundary that will be used in this work is an equilibrium distribution velocity condition. Figure 2.5 shows a node on that boundary on the left of the computational domain. The green DVs indicate that they are known after streaming while the red ones are undetermined. Grey dots indicate a neighboring node, and the numbers describe the indices α of the DVs.

After the streaming step is conducted, f_0, f_1, f_3, f_4, f_5 and f_8 are known. Additionally, $\mathbf{u}_{in} =$

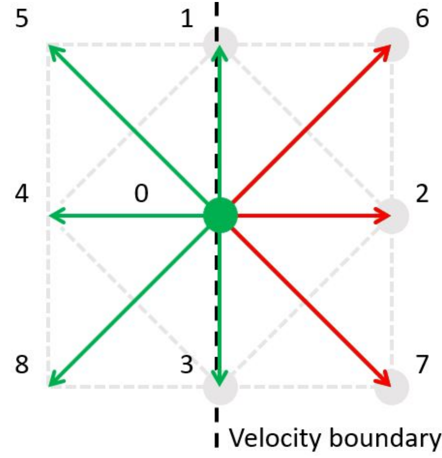


Figure 2.5: Velocity boundary

$(u_x, u_y) = (u_{inx}, 0)$ is the velocity at the inlet, which is known and $\rho_r = \rho_{in}$ is the density at the inlet, which is unknown. To determine the missing properties, the equations 2.12 are used as the velocity boundary lays along the y-axis on the left of the domain. $\rho_r - \rho_r u_x$ is reorganized to calculate the unknown density ρ_{in}:

$$\rho_{in} = \frac{1}{1-u_x}\left[f_0 + f_1 + f_3 + 2(f_4 + f_5 + f_8)\right]. \tag{2.20}$$

Now, only the DVs f_2, f_6 and f_7 remain unknown. Again, the bounce-back rule is used to obtain the DV f_2. Besides, $\rho_r u_x + \rho_r u_y$ and $\rho_r u_x - \rho_r u_y$ in 2.12 are reorganized to obtain the equation for f_6 and f_7 respectively:

$$\begin{aligned}
f_2 &= f_2^{eq} + (f_4 - f_4^{eq}) \\
f_6 &= \frac{1}{2}(\rho_r(u_x + u_y) - f_1 - f_2 + f_3 + f_4 + 2f_8) \\
f_7 &= \frac{1}{2}(\rho_r(u_x - u_y) + f_1 - f_2 - f_3 + f_4 + 2f_5).
\end{aligned} \tag{2.21}$$

As with the pressure boundary, the expressions in 2.21 can be further simplified to:

$$
\begin{aligned}
f_2 &= f_4 + \frac{2}{3}\rho_{in}u_x \\
f_6 &= f_8 - \frac{1}{2}(f_1 - f_3) + \frac{1}{6}\rho_{in}u_x \\
f_7 &= f_5 + \frac{1}{2}(f_1 - f_3) + \frac{1}{6}\rho_{in}u_x.
\end{aligned} \tag{2.22}
$$

Now, all the unknown DVs for a velocity boundary at the right of the domain can be calculated. f_2, f_6 and f_7 are obtained with the equations 2.22. As with the pressure boundary, velocity boundaries at other locations than the right would require the derivation for the respective unknown DVs. For the following application, only the described open boundaries are applied, so that all the required theory to implement the LBM is covered.

2.2 Porosity

To conduct topology optimization, a variable in the system must be adjustable. In this case, the porosity of the nodes in the domain will be subject to change. Depending on the porosity model, the key variables defining the porous medium can vary. Pingen et al. (2007) adopts the LBM porosity model by Spaid and Phelan (1997) which will also be applied in this work. The porosity of the system is described by the kinematic viscosity ν and the permeability k_{tow}. Viscosity can be viewed as a measure of the internal friction between the molecules of a fluid. Consequently, the viscosity defines the resistance to deformation of a given fluid (Barnes, 2002). The permeability describes the rate at which a fluid can pass through a permeable material and is governed by Darcy's law (Hwang and Advani, 2010). For the porosity model by Spaid and Phelan (1997), a parameter β is introduced that defines the porosity of each node. For the D2Q9 velocity set, the kinematic viscosity can be described by:

$$
\nu = (\tau - \frac{1}{2})c_s^2\Delta t = \frac{2\tau - 1}{6}, \tag{2.23}
$$

And recovers the Brinkman equation when $\beta = \frac{\nu}{k_{tow}}$. More information about the recovery of the Brinkman equation is provided by Spaid and Phelan (1997).

The adjustable parameter for the topology optimization will be the porosity measure of the nodes β. As the relation between the viscosity and the permeability defines the porosity measure, only one of them can be fixed for the simulation. Unlike the suggestion by Spaid and Phelan (1997), the viscosity is set, the porosity measure is optimized, and the permeability is adjusted accordingly. The recommendation of Spaid and Phelan (1997) is to predefine the permeability, so that the viscosity and the porosity can be adjusted by the optimization algorithm. Due to the interconnection of the viscosity and the relaxation time, this leads to a changing upper limit for the porosity measure. As the level of porosity is set by the algorithm, it is more convenient to have a defined range of possible values rather than a dynamic one. Consequently, the viscosity is chosen to be constant for all nodes and time steps. However, the porosity and the ability of a fluid to pass through the computational domain of each node is determined individually. Thereby, a solid node is represented by a low-porous material that does not let fluid pass through it. By contrast, a fluid node is modelled by a highly porous medium through which fluid can pass easily. This approach enables a smooth transition between fluid and solid nodes.

As shown in equation 2.23, the viscosity has a direct relation to the relaxation time. The relaxation time τ mentioned in equation 2.3 has a major influence on the collision step. When $\frac{\tau}{\Delta t} > 1$, $f(t+\Delta t)$ continuously approaches f^{eq}. However, the bigger τ, the smaller is the influence of f^{eq} so that the equilibrium distribution has a negligible influence for large relaxation times. This is called under-relaxation. When $\frac{\tau}{\Delta t} = 1$, $f(t+\Delta t)$ equals the equilibrium distribution f^{eq}, which is called full relaxation. When $0.5 < \frac{\tau}{\Delta t} < 1$, $f(t+\Delta t)$ fluctuates around f^{eq} and continuously approaches it when regarding successive time steps, which is called over-relaxation. To maintain stability, the relaxation time cannot be smaller than 0.5 as this would lead to a fluctuation with increasing amplitude. (Krüger et al., 2017)

When adding the porosity model to the LBM a modification of the equilibrium distribution

calculation is required. The velocity $u(t,r)$ is replaced with the velocity $U(t,r)$:

$$U(t,r) = (1-(\beta(r)\tau)^k)u(t,r) = d(r)u(t,r), \tag{2.24}$$

Where $0 \leq d_r \leq 1$, and 1 translates to a pure fluid while 0 represents a fluid and everything in between is a porous medium. k is a factor for polynomial scaling to improve convergence. Pingen et al. (2007) recommend $k \in [2,3]$ for best results. For the porosity model with the introduced variable d_r the Taylor expansion in equation 2.4 is modified to:

$$f_\alpha^{eq} = w_\alpha \rho_r \left[1 + 3(d_r \mathbf{c}_\alpha \mathbf{u}_r) + \frac{9}{2}(d_r \mathbf{c}_\alpha \mathbf{u}_r)^2 - \frac{3}{2}(d_r \mathbf{u}_r)^2\right], \tag{2.25}$$

Where $\beta\tau \in [0,1]$ (Evgrafov, 2005). This model is not limited to specific kinds of flow and is therefore applicable for the same range of applications as the LBM. It should be noted that the porosity model is not intended to represent porous media in this implementation but to provide a smooth transition from liquid to solid material for the optimization method that is described in the following chapter.

2.3 Topology Optimization

To find the optimal topology of the problem without optimization algorithm, every combination of solid and fluid nodes must be calculated and the combination that meets the objectives best would be the solution. In a problem with as many parameters as nodes, this requires enormous computational resources. For a computational domain with $10*10 = 100$ nodes (excluding the marginal boundaries) and two possible properties that are solid or fluid, this leads to $2^{100} = 1,27*10^{30}$ simulations. Consequently, this approach is infeasible, especially for real-world applications with significantly higher node numbers. The adjoint method provides a solution to this issue by calculating the gradient of the cost function independently from the number of optimization parameters (Cheylan et al., 2019).

Optimization approaches with the adjoint method in combination with the LBM are chosen by Pingen and Maute (2010) for topology optimization with non-Newtonian fluids, Dugast et al. (2018) for thermal fluid flows, Cheylan et al. (2019) for aerodynamic applications, Rutkowski et al. (2020) for wing movement, Li et al. (2018) for airfoil design and Vergnault and Sagaut (2014) for noise control problems. Besides, Hamed and Ramin (2020) use an adjoint LBM for unsteady flow. Hekmat and Mirzaei (2014) extract continuous and discrete adjoint LB equations that can be used for steady and unsteady flow. In this case, the adjoint method is implemented in the LBM instead of running the LB simulation multiple times with the results of the adjoint method. The adjoint method is explained in a general manner by Cao et al. (2003), Errico (1997), Strang (2007) and Johnson (pers. comm., 2021) who gave a lecture on adjoint methods with notes on `https://math.mit.edu/~stevenj/18.336/adjoint.pdf`, while Plessix (2006) and Cheylan et al. (2019) describe the steps of the adjoint Lagrange multiplier method in more detail and are recommended for additional guidance and a deeper understanding.

2.3.1 Adjoint Method

First, the theory of the adjoint method is explained for linear equations to introduce the method in an understandable way. Later the theory of the Lagrange multiplier method and a corresponding example are presented.

Linear System

The aim is to optimize the system $A(\kappa)u = b(\kappa)$ with respect to the scalar function $\mathscr{F}(u, \kappa)$. Thereby, $Au = b$ is a system of M equations that are dependent on the design variables κ. A constitutes a $M \times M$ matrix and u is the column-vector that solves the system. In the literature, $\mathscr{F}$ is called objective function, cost function or constraint. The objective function $\mathscr{F}$ is dependent on the solution of the linear equation system $Au = b$ and the design variables κ which the linear equation system depends on. Since A and b are dependent on κ, the gradient of the function $\mathscr{F}$ with respect to the design variables κ must be obtained for the topology optimization

and can be calculated as follows (Johnson, pers. comm., 2021):

$$\frac{d\mathscr{F}}{d\kappa} = \frac{\partial \mathscr{F}}{\partial \kappa} + \frac{\partial \mathscr{F}}{\partial u}\frac{\partial u}{\partial \kappa}. \tag{2.26}$$

This equation describes the sensitivities of the objective function with respect to changes in the design variables. Since the function $\mathscr{F}(u,\kappa)$ is known, $\frac{\partial \mathscr{F}}{\partial \kappa}$ and $\frac{\partial \mathscr{F}}{\partial u}$ can be calculated analytically. However, calculating $\frac{\partial u}{\partial \kappa}$ is difficult as u can be obtained only by calculating $A(\kappa)$ and $b(\kappa)$, so that $A(\kappa)u = b(\kappa)$ must be differentiated by κ first. When M is large and the number of design variables P is large as well, this leads to a cumbersome calculation (Strang, 2007):

$$\frac{\partial u}{\partial \kappa} = \underbrace{A^{-1}}_{M\times M}\underbrace{(\frac{\partial b}{\partial \kappa} - \frac{\partial A}{\partial \kappa}x)}_{M\times P}. \tag{2.27}$$

Here the adjoint method is applied to rephrase the problem and avoid the calculation of the term $\frac{\partial u}{\partial \kappa}$ as follows:

$$\frac{\partial \mathscr{F}}{\partial u}\frac{\partial u}{\partial \kappa} = \underbrace{\frac{\partial \mathscr{F}}{\partial u}}_{1\times M}\left[\underbrace{A^{-1}}_{M\times M}\underbrace{\left(\frac{\partial b}{\partial \kappa} - \frac{\partial A}{\partial \kappa}u\right)}_{M\times P}\right] = \overbrace{\left[\underbrace{\frac{\partial \mathscr{F}}{\partial u}A^{-1}}_{1\times M}\right]}^{\lambda^T}\underbrace{\left(\frac{\partial b}{\partial \kappa} - \frac{\partial A}{\partial \kappa}u\right)}_{M\times P}. \tag{2.28}$$

Depending on the problem, the exact approach of the adjoint method slightly varies. The adjoint method can be applied to linear systems, nonlinear systems and dualities (Strang, 2007). For topology optimization commonly the Lagrange multiplier method is applied, which is described next. For that reason, the first part of the rephrased equation 2.28 is denoted as λ^T. The Lagrange multipliers are called λ and they can be viewed as the specified part of the rearranged expression.

Lagrange Multiplier Method

The next problem is nonlinear and more abstract. The aim is to optimize the scalar function $\mathscr{F}(u,\kappa)$ with respect to design variables κ while satisfying the set of equations $g(u,\kappa)=0$. Again, κ represents the design variables and u is a vector of physical variables. In topology optimization, $g(u,\kappa)$ is viewed as a set of constraints of the system. Universally, g is a set of functions. The Lagrange multiplier method is used to combine the scalar function and the constraints. The augmented objective function $\mathscr{L}(u,\kappa,\lambda)$, the Lagrangian function, is defined by introducing Lagrange multipliers λ so that:

$$\mathscr{L}=\mathscr{F}-\lambda^T g. \tag{2.29}$$

One Lagrange multiplier is added to each function g as a factor. While $\lambda^T g$ are subtracted from $\mathscr{F}$ in this explanation, the sign is irrelevant as the factor λ will be chosen based on the respective definition. In the literature, both expressions can be found. When the Lagrangian equals zero, the constraints are fulfilled and the objective function indicates the ideal solution. The gradient of the Lagrangian with respect to the physical parameters u is called adjoint equation:

$$\frac{d\mathscr{L}}{du}=\frac{d\mathscr{F}}{du}-\lambda^T\frac{\partial g}{\partial u}. \tag{2.30}$$

The adjoint equations are needed to determine the Lagrange multipliers. The last term $-\lambda^T\frac{\partial g}{\partial u}$ is only a partial derivative with respect to u to later recover the sensitivity of the Lagrangian with respect to the design variables. By choosing λ, so that $\frac{d\mathscr{L}}{du}=0$, the gradient of the Lagrangian equals equation 2.30 (Strang, 2007). In this way, the computation of the sensitivity of the Lagrangian with respect to the design variables can be expressed as:

$$\nabla\mathscr{L}=\frac{d\mathscr{L}}{d\kappa}=\frac{d\mathscr{F}}{d\kappa}-\lambda^T\frac{\partial g}{\partial \kappa}. \tag{2.31}$$

By introducing λ, the order of the computations is adjusted, so that the physical parameters of the system are determined before the design variables are changed. This simplifies the calcula-

tion when the number of design variables is higher than the number of physical parameters. By choosing the Lagrange multipliers as described, the gradient of the Lagrangian is equivalent to the gradient of the objective function $\nabla\mathscr{L} = \nabla\mathscr{F}$. The update of the design variables can now be calculated independently of the other design variables by an iterative algorithm such as for example the steepest descent method or the Newton method.

The Lagrange multiplier method can be applied to the linear problem that was discussed previously. In that case, the linear system $Au = b$ must be reorganized so that $g(u,\kappa) = 0$ resulting in $g = Au - b$. Next, λ is determined by setting equation 2.30 to zero. With $\frac{\partial g}{\partial u} = \frac{\partial(Au-b)}{\partial u} = A$ the equation can be simplified to:

$$\begin{aligned} \frac{d\mathscr{L}}{du} = \frac{d\mathscr{F}}{du} - \lambda^T A &= 0 \\ \lambda^T A &= \frac{d\mathscr{F}}{du} \\ \lambda^T &= \frac{d\mathscr{F}}{du} A^{-1}, \end{aligned} \tag{2.32}$$

Which equals the term that is denoted as λ^T in equation 2.28.

To apply the adjoint Lagrange multiplier method on a topology optimization problem, the description above can be summarized to these steps:

- Define the problem including objective function $\mathscr{F}$, governing equations g, and design variables κ.
- Define the Lagrangian $\mathscr{L}$ with the Lagrange multipliers λ and the constraints g.
- Differentiate the Lagrangian $\mathscr{L}$ with respect to the design variables κ and simplify to obtain the adjoint equations.
- Identify the adjoint equations where the physical variables u are dependent on the change of the design variables κ which should look similar to the governing equations. Determine the Lagrange multipliers λ so that $\frac{d\mathscr{L}}{du} = 0$. Rename the Lagrange multipliers if desired.

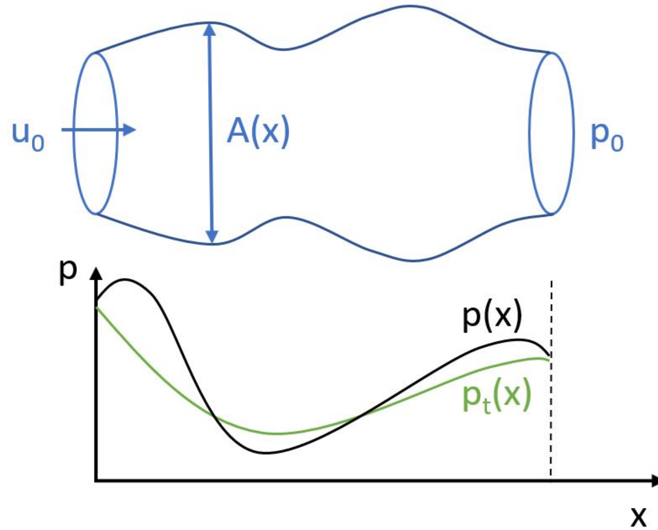


Figure 2.6: Problem definition for adjoint method example

- Find the sensitivity variables $\frac{\partial \mathscr{L}}{\partial \kappa} = \frac{\partial \mathscr{F}}{\partial \kappa}$.
- Conduct the optimization with the obtained sensitivities.

2.3.2 Example

Now the adjoint method is applied to a 1D problem to enhance understanding.

Problem Definition

The problem describes a pipe with varying diameter which is defined by multiple shape parameters that are subject to optimization. The inlet has a velocity BC and the outlet a pressure boundary. The pressure $p(x)$ depending on the position in the pipe x is supposed to match a target pressure curve $p_t(x)$ as visualized in figure 2.6. The target pressure is predefined. Additionally, one set of initial shape parameters must be given. These parameters are then adjusted by the adjoint method until the pressure curve matches the target pressure sufficiently. For this problem, the Euler equations for mass and momentum conservation are used as governing equations:

$$\frac{\partial}{\partial x}(Au) = 0, \tag{2.33}$$

$$u\frac{\partial u}{\partial x}+\frac{\partial p}{\partial x}=0, \tag{2.34}$$

Where A is the diameter of the pipe, x is the position along the pipe, p is the pressure, and u is the velocity. The design variables κ define the area of the pipe:

$$A(x,\kappa_1,\kappa_2,\kappa_3,\kappa_4)=\kappa_1(-x^3+3x^2-3x+1)+\kappa_2(3x^3-6x^2+3x)+\kappa_3(-3x^3+3x^2)+\kappa_4x^3, \tag{2.35}$$

And the objective function $\mathscr{F}$ is:

$$\mathscr{F}=\frac{1}{2}\int_0^1[p-p_t]^2\,dx. \tag{2.36}$$

The observed part of the pipe reaches from $x=0$ to $x=1$ which are incorporated into the objective as lower and upper boundaries. When the pressure matches the target pressure, the objective equals zero. The further they are apart, the bigger the objective becomes. The optimization algorithm is supposed to move the pressure closer to the target pressure which is done by minimizing the objective. After the problem is defined the adjoint method is applied.

Definition of the Lagrangian

The example is governed by the mass and momentum conservation as constraints g. Both must be zero to maintain the physical properties of the system. Theoretically, g is described as a set of equations. To conduct the adjoint method, they can be incorporated into the Lagrangian as follows:

$$\mathscr{L}=\mathscr{F}+\int_0^1\lambda_1\frac{\partial}{\partial x}(Au)dx+\int_0^1\lambda_2\left(u\frac{\partial u}{\partial x}+\frac{\partial p}{\partial x}\right)dx, \tag{2.37}$$

Where the constraints are multiplied with the Lagrange multipliers λ_1 and λ_2 and define the Lagrangian together with the objective function. The Lagrangian of the best possible solution equals zero. To reinforce physical behavior, the accumulated deviations from the governing

equations are added to the objective function. The Lagrange multipliers define how strongly the governing equation must be satisfied. To compute the constraints, the governing equations are integrated over the coordinates of the system, which is x in this example. The whole computational domain that is subject to optimization is covered by setting the boundaries of the integral accordingly. In this example the lower boundary is 0 and the upper boundary is 1.

Derivation of the Lagrangian

Even though the adjoint equations have been determined by calculation equation 2.30 in section 2.3.1, the same result can be achieved by differentiating with respect to κ. To explain this, the Lagrangian $\mathscr{L}$ must first be differentiated with respect to the design variables as follows:

$$\frac{\partial \mathscr{L}}{\partial \kappa} = \frac{\partial \mathscr{F}}{\partial \kappa} + \int_0^1 \lambda_1 \frac{\partial}{\partial \kappa}\frac{\partial}{\partial x}(Au)dx + \int_0^1 \lambda_2 \frac{\partial}{\partial \kappa}\left(u\frac{\partial u}{\partial x} + \frac{\partial p}{\partial x}\right)dx. \tag{2.38}$$

It can be clearer to first differentiate the objective function $\mathscr{F}$ and the governing equations with respect to the design variables κ, before inserting the derivatives into equation 2.38. The objective function defined in 2.36 and the Euler equations defined in 2.33 and 2.34 are differentiated, respectively:

$$\frac{\partial \mathscr{F}}{\partial \kappa} = \frac{1}{2}\int_0^1 \frac{\partial}{\partial \kappa}[p - p_t]^2 dx = \int_0^1 [p - p_t]\frac{\partial p}{\partial \kappa}dx, \tag{2.39}$$

$$\frac{\partial}{\partial \kappa}\frac{\partial}{\partial x}(Au) = \frac{\partial}{\partial \kappa}\left(u\frac{\partial A}{\partial x} + A\frac{\partial u}{\partial x}\right) = \frac{\partial u}{\partial \kappa}\frac{\partial A}{\partial x} + u\frac{\partial}{\partial \kappa}\frac{\partial A}{\partial x} + \frac{\partial A}{\partial \kappa}\frac{\partial u}{\partial x} + A\frac{\partial}{\partial \kappa}\frac{\partial u}{\partial x} = 0, \tag{2.40}$$

$$\frac{\partial}{\partial \kappa}\left(u\frac{\partial u}{\partial x} + \frac{\partial p}{\partial x}\right) = u\frac{\partial}{\partial \kappa}\frac{\partial u}{\partial x} + \frac{\partial u}{\partial \kappa}\frac{\partial u}{\partial x} + \frac{\partial}{\partial \kappa}\frac{\partial p}{\partial x} = 0. \tag{2.41}$$

The derivative of the mass conservation equation seems more complicated than the initial form. Therefore, the initial form is retained. Consequently, the partially differentiated Lagrangian can

be described as:

$$
\begin{aligned}
\frac{\partial \mathscr{L}}{\partial \kappa} = & \int_0^1 [p - p_t] \frac{\partial p}{\partial \kappa} dx + \int_0^1 \lambda_1 \frac{\partial}{\partial \kappa} \frac{\partial}{\partial x} (Au) dx \\
& + \int_0^1 \lambda_2 \left(u \frac{\partial}{\partial \kappa} \frac{\partial u}{\partial x} + \frac{\partial u}{\partial \kappa} \frac{\partial u}{\partial x} + \frac{\partial}{\partial \kappa} \frac{\partial p}{\partial x} \right) dx.
\end{aligned}
\tag{2.42}
$$

Next, integration by parts is applied to solve equation 2.42:

$$
\begin{aligned}
\int_0^1 \lambda_1 \frac{\partial}{\partial \kappa} \frac{\partial Au}{\partial x} dx &= \left[\lambda_1 \frac{\partial Au}{\partial \kappa} \right]_{x=0}^{x=1} - \int_0^1 \frac{\partial Au}{\partial \kappa} \frac{\partial \lambda_1}{\partial x} dx \\
\int_0^1 \lambda_2 u \frac{\partial}{\partial \kappa} \frac{\partial u}{\partial x} dx &= \left[\lambda_2 u \frac{\partial u}{\partial \kappa} \right]_{x=0}^{x=1} - \int_0^1 \frac{\partial u}{\partial \kappa} \frac{\partial \lambda_2 u}{\partial x} dx \\
\int_0^1 \lambda_2 \frac{\partial}{\partial \kappa} \frac{\partial p}{\partial x} dx &= \left[\lambda_2 \frac{\partial p}{\partial \kappa} \right]_{x=0}^{x=1} - \int_0^1 \frac{\partial p}{\partial \kappa} \frac{\partial \lambda_2}{\partial x} dx,
\end{aligned}
\tag{2.43}
$$

Where $\left[\lambda_1 \frac{\partial Au}{\partial \kappa} \right]_{x=0}^{x=1} = \left[\lambda_1 A \frac{\partial u}{\partial \kappa} + \lambda_1 u \frac{\partial A}{\partial \kappa} \right]_{x=0}^{x=1}$ so that equation 2.38 can be simplified to:

$$
\begin{aligned}
\frac{\partial \mathscr{L}}{\partial \kappa} = & \int_0^1 \left[p - p_t - \frac{\partial \lambda_2}{\partial x} \right] \frac{\partial p}{\partial \kappa} dx + \left[\lambda_2 \frac{\partial p}{\partial \kappa} \right]_{x=0}^{x=1} - \int_0^1 \frac{\partial Au}{\partial \kappa} \frac{\partial \lambda_1}{\partial x} dx \\
& + \left[\lambda_1 A \frac{\partial u}{\partial \kappa} + \lambda_1 u \frac{\partial A}{\partial \kappa} \right]_{x=0}^{x=1} + \int_0^1 \lambda_2 \frac{\partial u}{\partial \kappa} \frac{\partial u}{\partial x} dx - \int_0^1 \frac{\partial u}{\partial \kappa} \frac{\partial \lambda_2 u}{\partial x} dx + \left[\lambda_2 u \frac{\partial u}{\partial \kappa} \right]_{x=0}^{x=1} \\
= & \int_0^1 \left[\lambda_2 \frac{\partial u}{\partial x} - \frac{\partial \lambda_2 u}{\partial x} - A \frac{\partial \lambda_1}{\partial x} \right] \frac{\partial u}{\partial \kappa} dx + \int_0^1 \left[p - p_t - \frac{\partial \lambda_2}{\partial x} \right] \frac{\partial p}{\partial \kappa} dx \\
& - \int_0^1 u \frac{\partial \lambda_1}{\partial x} \frac{\partial A}{\partial \kappa} dx + \left[(\lambda_1 A + \lambda_2 u) \frac{\partial u}{\partial \kappa} \right]_{x=0}^{x=1} + \left[\lambda_2 \frac{\partial p}{\partial \kappa} \right]_{x=0}^{x=1} + \left[\lambda_1 u \frac{\partial A}{\partial \kappa} \right]_{x=0}^{x=1}.
\end{aligned}
\tag{2.44}
$$

Adjoint equations

The simplified equations are now observed to identify the adjoint equations. As described in section 2.3.1, the Lagrange multipliers are determined so that $\frac{d\mathscr{L}}{du} = 0$, where u are the physical variables. In this case, the first and the second term of equation 2.44 are differentiating one of the physical variables (the velocity u or the pressure p) with respect to κ. Consequently, these

expressions that are not yet differentiated are the adjoint equations. The physical parameters in the remaining terms are not sensitive to changes of the design variables and are therefore no adjoint equations.

A byproduct of the adjoint method is the similarity between the adjoint equations and the governing equations. This condition can be used to rename the Lagrange multipliers. The adjoint equations of equation 2.44 are like the governing equations when the Lagrange multipliers λ_1 and λ_2 are renamed to the adjoint pressure p_a and the adjoint velocity u_a, respectively:

$$\begin{aligned}\frac{\partial \mathscr{L}}{\partial \kappa} = & \int_0^1 \overbrace{\left[u_a\frac{\partial u}{\partial x} - \frac{\partial u_a u}{\partial x} - A\frac{\partial p_a}{\partial x}\right]}^{\text{similar to the momentum conservation}} \frac{\partial u}{\partial \kappa}dx + \int_0^1 \overbrace{\left[p - p_t - \frac{\partial u_a}{\partial x}\right]}^{\text{similar to } \mathscr{F}} \frac{\partial p}{\partial \kappa}dx \\ & - \int_0^1 u\frac{\partial p_a}{\partial x}\frac{\partial A}{\partial \kappa}dx + \left[(p_a A + u_a u)\frac{\partial u}{\partial \kappa}\right]_{x=0}^{x=1} + \left[u_a\frac{\partial p}{\partial \kappa}\right]_{x=0}^{x=1} + \left[p_a u\frac{\partial A}{\partial \kappa}\right]_{x=0}^{x=1},\end{aligned} \tag{2.45}$$

Where $-\frac{\partial u_a u}{\partial x}$ is called an Adjoint Transpose Convection (ATC) term as it lacks in the momentum conservation equation. It can lead to instability in some cases and therefore prevents the use of the adjoint method for these problems (Karpouzas et al., 2016). The isolated adjoint equations of this problem with the newly introduced adjoint variables are:

$$\begin{aligned} u_a\frac{\partial u}{\partial x} \overbrace{- \frac{\partial u_a u}{\partial x}}^{\text{ATC term}} - A\frac{\partial p_a}{\partial x} &= 0 \\ p - p_t - \frac{\partial u_a}{\partial x} &= 0. \end{aligned} \tag{2.46}$$

Sensitivity variables

As the Lagrange multipliers are chosen so that $\frac{\partial \mathscr{L}}{\partial u} = 0$, only the sensitivity variables remain for the calculation of $\frac{\partial \mathscr{L}}{\partial \kappa} = \frac{\partial \mathscr{F}}{\partial \kappa}$. The sensitivity variables are the terms of equation 2.45 where the area is differentiated with respect to the design variables $\frac{\partial A}{\partial \kappa}$, because A is directly dependent

on κ:

$$\begin{aligned}\frac{\partial \mathscr{L}}{\partial \kappa} = \frac{\partial \mathscr{F}}{\partial \kappa} &= -\int_0^1 u \frac{\partial p_a}{\partial x}\frac{\partial A}{\partial \kappa} dx + \left[p_a u \frac{\partial A}{\partial \kappa} \right]_{x=0}^{x=1} \\ &= -\int_0^1 u \frac{\partial p_a}{\partial x}\frac{\partial A}{\partial \kappa} dx + p_a u \frac{\partial A}{\partial \kappa}\bigg|_{x=1} - p_a u \frac{\partial A}{\partial \kappa}\bigg|_{x=0}.\end{aligned} \tag{2.47}$$

From equation 2.47 the objective functions differentiated with respect to each of the shape parameters are determined:

$$\begin{aligned}\frac{\partial \mathscr{F}}{\partial \kappa_1} &= -\int_0^1 u \frac{\partial p_a}{\partial x}(-x^3 + 3x^2 - 3x + 1) dx - p_a u|_{x=0} \\ \frac{\partial \mathscr{F}}{\partial \kappa_2} &= -\int_0^1 u \frac{\partial p_a}{\partial x}(3x^3 - 6x^2 + 3x) dx \\ \frac{\partial \mathscr{F}}{\partial \kappa_3} &= -\int_0^1 u \frac{\partial p_a}{\partial x}(-3x^3 + 3x^2) dx \\ \frac{\partial \mathscr{F}}{\partial \kappa_4} &= -\int_0^1 u \frac{\partial p_a}{\partial x} x^3 dx + p_a u|_{x=1},\end{aligned} \tag{2.48}$$

Where $A(x=0, \kappa_1, \kappa_2, \kappa_3, \kappa_4) = \kappa_1$ and $A(x=1, \kappa_1, \kappa_2, \kappa_3, \kappa_4) = \kappa_4$. With these sensitivities, the implementation procedure can begin.

3. Implementation

This chapter describes the implementation process of the two dimensional LBM combined with the porosity model by Spaid and Phelan (1997) as well as the implementation of the one dimensional topology optimization using the adjoint method. Python is used as programming language and the complete code can be found in appendix B. The corresponding user manual in appendix A gives an overview of the structure of the attached code beyond the comments contained in the code itself.

3.1 Lattice Boltzmann Method with Porosity Model

The theory of the LBM described in the previous chapter is now implemented in code. The complete code for this implementation is attached as appendix B.1 and excerpts of it are incorporated in the following description. Figure 3.1 visualizes the steps of one cycle of the LB algorithm. The blue boxes represent the consecutive steps of the cycle, which starts with the propagation step and repeats until the simulation converged. The black arrow indicates that the cycle can be exited after convergence is reached. It should be noted that according to the applied BCs, steps can be left out or must be added. Consequently, this schematic is applicable only for a setup with similar BCs.

The objective is to later optimize blood flow. Therefore, the setup of Pingen et al. (2007) is adopted, as it represents a possible problem when choosing the topology of a stent that replaces a blood vessel. The setup is visualized in figure 3.2. The grid size for the example setup is chosen to be equal in x and y direction. In the following, the Python code of the LBM is particularized in the order shown in figure 3.1. Beforehand, some variables and their content

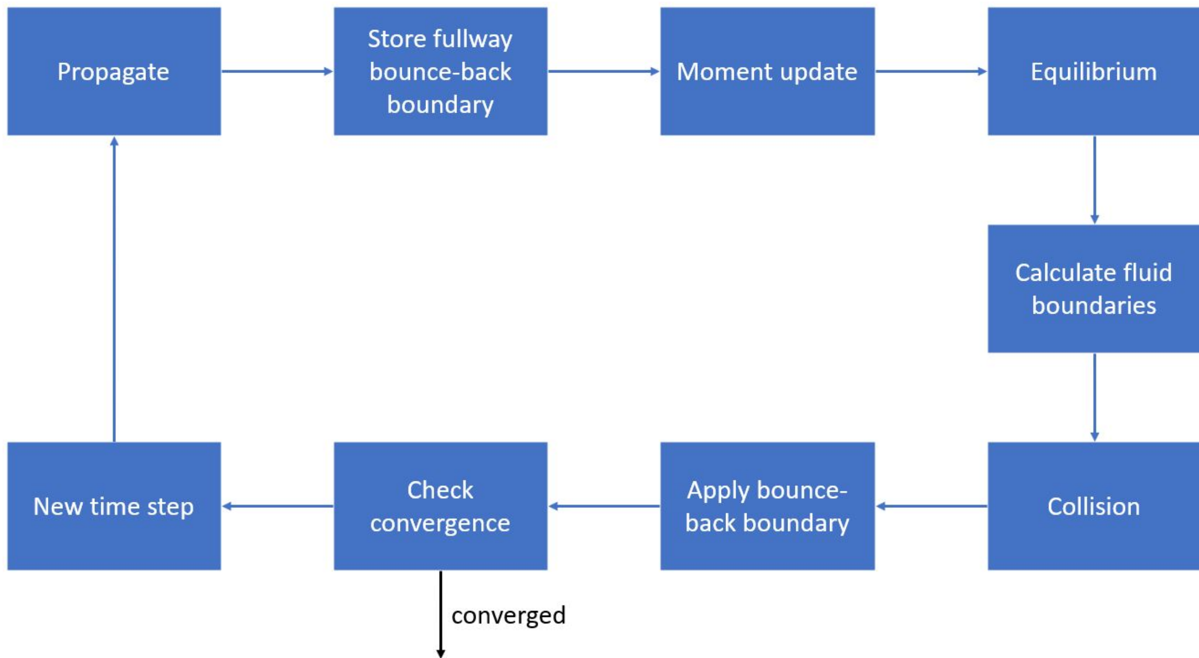


Figure 3.1: Schematic of one cycle of the LB algorithm

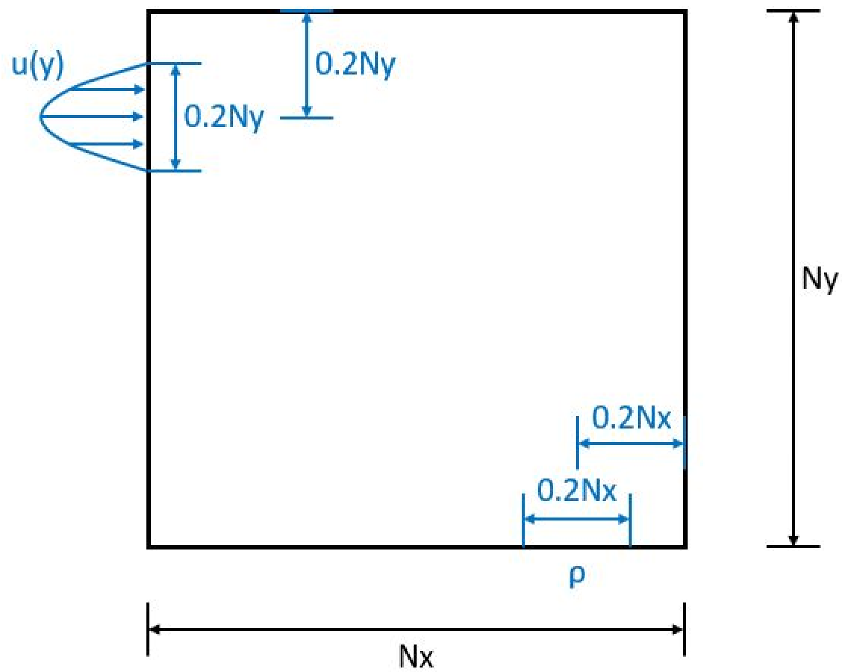


Figure 3.2: Geometry of the computational domain

must be introduced for the reader to understand the remarks in this section. The DVs of the distribution functions f and f^{eq} are stored in matrices of the shape (ny, nx, nv) where ny is the number of nodes along the y-axis, nx is the number of nodes along the x-axis and nv is the index of DVs per node. For each type of BC, one Boolean matrix with the shape (ny, nx) is introduced where *True* indicates that the respective node constitutes such a boundary. Matrices with the same shape are created for the macroscopic properties ρ, u_x and u_y. The remaining variables are either specified in the comments of the respective excerpt, in the user manual or in the code itself.

The cycle begins with the propagation or streaming of the DVs according to equation 2.7. Each velocity travels along the respective vector of the velocity set introduced with figure 2.2, which is incorporated into code as follows:

```
def D2Q9():
    ''' Velocity set D2Q9
    nv:      number of velocities
    idxs:    indices of the velocities
    cxs:     x-coordinates of each velocity vector
    cys:     y-coordinates of each velocity vector
    wghts:   weights of the respective velocity vectors
    '''
    nv    = 9
    idxs  = np.arange(nv)
    cxs   = np.array([ 0, 0, 1, 0,-1, -1, 1, 1,-1])
    cys   = np.array([ 0, 1, 0,-1, 0,  1, 1,-1,-1])
    wghts = np.array([4/9,1/9,1/9,1/9,1/9,1/36,1/36,1/36,1/36])
    return nv, idxs, cxs, cys, wghts
```

Code Listing 3.1: D2Q9 function

During the streaming step the DVs with index 0 remain at the same position, while the ones with indexes 2, 4, 5, 6, 7 and 8 travel along the x-axis and indexes 1, 3, 5, 6, 7 and 8 travel along the y-axis. Even though propagation does not apply to the boundary nodes, this is neglected here and considered at a later point. The incorrectly propagated values will be overwritten in

the steps *Calculate fluid boundaries* and *Apply bounce-back boundary*. This results in a simple propagation function:

```
def propagation(f, idxs, cxs, cys):
    ''' Propagation step of the Lattice Boltzmann Method
    f:       discrete velocities
    idxs:    indices of the velocity set
    cxs:     x-coordinates of the velocity set
    cys:     y-coordinates of the velocity set
    '''
    for i, cx, cy in zip(idxs, cxs, cys): # for each velocity
                                               discretization direction
        f[:,:,i] = np.roll(f[:,:,i], cx, axis=1) # stream along x axis
        f[:,:,i] = np.roll(f[:,:,i], cy, axis=0) # stream along y axis
    return f
```

Code Listing 3.2: Propagation function

The fullway bounce-back BC reflects the DVs it receives, so that they are propagated back to the neighboring fluid cell they came from at the next time step:

```
        # Store bounce-back values
        if bndry.any:
            bb = f[bndry]
            bb  = bb[:,[0,3,4,1,2,7,8,5,6]]
```

Code Listing 3.3: Store bounce-back values

These DVs are not subject to collision. Therefore, they must be stored before the collision step, so that they can be reflected to the sender unchanged. In the presented implementation, the DVs of the solid nodes are reflected and stored right after propagation. It should be noted that this step can be conducted at any time between the propagation and the collision step. When using Python and numpy, it is necessary to either copy the respective values as follows `bb = np.copy(f[bndry])` or to reflect the DVs before f is modified. Otherwise, `bb = f[bndry]` links *bb* to f and *bb* is modified likewise.

Now, the macroscopic parameters of each node are updated. This is done by calculating the density and the velocity of all nodes according to equation 2.8 and 2.9, respectively. Afterwards, the properties at the boundary nodes are overwritten with the correct ones. The applied equations are 2.8, 2.9, 2.14 and 2.20:

```
def macroscopicParam(f, vbndry, pbndry, bndryCond, cxs,cys):
    ''' Calculate the macroscopic parameters of the
    f:          discrete velocities
    vbndry:     boolean matrix that is True for the velocity boundary
    pbndry:     boolean matrix that is True for the pressure boundary
    bndryCond:  list of variables of the boundary conditions
    cxs:        x-coordinates of the velocity set
    cys:        y-coordinates of the velocity set
    rho:        densities
    ux:         velocities in x-direction
    uy:         velocities in y-direction
    '''
    uxIn = bndryCond[0]
    uyIn = bndryCond[1]
    rhoOut = bndryCond[2]

    # Calculate macroscopic parameters
    rho = np.sum(f,2)
    ux  = np.sum(f*cxs,2)/rho
    uy  = np.sum(f*cys,2)/rho

    # At the boundary (velocity, inlet)
    if vbndry.any:
        ux[vbndry] = uxIn
        uy[vbndry] = uyIn
        rho[vbndry] = 1/(1-ux[vbndry]) * (f[vbndry,0]+f[vbndry,1]+f[
                                      vbndry,3]+2*(f[vbndry,4]+f[vbndry
                                      ,5]+f[vbndry,8]))

```

```
28     # At the boundary (pressure, outlet)
29     if pbndry.any:
30         rho[pbndry] = rhoOut
31         ux[pbndry] = 0
32         uy[pbndry] = 1-((f[pbndry,0] + f[pbndry,2] + f[pbndry,4] + 2*(f
                                        [pbndry,3] + f[pbndry,7] + f[pbndry
                                        ,8])) /rho[pbndry]
33
34     return rho, ux, uy
```

Code Listing 3.4: Macroscopic parameter function

With the updated macroscopic parameters, the equilibrium distributions can be determined as described in equation 2.4:

```
1  def equiPoro(feq, rho, ux, uy, idxs, cxs, cys, wghts, poro):
2      ''' Calculate the equilibrium distribution
3      feq:    equilibrium distribution
4      rho:    density
5      ux:     velocities in x-direction
6      uy:     velocities in y-direction
7      idxs, cxs, cys, wghts: variables of the D2Q9 velocity set
8      poro:   porosity measure; 1: fluid; 0: porous
9      '''
10     for i, cx, cy, w in zip(idxs, cxs, cys, wghts):
11         feq[:,:,i] = rho * w * ( 1 + 3*(cx*ux*poro+cy*uy*poro) + 9/2*(
                                        cx*ux*poro+cy*uy*poro)**2 - 3/2*((
                                        ux*poro)**2+(uy*poro)**2) )
12     return feq
```

Code Listing 3.5: Equilibrium distribution function

Eventually, the DVs at the fluid boundaries are updated. As the equilibrium distribution and the macroscopic parameters must be up to date for this, the fluid boundaries are calculated right before collision. The applied equations are 2.16 and 2.21:

```python
def fcorrection(f, feq, rho, ux, uy, vbndry, pbndry):
    ''' Correct the discrete velocities at the open boundaries
    f:          discrete velocities
    feq:    equilibrium distribution
    rho:        densities
    ux:         velocities in x-direction
    uy:         velocities in y-direction
    vbndry:     boolean matrix that is True for the velocity boundary
    pbndry:     boolean matrix that is True for the pressure boundary
    bndryCond:  list of variables of the boundary conditions
    '''
    # At the boundary (velocity, inlet)
    if vbndry.any:
        #f[vbndry] = np.copy(feq[vbndry])

        f[vbndry,2]=feq[vbndry,2]+f[vbndry,4]-feq[vbndry,4]
        f[vbndry,6]=0.5*(rho[vbndry]*(ux[vbndry]+uy[vbndry])-f[vbndry
                                        ,1]-f[vbndry,2]+f[vbndry,3]+2*f[
                                        vbndry,8]+f[vbndry,4])
        f[vbndry,7]=0.5*(rho[vbndry]*(ux[vbndry]-uy[vbndry])+f[vbndry
                                        ,1]-f[vbndry,2]-f[vbndry,3]+f[
                                        vbndry,4]+2*f[vbndry,5])

    # At the boundary (pressure, outlet)
    if pbndry.any:
        f[pbndry,1] = feq[pbndry,1] + (f[pbndry,3] - feq[pbndry,3])
        f[pbndry,5] = 1/2 * (rho[pbndry]*(uy[pbndry]-ux[pbndry]) - f[
                                        pbndry,1] + f[pbndry,2] + f[pbndry
                                        ,3] - f[pbndry,4] + 2*f[pbndry,7])
        f[pbndry,6] = 1/2 * (rho[pbndry]*(ux[pbndry]+uy[pbndry]) - f[
                                        pbndry,1] - f[pbndry,2] + f[pbndry
                                        ,3] + f[pbndry,4] + 2*f[pbndry,8])

    return f
```

Code Listing 3.6: Open boundaries function

After all corrections of the propagated values at the boundaries have been conducted, the collision step takes place according to equation 2.6:

```
def collision(f, feq, tau, dt, nv):
    ''' Collision step for the Lattice Boltzmann Method
    f:       discrete velocities
    feq:     equilibrium distribution
    tau:     relaxation time
    dt:      time step
    nv:      number of discrete velocities
    '''
    for i in range(nv): # Apply collision for all the nodes
        f[:,:,i] -= (dt/tau) * (f[:,:,i]-feq[:,:,i])
    return f
```

Code Listing 3.7: Collision function

Collision applies to all nodes except the solid nodes. Their DVs have been stored in a separate matrix in the step *Store bounce-back boundary*. These will be applied next, where they replace the incorrect results of the collision. The step of overwriting the incorrect DVs must be conducted between collision and propagation:

```
        # Collision for fullway bounce-back boundary
        if bndry.any:
            f[bndry] = bb
```

Code Listing 3.8: Bounce-back collision

Before the next iteration is calculated, the convergence criterion must be checked. If the criterion is satisfied, the simulation stops. In the following, two different convergence criteria are implemented:

```
def convergenceVelo(ux,uy,umaxOld):
    # Check convergence with the max velocity
```

```
    umaxNew = np.max(np.sqrt(ux**2+uy**2)) # save maximum velocity
    criterion = np.abs((umaxOld-umaxNew)/umaxNew)
    umaxOld = umaxNew
    return criterion, umaxOld

def convergenceF(f, fOld, ny, nx, nv):
    f = np.array(f)
    criterion = abs(f - fOld).reshape(ny*nx*nv)
    fOld = np.copy(f)
    return max(criterion), fOld
```

Code Listing 3.9: Convergence criterion

The first one accounts only for the difference between the maximum velocity in the entire system and has been used for testing. However, the more expressive criterion describes the difference between two consecutive flow steps as follows (Pingen et al., 2007):

$$||f_t - f_{t-1}|| \leq \varepsilon. \tag{3.1}$$

After all these steps are conducted, one cycle is completed, and the time is increased by one time step.

3.2 Topology Optimization

The theory of the adjoint method described in the previous chapter is now implemented in code. Therefore, figure 3.3 visualizes the steps of one cycle of the adjoint algorithm. The blue boxes again represent the consecutive steps of the cycle, and the black arrow indicates that the cycle can be exited after convergence has been reached.

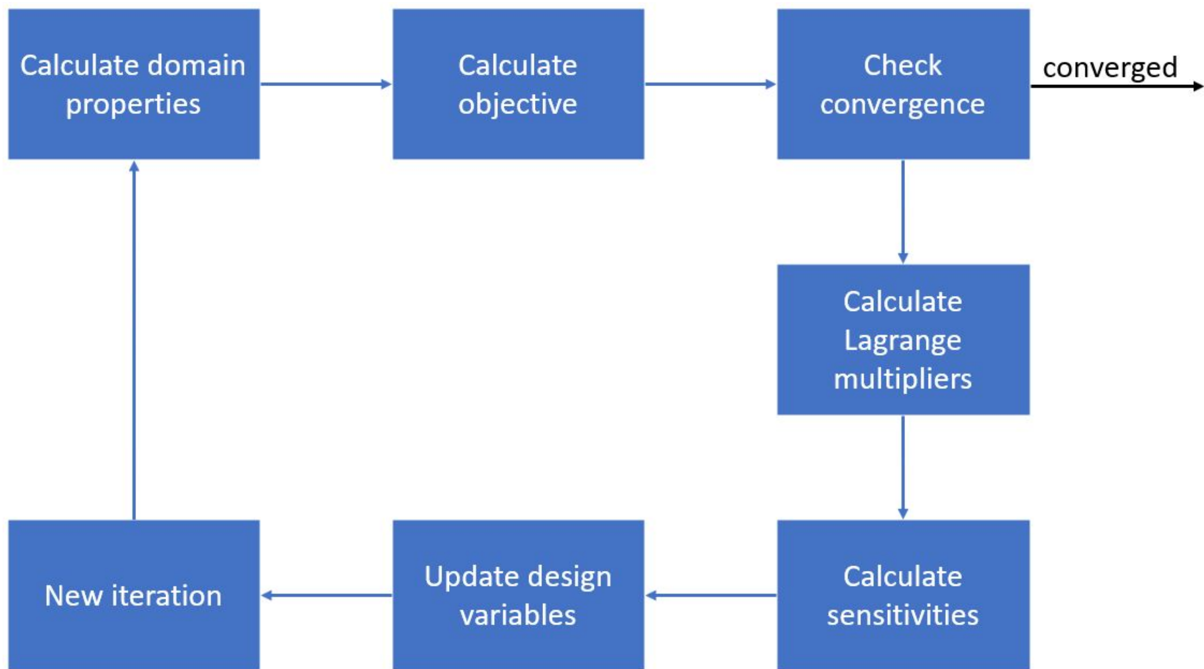


Figure 3.3: Schematic of one cycle of the adjoint method

The method begins with the calculation of the domain properties. This step equals the CFD simulation whenever the adjoint method is combined with a CFD problem. With the properties of the system, the objective can be computed, and the convergence is checked using the convergence criterion. As soon as this criterion is satisfied, the algorithm stops, and the results can be visualized as desired. When the result of the objective function is still too large, the optimization begins by calculating the Lagrange multipliers λ. Afterwards, the sensitivities of the objective function with respect to the design variables are determined, so that the design variables can be updated. Various algorithms are applicable here as for example the steepest descent method and the Newton method (Farahbakhsh, 2020). Once the design variables have been updated, the cycle repeats until convergence is reached.

3.2.1 Example

The procedure is now shown based on the example introduced in chapter 2.3.2. The complete code for this implementation is attached as appendix B.2 and excerpts of it are incorporated in the following description.

The 1D problem reaches from $x = 0$ to $x = 1$ with N nodes and the distance between nodes $\Delta x = \frac{1}{N}$. The pressure at the end of the tube is specified as $p_0 = 0$ and the velocity at the entrance of the tube is defined as $u_0 = 1$. With this information and the initial guess for the design variables κ, the adjoint method cycle can be entered. Additional parameters are needed for some steps and will be introduced at the corresponding section to preserve the context.

Calculate Domain Properties

The first step is to calculate the domain properties which are the area A, the velocity u, and the pressure p of the system. The area is calculated by equation 2.35 and the velocity can be obtained by the continuity equation. As the velocity u_0 at the entrance of the tube is defined as BC of the system, $x = 0$ is used as reference for the velocity calculation of all the remaining grid points:

$$\begin{aligned} A_0 u_0 &= A_k u_k \qquad \text{for} \quad k \in [1, N[\\ u_k &= \frac{A_0}{A_k} u_0. \end{aligned} \tag{3.2}$$

The pressure is determined with the momentum conservation equation 2.34 which can be transformed to:

$$\begin{aligned} \frac{\partial p}{\partial x} &= -u \frac{\partial u}{\partial x} \\ \int_k^{k+1} \frac{\partial p}{\partial x} dx &= -\int_k^{k+1} u \frac{\partial u}{\partial x} dx \qquad \text{for} \quad k \in [0, N-2] \\ [p]_k^{k+1} &= -u_k \frac{\partial u_k}{\partial x} \Delta x \\ p_{k+1} - p_k &= -u_k \frac{\partial u_k}{\partial x} \Delta x \\ p_k &= p_{k+1} + u_k \frac{\partial u_k}{\partial x} \Delta x. \end{aligned} \tag{3.3}$$

To obtain the pressure the equation is integrated. The velocity is dependent on x and must be

integrated individually. However, for very small distances Δx between the grid points, it can be assumed that $\int_k^{k+1} u\frac{\partial u}{\partial x}dx = u_k\frac{\partial u_k}{\partial x}\Delta x$. With this simplification, the pressure can be calculated by equation 3.3. The pressure $p(x=1) = p_0$ at the end of the tube is defined as BC of the system. Consequently, $x = 1$ is used as reference for the pressure calculation, so that p_{N-1} is determined by the aid of p_N and so on. The calculations are incorporated in the code as follows:

```
def properties(a,u0,p0,dx,n):
    ''' Calculate the domain properties
    a:  area of the tube
    u0: velocity at the boundary/entrance of the tube
    p0: pressure at the boundary/exit of the tube
    dx: distance between the grid points
    n:  number of grid points
    u:  velocity along the tube
    p:  pressure along the tube
    '''
    # Calculate velocity; Continuity equation: A1*u1=A2*u2
    c = u0*a[0]
    u = c/a
    # Calculate pressure; Momentum conservation
    dudx = gradient(u,dx)
    p = np.zeros(n)
    p[-1] = p0
    for i in range(n-2,-1,-1):
        p[i] = p[i+1] + u[i]*dudx[i]*dx
    return u,p
```

Code Listing 3.10: Domain properties

Objective and Convergence

Now that the pressure is known, the objective in equation 2.36 can be calculated. Again, the integral can be approximated for small distances Δx between the grid points:

$$\mathscr{F} = \frac{1}{2}\int_0^1 [p - p_t]^2 \, dx \simeq \frac{1}{2} N \Delta x \sum_{k=0}^{N-1} [p_k - p_{tk}]^2 = \frac{1}{2} \sum_{k=0}^{N-1} [p_k - p_{tk}]^2 . \tag{3.4}$$

The aim is to minimize the value of the objective function $\mathscr{F} \to 0$. When the objective is smaller than the convergence criterion, the solution has converged, and the algorithm stops. In the following code, the target pressure `pt` is calculated by the function `pTarget(x)` and the convergence criterion `conv` is defined at an earlier part of the code:

```
# Calculate objective
pt = pTarget(x)
dp = p-pt
F = 0.5*np.sum(dp**2)

# Check for convergence
if F < conv:
    break
```

Code Listing 3.11: Objective and convergence

Calculate Lagrange Multipliers

Next, the Lagrange multipliers that were renamed to adjoint variables are calculated. This is done by the aid of the adjoint equations 2.46 derived in the theory chapter:

$$
\begin{aligned}
\frac{\partial u_a}{\partial x} &= p - p_t \\
\int_{k-1}^{k} \frac{\partial u_a}{\partial x} dx &= \int_{k-1}^{k} p - p_t dx \qquad \text{for} \quad k \in [1, N[\\
[u_a]_{k-1}^{k} &= (p_k - p_{tk})\Delta x \\
u_{ak} - u_{ak-1} &= (p_k - p_{tk})\Delta x \\
u_{ak} &= u_{ak-1} + (p_k - p_{tk})\Delta x.
\end{aligned} \tag{3.5}
$$

The expression is integrated to obtain the adjoint velocity. With the approximation $\int_{k-1}^{k} p - p_t dx = (p_k - p_{tk})\Delta x$, the adjoint velocity can be calculated by equation 3.5. The adjoint pressure is calculated with the remaining adjoint equation by the same concept:

$$
\begin{aligned}
u_a \frac{\partial u}{\partial x} - \frac{\partial u_a u}{\partial x} - A \frac{\partial p_a}{\partial x} &= 0 \\
\frac{\partial p_a}{\partial x} &= \frac{u_a \frac{\partial u}{\partial x} - \frac{\partial u_a u}{\partial x}}{A} \\
\frac{\partial p_a}{\partial x} &= \frac{u_a \frac{\partial u}{\partial x}}{A} - \frac{u \frac{\partial u_a}{\partial x}}{A} - \frac{u_a \frac{\partial u}{\partial x}}{A} \\
\frac{\partial p_a}{\partial x} &= -\frac{u}{A} \frac{\partial u_a}{\partial x} \\
[p_a]_{k}^{k+1} &= -\int_{k}^{k+1} \frac{u}{A} \frac{\partial u_a}{\partial x} dx \qquad \text{for} \quad k \in [0, N-2] \\
p_{ak} &= p_{ak+1} + \frac{u_k}{A_k} \frac{\partial u_{ak}}{\partial x} dx.
\end{aligned} \tag{3.6}
$$

As with the domain properties u and p, the adjoint velocity and adjoint pressure at the boundaries are utilized to calculate the adjoint variables along the whole tube. $u_a(x=0)$ and $p_a(x=1)$ are obtained by the aid of the solved integrals in equation 2.45 derived in the theory chapter and the BCs $u_0 = 1$ and $p_0 = 0$. The BCs at the inlet and outlet are constant and independent of

the design variables κ_n. Consequently, the derivative of the velocity with respect to the design variables at $x = 0$ is zero $\frac{\partial u}{\partial \kappa_n}|_{x=0} = 0$ and the derivative of the pressure with respect to the design variables at $x = 1$ is also zero $\frac{\partial p}{\partial \kappa_n}|_{x=1} = 0$. These expressions are used to calculate the adjoint velocity and the adjoint pressure, respectively:

$$\begin{aligned}
\left[u_a \frac{\partial p}{\partial \kappa_n}\right]_{x=0}^{x=1} &= 0 \\
u_a \frac{\partial p}{\partial \kappa_n}\bigg|_{x=0} - 0 &= 0 \\
\frac{\partial p}{\partial \kappa_n}\bigg|_{x=0} \neq 0 \qquad u_a(x=0) &= 0,
\end{aligned} \tag{3.7}$$

$$\begin{aligned}
\left[(p_a A + u_a u)\frac{\partial u}{\partial \kappa_n}\right]_{x=0}^{x=1} &= 0 \\
0 - (p_a A + u_a u)\frac{\partial u}{\partial \kappa_n}\bigg|_{x=1} &= 0 \\
\frac{\partial u}{\partial \kappa_n}\bigg|_{x=1} \neq 0 \qquad (p_a A + u_a u)|_{x=1} &= 0 \\
p_a(x=1) &= -\frac{u_a u}{A}.
\end{aligned} \tag{3.8}$$

As $\frac{\partial p}{\partial \kappa_n}|_{x=0}$ and $\frac{\partial u}{\partial \kappa_n}|_{x=1}$ are unknown, the adjoint velocity at $x = 0$ must be zero and $p_a A + u_a u$ at $x = 1$ must be zero to fulfill the requirement of the whole term being zero, respectively. After these derivations have been completed, the equations can be incorporated as a function as follows:

```
1 def lm(a,u,dp,dx,n):
2     ''' Calculate Lagrange multipliers
3     a:  area of the tube
4     u:  velocity along the tube
5     dp: difference between the actual pressure and the target pressure
                                          along the tube
6     dx: distance between the grid points
7     n:  number of grid points
```

```
    ua: adjoint velocity along the tube
    pa: adjoint pressure along the tube
    '''
    # Calculate adjoint velocity by adjoint equation
    ua = np.zeros(n)
    for i in range(1,n):
        ua[i] = ua[i-1]+dp[i]*dx
    # Calculate adjoint pressure by adjoint equation
    duadx = gradient(ua, dx)
    pa = np.zeros(n)
    pa[-1] = - (u[-1]*ua[-1])/a[-1]
    for i in range(n-2,-1,-1):
        pa[i] = pa[i+1] + u[i]/a[i] * duadx[i] * dx
    return ua,pa
```

Code Listing 3.12: Lagrange multipliers

Sensitivities and Design Variable Update

After the adjoint variables have been computed, the sensitivities for the design variables can be calculated by equations 2.48. They are calculated in Python as follows, where `dFdkappa` is a list comprising the sensitivities of each design variable:

```
def sensitivities(x,u,pa,dx,kappa):
    ''' Calculate the sensitivities '''
    dpadx = gradient(pa,dx)
    # Solve sensitivity equations
    dFdkappa = np.zeros(len(kappa))
    dFdkappa[0] = - np.trapz(u*dpadx*(-x**3+3*x**2-3*x+1),x,dx) - pa
                                    [0]*u[0]
    dFdkappa[1] = - np.trapz(u*dpadx*(3*x**3-6*x**2+3*x),x,dx)
    dFdkappa[2] = - np.trapz(u*dpadx*(-3*x**3+3*x**2),x,dx)
    dFdkappa[3] = - np.trapz(u*dpadx*x**3,x,dx) + pa[-1]*u[-1]
    return dFdkappa
```

Code Listing 3.13: Sensitivities

Once the sensitivities have been determined, the current design variables are updated with the steepest descent method, which can be described as an iterative procedure of modifying the design variables. Steps in the direction of the sensitivities $\partial\mathscr{F}\partial\kappa_{it}$ are taken from the design variables κ_{it} of the iteration it. The magnitude of each step is defined by the sensitivities itself and the step length η:

$$\kappa_{it+1} = \kappa_{it} - \eta\frac{\partial\mathscr{F}}{\partial\kappa_{it}}, \tag{3.9}$$

Which is implemented in code as follows:

```
1  # Update the design variables
2  kappa -= eta*dFdkappa
```

Code Listing 3.14: Design variable update

The procedure is repeated with the resulting design variables κ_{it+1}.

3.2.2 Combination with the Lattice Boltzmann Method

When combining the LBM with the adjoint method, the procedure shown in figure 3.3 can be followed likewise. The properties of the computational domain and the BCs are defined in the setup of the LB simulation and the initial guess of the design variables is used to conduct a simulation, which represents the step *Calculate domain properties* in the figure. After the simulation is converged, the objective can be calculated. In the topology optimization problem of chapter 3.1, a possible objective function determines the pressure drop that should be minimized:

$$p_{drop} = \sum_{inlet} p_{in} - \sum_{outlet} p_{out}, \tag{3.10}$$

Which can be used for flows with a small Mach number and a negligible elevation change (Pingen et al., 2007). Using equation 2.10 this can be written as:

$$p_{drop} = c_s^2 \left[\sum_{inlet} \rho_{in} - \sum_{outlet} \rho_{out} \right]. \tag{3.11}$$

Also, the drag or the flow rate can be used as an objective function subject to minimization as described by Pingen et al. (2007) and Pingen et al. (2009). Besides, it is possible to incorporate multiple objective functions when using the adjoint method. Next to the mentioned objective functions, Pingen et al. (2007) uses a volume constraint describing the maximum fraction of nodes that are allowed to be fluid.

4. Results

Now that the implementation of the methods is complete, the simulations can be run.
Even though the LBM and the adjoint method have not been combined, several important aspects have been discovered that should be considered when using the LBM for topology optimization.

4.1 Lattice Boltzmann Method

First, the LBM is discussed. Before the simulation results are presented, the computational setup is specified including important variables, the grid size and the BCs.

4.1.1 Computational Setup

Until now, the implementation of the general method including the BCs has been described and the test case has been visualized in figure 3.2. Some missing parameters are described here.
The computational domain is composed of 27 nodes in x- and y-direction whereof the first and last ones represent solid boundaries or open boundaries. This results in $25 \times 25 = 625$ nodes that are subject to topology optimization plus the nodes of the open boundaries. Depending on the required precision the number of nodes must be increased. As a high number of nodes requires considerable computational resources, a small number of nodes is chosen for this illustration. The grid size can be adjusted as described in the manual in appendix A.
The velocity boundary is defined by the maximum velocity at the boundary. The velocities at the solid boundaries are supposed to be zero, so that the parabolic velocity profile at the inlet can be determined as follows:

```python
def parabolVelo(Y, uxInMax, vbndry):
    ''' Calculate inlet velocities with parabolic profile
    ny:          number of nodes in y-direction
    Y:           matrix with y-coordinates of the nodes
    uxInMax:     maximum velocity in x-direction at the inlet
    uxLen:       Size of the inlet
    uxIn:        velocities in x-direction at the inlet
    '''
    inletLen = np.count_nonzero(vbndry)
    uxLen = inletLen+2
    a = -4*uxInMax/(uxLen-1)**2
    uxIn = a*Y[:uxLen,0]**2 - a*Y[:uxLen,0]*(uxLen-1)
    return uxIn[1:-1]
```

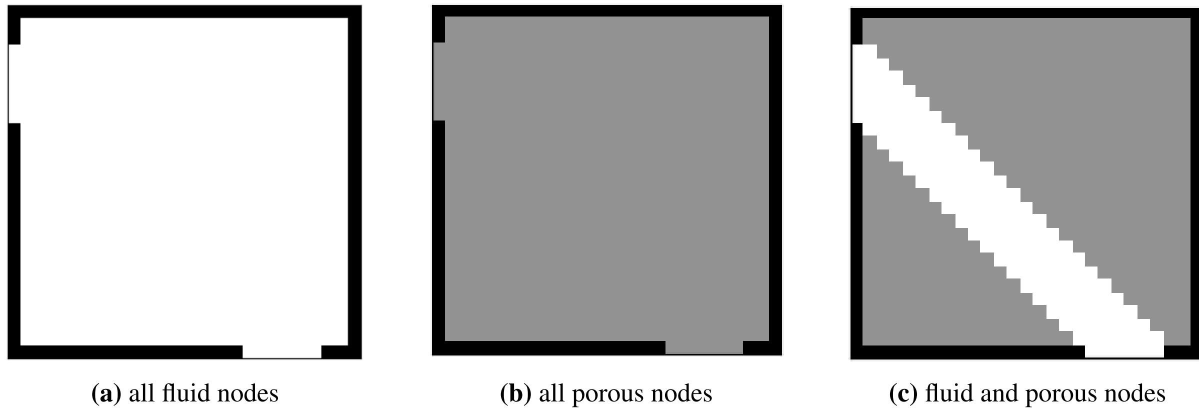
Code Listing 4.1: Parabolic inlet velocity profile

The pressure at the outlet is constant. All the nodes at the open pressure boundary show the same pressure. The maximum velocity of the parabolic velocity profile at the inlet is 0.025 and the pressure at the outlet is 0.33. All values are in lattice units.

The porosity model provides the design variables β for the topology optimization. To run a simulation, a β matrix must be chosen. When running the simulation in combination with topology optimization, this matrix is provided by the adjoint algorithm except for the initial guess. In the following, the LB simulation is conducted for three different β matrices.

As explained in chapter 2.2, the expression $\beta = \frac{\nu}{k_{tow}}$ must be fulfilled to recover the Brinkman equation. This is realized by fixing the viscosity for the whole computational domain. By determining β, the permeability varies likewise. As the permeability is not required for further calculations for the LBM, there is no need to calculate it. When the maximum value is chosen for the porosity measure which translates to a solid, the permeability adapts, meaning that no fluid can pass through the permeable material anymore.

On the contrary, fixing the permeability is not viable when determining β with the adjoint method. If doing so, the viscosity varies with the porosity measure. As the relaxation time



(a) all fluid nodes **(b)** all porous nodes **(c)** fluid and porous nodes

Figure 4.1: Porosities of the simulations

τ is dependent on the viscosity and the relaxation time determines the maximum value of the porosity measure, the algorithm cannot be given a specific range in which the porosity measure has to be optimized. Consequently, keeping the viscosity and the relaxation time constant throughout the computational domain and the iterations of the adjoint method is recommended for the optimization algorithm to work smoothly.

In the following simulations the viscosity is chosen to be $\nu = 0.4$ which leads to a relaxation time of $\tau = 1.7$. Different guesses are chosen for the porosity measure β as visualized in figure 4.1. It shows the computational domain with the solid boundary in black. A light color indicates a highly fluid node while a dark color indicates a solid or porous medium. Depending on the β value at the respective node, the cell will show a white, grey, or black color. The shade of grey translates to the level of porosity.

4.1.2 All Fluid Nodes

First, the test case is simulated with all fluid nodes. Figure 4.1a presents the corresponding visualization with cells that are all white except the solid boundary. This translates to a β matrix with all zero elements as 0 translates to a fluid and $\frac{1}{\tau} \approx 0.588$ translates to a porous node, more specifically a solid.

The simulation converges after 2258 iterations. Figure 4.3 presents three velocity profiles at distinct locations of the computational domain after convergence. These locations are visual-

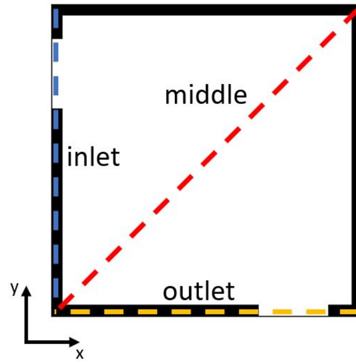


Figure 4.2: Illustration of locations

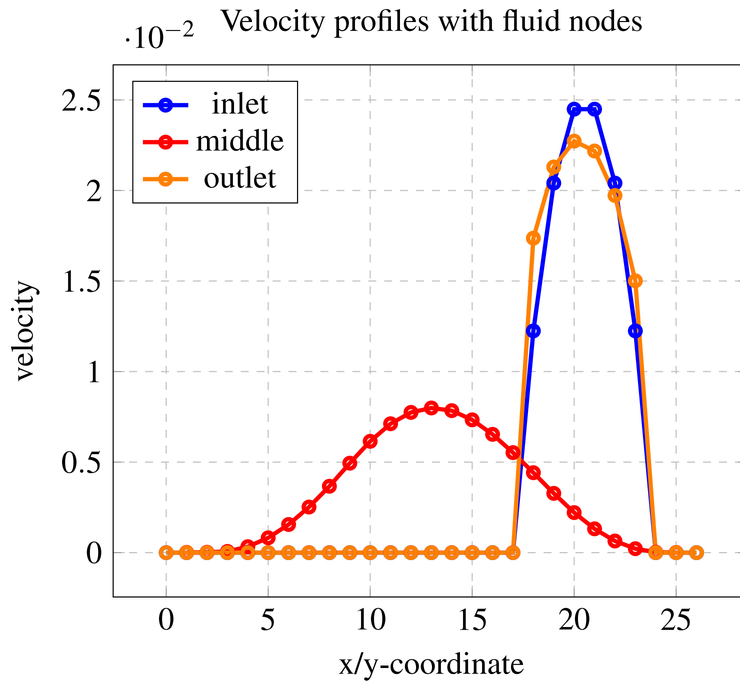


Figure 4.3: Velocity profiles with fluid nodes

ized in figure 4.2 which depicts the computational domain of figure 3.2 with the dashed colored lines in blue, red and orange for the inlet, middle and outlet velocity profile, respectively. The blue curve shows the profile at the inlet. For this profile, the velocities at $x = 0$ along the y-axis are visualized. This means, that the x-axis of figure 4.3 refers to the y-coordinates in the computational domain. The orange curve represents the outlet velocity profile which refers to the velocities at $y = 0$ along the x-axis as visualized by the dashed orange line in figure 4.2. The red curve shows the velocity profile of the diagonal or the middle between the inlet and outlet where $x = y$.

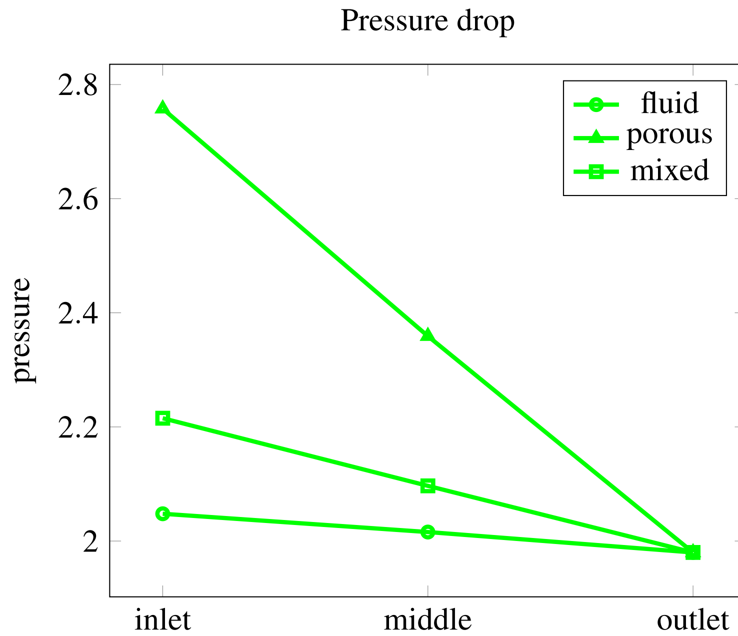
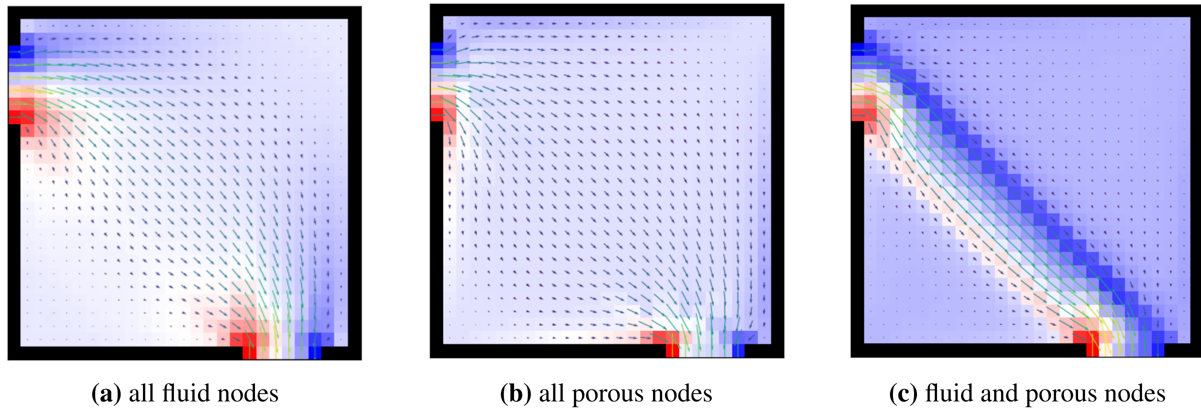


Figure 4.4: Pressure drop

The velocity curve at the inlet shows the parabolic profile that was created based on the maximum inlet velocity that is predefined before the simulation. The outlet velocity profile is similar to the one of the inlet although the parabolic shape is slightly deformed. As the profile in the middle is not constraint by solid nodes, the curve stretches from the left to the right and shows a lower maximum. Figure 4.4 shows the pressure at the same locations. It can be observed that the pressure drops continuously starting at the inlet passing the diagonal and ending at the outlet. The pressure drop can be used as an objective function as described in equation 3.11. For this simulation, the pressure drop is approx. 0.0678.

Figure 4.5 shows the representations of the computational domain with the different porosity setups. Figure 4.5a belongs to the current setup with all fluid nodes. The solid boundaries are presented in black; the white areas are the nodes that are subject to topology optimization and the open boundaries. The velocities are indicated by small arrows that point in the direction of the macroscopic velocity of the respective node. The length of the arrow implies the magnitude. Additionally, the vorticity is indicated by a red or blue background of the grid. Here the magnitude is visualized by the intensity of the color.



(a) all fluid nodes **(b)** all porous nodes **(c)** fluid and porous nodes

Figure 4.5: Velocity and vorticity of the simulations

4.1.3 All Porous Nodes

Second, the test case is simulated with all porous nodes, which is visualized in figure 4.1b. All elements of β are set to 0.45 which lays inside the porosity range $0 < \beta < 0.588$. This setup takes much longer to converge with 29783 time steps. Due to the porous medium the flow travels slower so that more time is required to reach convergence.

In figure 4.6, the results of this simulation can be observed. The figure has the same setup as the figure 4.3 with the velocity profiles at three locations across the domain. When viewing the velocity profiles, there are conspicuous changes. The velocity profile at the inlet can be used as a reference as the same BCs are used and the profile matches the one of the simulation with all fluid nodes. However, the outlet profile does not show a parabolic profile, but a ramp with the maximum at the left-hand side of the outlet and a continuous decrease when moving to the right. Additionally, the velocity profile in the middle of the tube shows a lower amplitude while the curve is flatter so that the velocities at the corners are faster. This can also be observed in figure 4.5b where the length of the arrows indicating the velocity magnitude show less variation in comparison with figure 4.5a. The pressure drop is now greater than with all fluid nodes as it can be examined in figure 4.4. The pressure drop that is calculated by equation 3.11 equals 0.7777 compared to the previously achieved 0.0678. According to the objective function, the β matrix with all fluid nodes performs better than the one with all porous nodes.

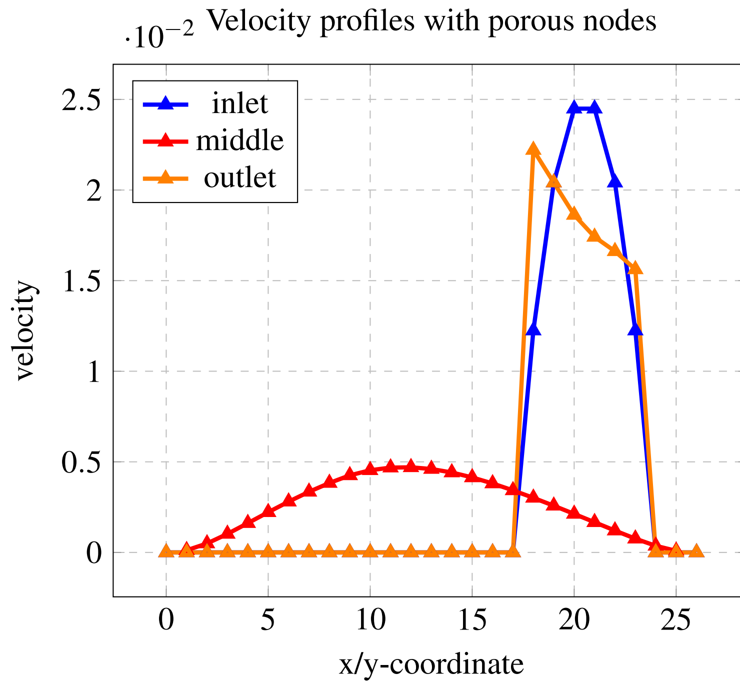


Figure 4.6: Velocity profiles with porous nodes

4.1.4 Fluid and Porous Nodes

Finally, the test case is simulated with fluid and porous nodes, which is visualized in figure 4.1c. Most elements of β are set to 0.45 as in the example with all porous nodes, while the elements drawing a diagonal between the open boundaries are set to 0. With 8517 time steps this setup is faster to converge than the previous setup with all porous nodes but takes nearly four times as long as the setup with all fluid nodes.

In figure 4.7, the results of this simulation can be observed. For a better comparison with the previous plots, the velocity profiles in the middle and at the outlet are summarized in figure 4.8 and 4.9 respectively. In figure 4.9 the flow profile of the fluid is close to the profile on the inlet. The flow profile of the mixed porosity reaches the same magnitude, but the shape is no longer parabolic. The profile with the fully porous medium has the same shape as the mixed medium, but with a lower magnitude. In figure 4.8 the porous medium also results in the lowest flow magnitude. Another conspicuity is the location of the peak of the profiles. While the peak of the fluid medium matches the center of the domain, the other simulations result in a peak that is slightly moved to the left. This can be explained by the location of the inlet and outlet.

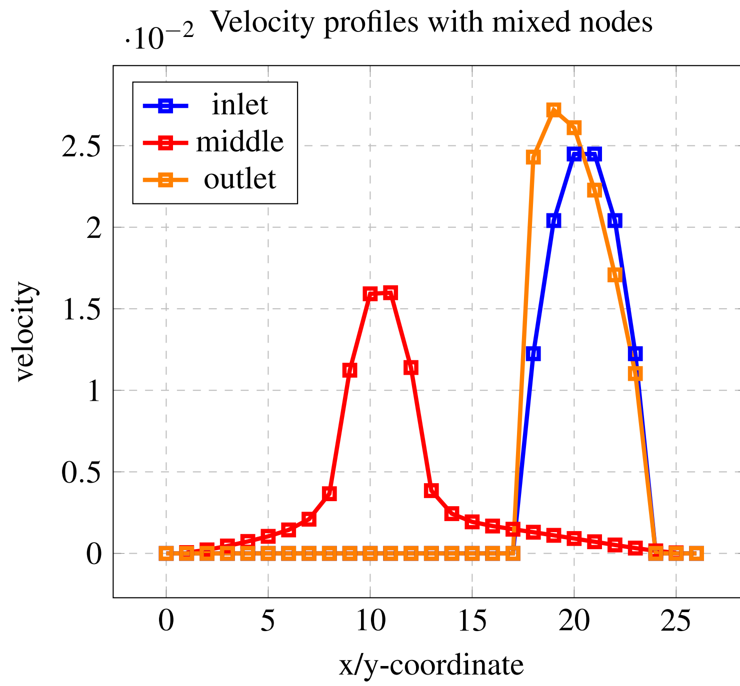


Figure 4.7: Velocity profiles with fluid and porous nodes

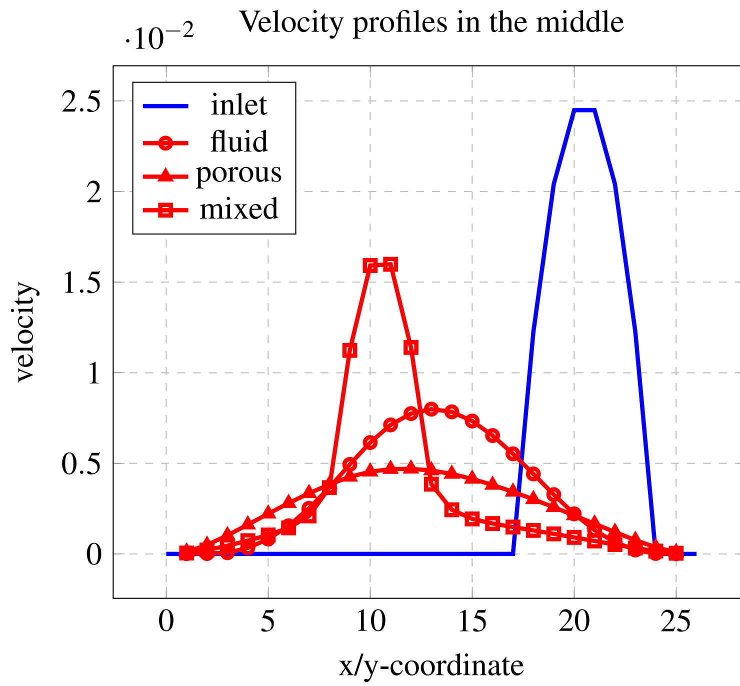


Figure 4.8: Velocity profiles in the middle

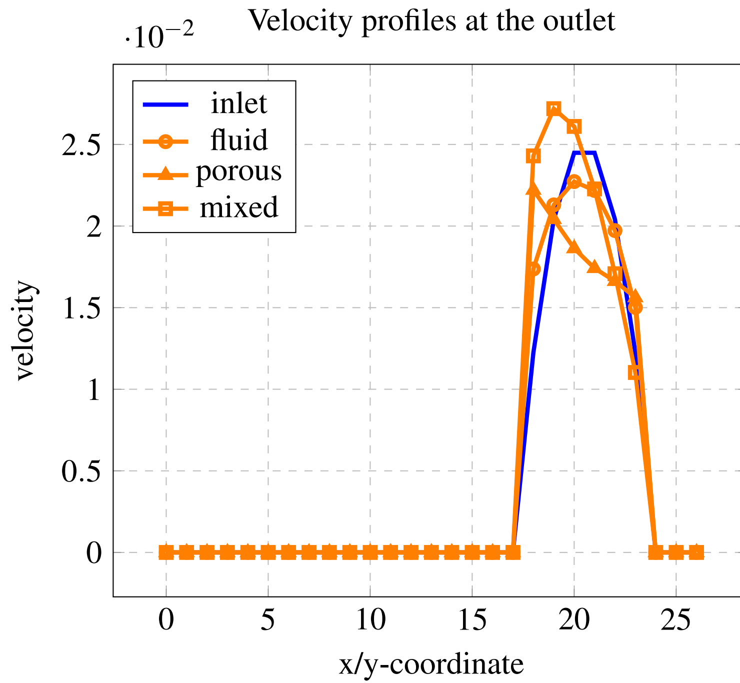


Figure 4.9: Velocity profiles at the outlet

The pressure drop of each simulation can be compared in figure 4.4. The simulation with all fluid nodes shows the smallest pressure drop with 0.0678, followed by the mixed domain with 0.2352. The pressure drops significantly more in the domain with all porous nodes which obtains a score of 0.7777 according to equation 3.11. Consequently, the all fluid domain is the best performing of the tested configurations.

4.2 Topology Optimization

The results of topology optimization using the adjoint method are presented next. Beforehand, the computational setup is specified including important variables, the grid size and the initial guess of the design variables.

4.2.1 Computational Setup

Until now, the implementation of the general adjoint method has been described and the test case has been visualized in figure 2.6. Some parameters that have not been given are described

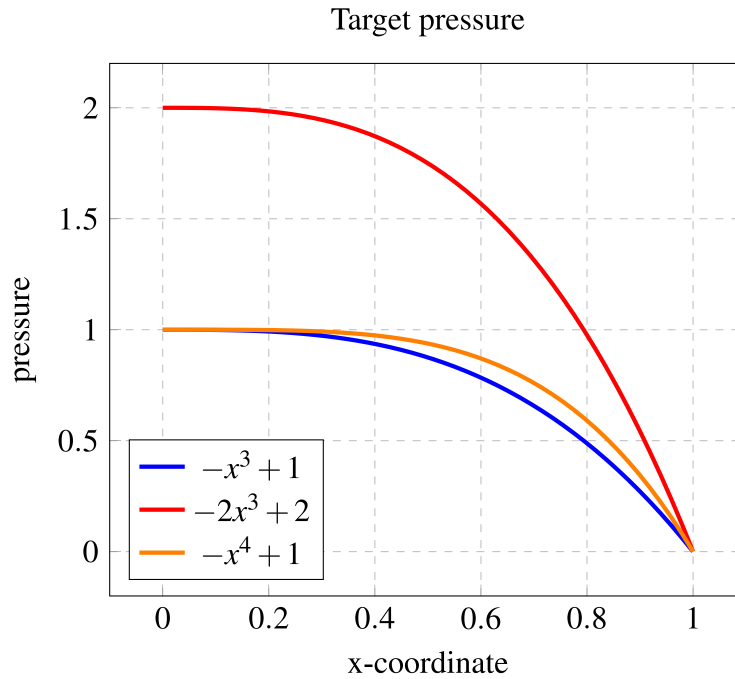


Figure 4.10: Target pressure functions

in the following. The simulations are conducted with $n = 50$ grid points along the x-axis starting at $x = 0$ until $x = 1$. The update factor of the steepest descent method $\eta = 0.02$, and the convergence criterion is 0.01. The initial guesses for the design variables κ are set to 0.6, 0.6, 1 and 0.5. As for the LBM, these parameters can be adjusted in the code as described in the manual in appendix A.

Three different target pressure functions are chosen for the simulations as visualized in figure 4.10, which all meet the pressure BC $p0 = 0$ at the end of the tube $x = 1$. This is important as the optimization algorithm cannot change the boundaries. If the pressure at $x = 1$ dissents from the boundary $p0$, the algorithm cannot approach the target function at the specified location. This either prevents the algorithm from converting or, if the target pressure differs only slightly from the BC, leads to an imperfect fit at the boundary. When choosing a function where the pressure at the end of the tube $x = 1$ is different, the BC $p0$ must be adjusted accordingly.

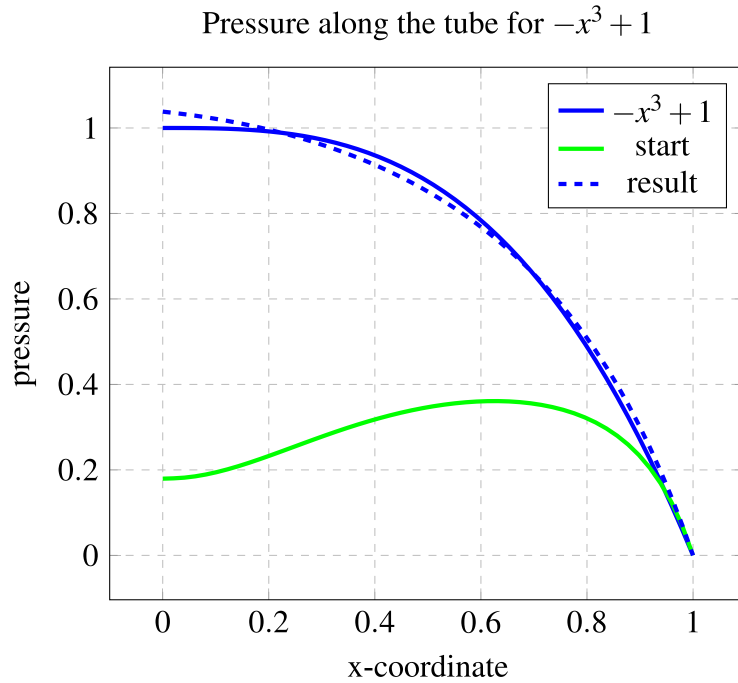


Figure 4.11: Pressure along the tube for $-x^3+1$

4.2.2 Target Pressure with Exponent 3

First, the target pressure is set to $-x^3+1$. The optimization converges after 1338 iterations and figure 4.11 visualizes the target pressure along the tube as solid blue line, the optimization result as dashed blue line and the pressure before the optimization as solid green line. The result looks similar to the target pressure. For an even better fit, the convergence criterion should be decreased. The shape of the tube is shown in figure 4.12, where the green lines indicate the initial shape, and the blue lines represent the result of this optimization.

4.2.3 Target Pressure with Exponent 3, Doubled

Second, the target pressure is set to $-2x^3+2$, which is a steeper pressure drop than at the previous example. The optimization converges after 3277 iterations, which are more than twice as many iterations as previously. The initial guess might be the reason for this, as it is much closer to the first example than this one. Figure 4.13 shows the target pressure along the tube as solid red line, the optimization result as dashed red line and the pressure before the optimization

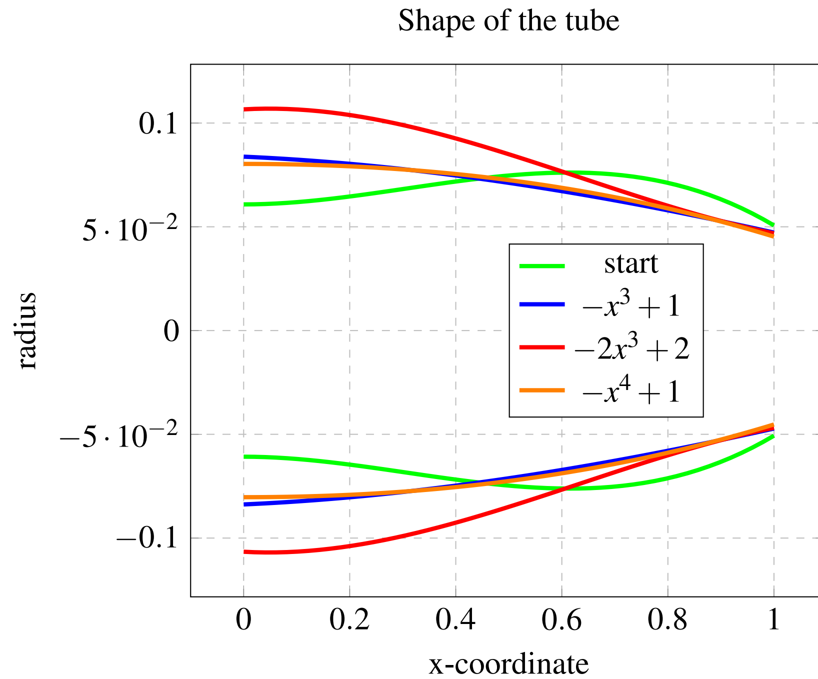


Figure 4.12: Shape of the tube

as solid green line.

The red line in figure 4.12 represents the resulting shape of the tube, where it can be compared to the previous shape in blue. While the inlet is bigger, the outlet is approximately the same size. The bigger pressure drop results in a larger change of the tube diameter for the same tube length. It also leads to a greater increase of the velocity in the tube as can be seen in figure 4.14.

4.2.4 Target Pressure with Exponent 4

Finally, the target pressure is set to $-x^4+1$ and the optimization converges after 1642 iterations, which are slightly more than for the first example. Again, figure 4.15 visualizes the target pressure as solid line and the optimization result as dashed line along with the pressure before the optimization as solid green line. The color of this example is orange, which is also used for the shape in figure 4.12 and the velocity in figure 4.14. The geometry of the tube is remarkably similar to the one of the first example which makes sense given the similar target pressure curves.

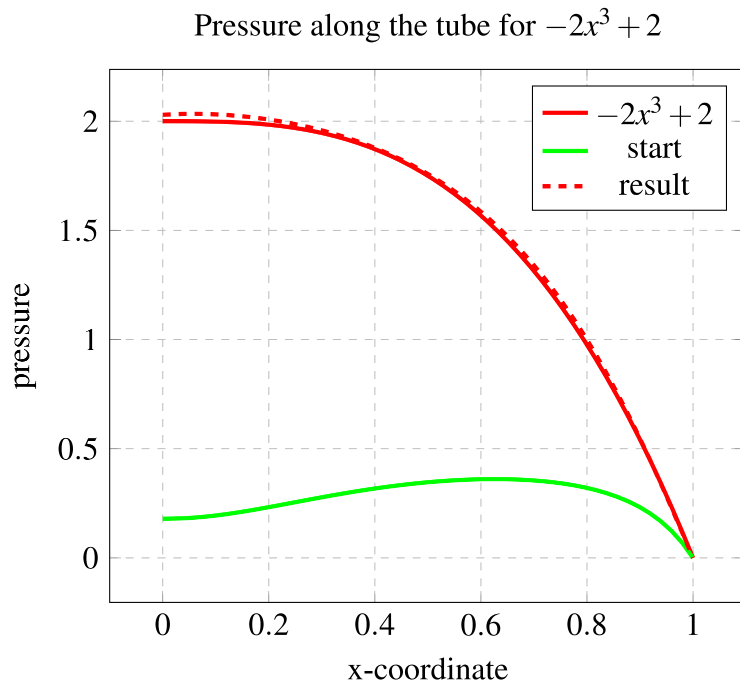


Figure 4.13: Pressure along the tube for $-2x^3+2$

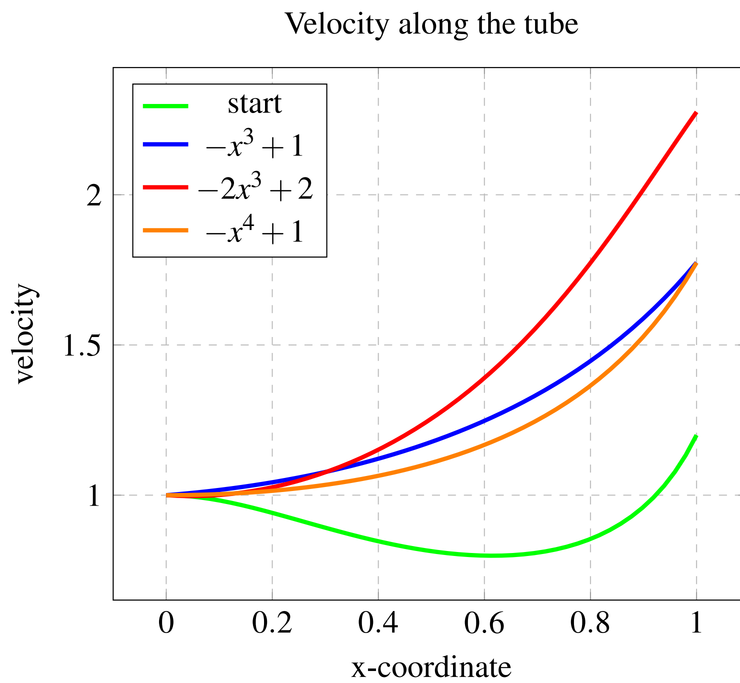


Figure 4.14: Velocity along the tube

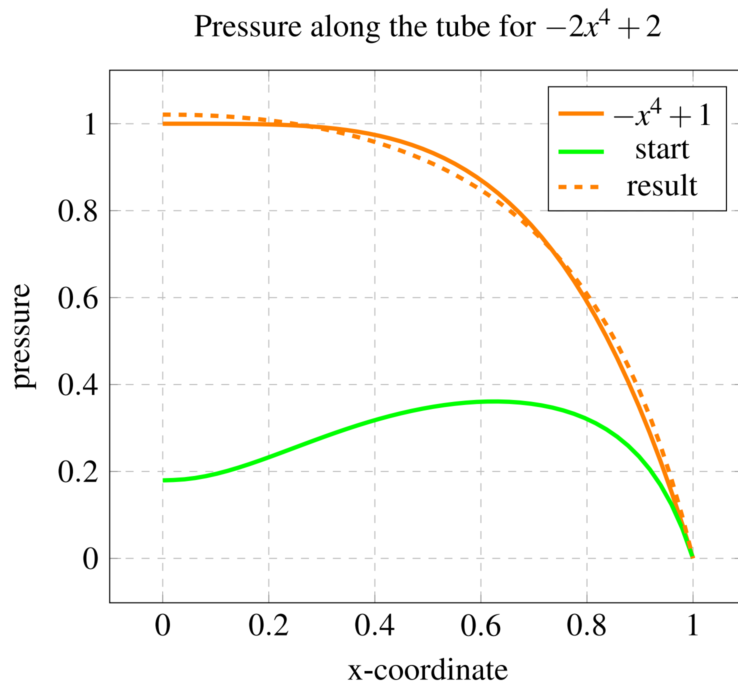


Figure 4.15: Pressure along the tube for $-x^4+1$

The development of the objective for the first 50 iterations of each optimization are visualized in figure 4.16. The objectives of the first and last optimization in blue and orange respectively start with a low objective and drop steadily, while the second starts at an elevated level and oscillates considerably. This is also the one that took most iterations to reach convergence. This illustrates that the initial guess for the design variables is a crucial factor for fast convergence.

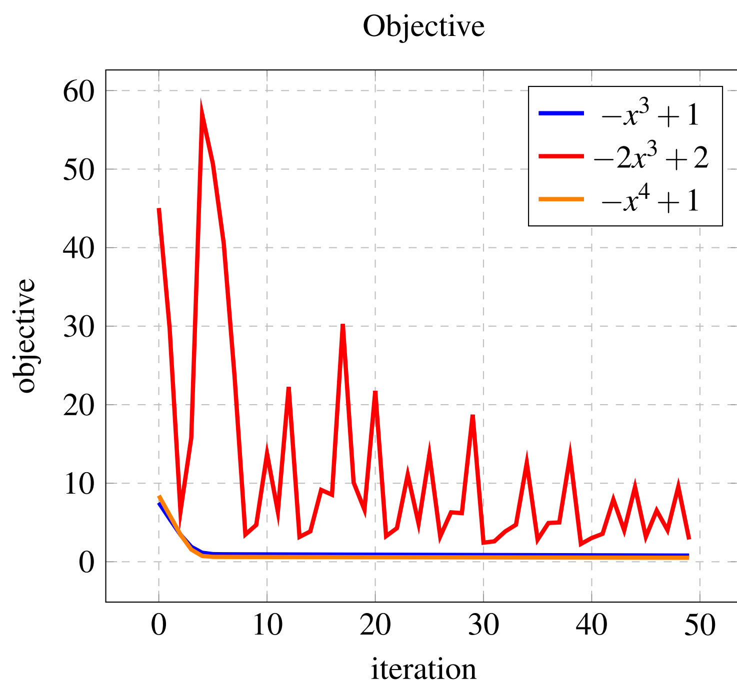


Figure 4.16: Objective at the first 50 iterations

5. Discussion

The topics LBM, topology optimization, adjoint method and blood flow modelling have been examined to prepare and conduct the described simulations. The LBM has been implemented in Python for two dimensions as well as a one dimensional adjoint method problem. The next step is to combine both approaches so that the porosities in the computational domain of the LBM are solved by the adjoint method. With the provided code, resources, and explanations, it should be possible to combine the LBM with topology optimization using the adjoint method relatively quickly. Numerical validations for the presented implementations are not given here. In future work, appropriate validations should be considered.

5.1 Usage of the Lattice Boltzmann Method

In the following, some of the difficulties that have been faced during the implementation process of the LBM are listed. The first difficulty was to transfer the theoretical knowledge into code. Even though the theory is quite straightforward, the implementation process requires high-level abstract thinking. It might have been of advantage to have more experience with implementing conventional CFD solvers or solvers for problems that exceed one dimension. The main, but not the only, challenge was to decide how to store the required information in a logical manner that supports efficient computation. By observing and comparing publicly available implementations, some conclusions were drawn. It appears reasonable to store the DVs in a matrix of the shape (ny, nx, nv) where ny is the number of nodes along the y-axis, nx is the number of nodes along the x-axis and nv is the index of DVs per node. For each type of BC, one Boolean matrix with the shape (ny, nx) is introduced where *True* indicates that

the respective node constitutes such a boundary. Matrices with the same shape are created for the macroscopic properties. Another drawback of few experiences in the implementation of CFD solvers is the insufficient knowledge about handy functions that are available in the standard libraries of the programming language. Useful sources for improvements are therefore other implementations that can be found on the internet. Using these functions helps create a well-structured code that is easily comprehensible.

The second obstacle was the ordering of the simulation steps. The final order is visualized in figure 3.1 and was explained in chapter 3.1. Conducting the steps in the proper order is essential for correct results. However, it can seem as if the procedures are interconnected, especially as the theory of the open and closed boundaries is usually explained separately and both must be divided to provide correct, updated information for each calculation. A deep understanding is necessary to decide where to begin the cycle and where to end it so that the convergence criterion is placed at a reasonable position. Also, the location to calculate the macroscopic parameters should be well-conceived. Overall, when being introduced to the theory of the LBM, it is still not obvious which steps must be conducted before others, which can lead to mistakes.

This entails the difficulty of appropriately grouping the individual steps into functions. As the logical unit of the calculation of open boundaries and the calculation of closed boundaries must be divided, in the beginning it is ambiguous how to arrange functions to add structure to the implementation.

The adjustments of the collision step for the different boundary types are another part where the initial approach was more complicated than necessary. At first, each boundary type that requires a change of the collision calculation had an individual collision function to perform these adjustments. However, the adjustments can be conducted at the velocity distribution f, so that the collision step remains unchanged for every node, independently of its type. The new function that accounts for these changes is called `fcorrection` and was discussed in chapter 3.1. Afterwards the general collision function is executed.

Another source of mistakes is the derivation of the BCs. The equations to calculate the unknown

DVs must be derived depending on the indexing of the velocity set and the location of the boundary. These calculations are prone to careless mistakes which are extremely hard to detect especially in combination with the BCs.

When being introduced to the LBM, a considerable amount of time was spent on understanding the different BCs and how they operate, some of which are fairly complex. This time is necessary to develop a deeper understanding from the general overview of the LBM and to gain the ability of choosing proper BCs for the problem at hand. Especially the details must be considered carefully when working with the LBM even if they are not relevant for the implementation in the end. This can appear overwhelming in the beginning, when the knowledge of the method is still not consolidated. Therefore, whenever possible, the BCs to be applied should be defined by a more experienced person, to limit the amount of different BC concepts that must be learnt at the same time. Then a deeper understanding of the concepts can be gained without confusing the methods among themselves.

Even though the implementation process can bring some difficulties, there are still good reasons for the use of the LBM, especially when examining blood flow. The LBM operates on the mesoscopic scale, which makes it applicable for a wider range of problems than standard CFD methods. Micro scale flows behave differently than macro scale flows which is simpler to consider with the LBM. As the LBM is based on kinetic theory, the interaction between molecules can be incorporated easily (Suga, 2013). This may be useful for the further development of the LBM in combination with topology optimization in the context of blood flow. Even though the background of this work is the optimization of larger blood vessels, the ability to model micro flows opens the possibility to extend the approach to tiny blood vessels.

Also, for topology optimization, the LBM is well suited as

- it is easy to implement the porosity model and
- the flow through complex geometries and through porous media has a better accuracy than with conventional CFD methods.

5.2 Usage of the Adjoint Method

The implementation that illustrates topology optimization has been conducted with the adjoint method, which is widely used for topology optimization. However, only a few descriptions of the procedure were found (Strang, 2007; Cao et al., 2003; Errico, 1997) even though the method can be applied to various problems for example in meteorology (Errico, 1997), for eigenproblems (Johnson, pers. comm., 2021) and for topology optimization (Strang, 2007). Most explanations are kept general, so that they can appear very abstract. This resulted in some difficulties when gaining understanding and during the implementation process. Firstly, the abstract explanations in combination with abstract examples make it hard to transfer the theory to a practical application. Beyond, different approaches must be considered depending on the nature of the problem. Consequently, each approach needs to be understood to decide on the correct one for the problem in question. Once everything is understood the next challenge is to simplify the derivative of the augmented objective function. As the integration by parts is utilized, the two parts must be chosen correctly to receive a reasonable result. Overall, the calculation to receive the equations for the topology optimization can be cumbersome. As it must be conducted only once for one system, the effort can be viewed as acceptable.
For topology optimization, usually the Lagrangian multiplier approach is applied. However, the Eulerian method is used for example by Kim and Chang (2005) and Sato et al. (2019). Currently, the advantages and disadvantages of the Lagrangian over the Eulerian method are not yet sufficiently analyzed (Sigmund and Maute, 2013). Therefore, no conclusion can be drawn about which approach should be preferred. As both approaches follow similar concepts, the differences are expected to be small.

5.3 Blood Flow

To model blood flow, several details must be considered. The models that are usually added to the LBM, implement a shear rate dependent viscosity by adjusting the relaxation time. They

are called power-law models and the Casson model as well as the Carreau-Yasuda model are most used (Boyd et al., 2007). Pingen and Maute (2010) mention the need to scale the non-Newtonian effects in porous regions for the simulation to maintain stability. After reading and understanding Boyd et al. (2007) Pingen et al. (2009) and Pingen and Maute (2010), adding a non-Newtionian model to the LBM implementation should be viable.

The example of the adjoint method was applied to the Euler equations which describe adiabatic and inviscid flow. Blood flow is usually modelled by integrating the continuity and Navier-Stokes equations along the radius of the blood vessel. For a one-dimensional model with variable internal radius $R(x,t)$ of the vessel depending on the axial distance x and time t, this can be expressed in the equation of mass and the equation of momentum respectively(Wang, 2014):

$$\frac{\partial A}{\partial t} + \frac{\partial Q}{\partial x} = 0, \tag{5.1}$$

$$\frac{\partial Q}{\partial t} + \frac{\partial}{\partial x}(\alpha \frac{Q^2}{A}) + \frac{A}{\rho}\frac{\partial p}{\partial x} = -2\pi\nu \left[\frac{\partial u_x}{\partial x}\right]_{r=R}, \tag{5.2}$$

Where A is the cross-sectional area, Q is the flow rate, p is the internal pressure, ρ is the density of the fluid, ν is the kinematic viscosity of the fluid and u_x is the axial velocity of the fluid, which is dependent on the profile of the vessel that can be described by the radius (Wang, 2014). The momentum correction coefficient α, sometimes called Coriolis coefficient, is described in more detail by Formaggia et al. (2003). For blood flow $\alpha = \frac{11}{10}$ should be used, while for a parabolic velocity profile (Poiseuille flow) $\alpha = \frac{4}{3}$ is suitable (Formaggia et al., 2003). For implementing a two-dimensional model, the reader is referred to Ghigo et al. (2017), Chakravarty and Mandal (1996), Chow and Mak (2006) and Chabannes et al. (2013) who describe blood flow in axisymmetric arteries.

During topology optimization, the geometry cannot always be described as a tube, so that the standard models for blood flow as described above provide limited applicability. Instead Pingen et al. (2009) and Pingen and Maute (2010) describe how the adjoint method can be applied to the LBM by using the streaming and collision step as governing equations. Pingen

et al. (2009) describe the procedure for Newtonian fluids and Pingen and Maute (2010) explain the additional changes to account for non-Newtonian effects. The reader is referred to these resources to implement the adjoint method for topology optimization using the LBM for non-Newtonian fluids. Other studies using the LBM in combination with non-Newtonian fluids are conducted by Chuanhu et al. (2016), Vikhansky (2012) and Conrad et al. (2015).

Besides the non-Newtonian effects of blood flow, for topology optimization of stents and arterial bypasses it might be necessary to consider the varying flow parameters of blood inside the vessels depending on the heart rate of the individual (Jiang and Strother, 2009). The ideal solution should therefore be suitable for a range of flow parameters instead of being ideal for only one pair of parameters. Further research is needed to examine how big the influence of the varying heart rates on the geometry of the blood vessels is and what measures can be taken to obtain the ideal topology for the entire range of occurring flow parameters.

5.4 Porosity

The topology optimization approach that this work aims at, is based on the idea of an initial guess of the design variable matrix β which is then optimized. The results regarding the pressure drop in figure 4.4 indicate that the domain will be predominantly occupied by fluid nodes ad the simulation with all fluid nodes received the lowest score of the objective function calculating the pressure drop. To prevent this, a volume constraint can be added as done by Pingen et al. (2007).

Moreover, the solution of the final β matrix contains values between 0 and $\frac{1}{\tau}$. It should be evaluated if the result with the minimal score of the objective function mainly consists of fluid and solid nodes or rather a great variety of porous nodes. As the real-world applications in a patient's body are not designed to involve porous media, a result with predominantly porous nodes might lead to issues when transferring the obtained results to the application in stent and bypass design. Careful adjustment of a volume constraint might minimize porous nodes to a sufficient degree. Otherwise, an activation function can be applied to transform the solution

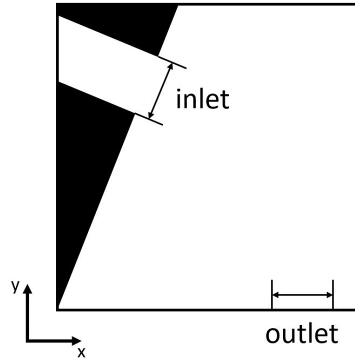


Figure 5.1: Diagonal inlet flow

matrix β to a matrix with 0 and $\frac{1}{\tau}$ elements without intermediate values. Besides, a non-homogenization method might be considered to avoid solely theoretical outcomes (Rozvany et al., 1992).

5.5 Unidirectional Inlet Flow

The current perpendicular in- and outflow of the domain are another detail, where the implementation can be improved. The implemented setup of the LBM assumes perpendicular flow in and out of the domain. However, a patient will rarely need a replacement of a blood vessel where the in- and outlet are placed at an angle of exactly 90 degrees or its multiple.

Therefore, it should be examined how to approximate real-world behavior of blood flow at specific angles. As the LBM is only defined for equidistant Cartesian grids, it is not possible to adjust the computational domain to a non-rectangular one. Additionally, no unidirectional, angled BCs are available for the LBM yet (Mattila, 2010). However, the elements of the porosity matrix can be fixed until a diagonal as visualized in figure 5.1. The inlet BC will still be perpendicular to the wall on the left-hand side, but the desired flow profile with correct direction will enter the domain that is subject to topology optimization at the diagonal. The tube length might need to be extended to reach the desired flow profile at the diagonal. Also the flow velocity should be checked and adjusted. Also Zhang and Liu (2015) achieve an angled in- and outflow by guiding the flow in a pipe before entering the domain that is subject to topology optimization.

5.6 Applicability in Medicine

The most important question about the implementation of topology optimization using the LBM is the applicability of the result to real-world problems. Customized stent adjustment might be reasonable for bifurcation lesions and for very unusual coronary geometries. For most lesions, the current approach with a flexible stent that adapts to the coronary geometry to a reasonable level appears to be a sophisticated treatment with reliable results and affordable costs. However, stent flexibility can influence clinical outcome, especially in bifurcation lesions. When a stent is ill-fitting, mechanical stress can be imposed on the artery at the stent edges and modify arterial geometry and blood flow dynamics in bifurcations. These issues are commonly referable to the rigidity of the stent rather than a poor geometry (Saito et al., 2020). A detailed analysis should be performed to identify if stents with customized geometry provide an adequate advantage over the current standard.

Nevertheless, topology optimization might already be applicable for arterial bypass operations where parts of the artery of the patient are replaced with artificial material. The geometry is usually determined by the surgeon based on experience (Abraham et al., 2005) and research on arterial bypass configurations has been done (Zhang and Liu, 2015; Abraham et al., 2005; Quarteroni and Rozza, 2003)

Another challenge is then to transfer the knowledge about the topology optimization software to the surgeons and ensure a convenient applicability. Also, the manufacturability in the available time while maintaining medical standards should be considered.

6. Conclusion

Even though the LBM is recognized among the scientific community as easy to implement, especially the implementation of BCs can be cumbersome depending on the case of application and the required complexity. For industrial applications, the lack of unidirectional BCs that are not perpendicular to the wall of the domain is especially unpractical. However, the benefit of the method might be found in the ease to incorporate various models into the LBM to model difficult flow problems or special fluids with uncommon behavior. The implementation of the porosity model was expedient and the procedure to incorporate the non-Newtonian effects seems to be simple likewise.

Regarding the applicability of the examined methods for medical problems, customized stent adjustment might be reasonable for bifurcation lesions and for very unusual coronary geometries. For most lesions, the current approach with a flexible stent that adapts to the coronary geometry to a reasonable level appears to be a sophisticated treatment with reliable results and affordable costs. The geometry of arterial bypasses on the contrary is always customized to the patient, so that the usage of topology optimization to improve bypass geometries seems more promising.

To model blood flow and use it as a tool to improve stent or arterial bypass geometries further work needs to be done to

- validate the implementations,
- combine the implemented LBM with the adjoint method,
- incorporate a volume constraint,

- add a model for non-Newtonian fluids to the LBM implementation,
- optimize geometries based on a range of heart rates instead of one set of flow parameters, and
- create a simple approach for implementing unidirectional, angled boundary flow.

Beyond that, interesting research topics regarding the LBM might include

- the comparison of different approaches of topology optimization using the LBM, and
- the implementation and validation of a porosity model to optimize topologies containing porous areas instead of only fluids and solids.

Bibliography

Aage, N., N. A. Mortensen, and O. Sigmund (2010). Topology optimization of metallic devices for microwave applications: Topology optimization of electromagnetic waves. *International journal for numerical methods in engineering 83*(2), 228–248.

Abraham, F., M. Behr, and M. Heinkenschloss (2005). Shape optimization in steady blood flow: A numerical study of non-newtonian effects. *Computer methods in biomechanics and biomedical engineering 8*(2), 127–137.

Alonso, D. H., J. S. R. Saenz, and E. C. N. Silva (2019). Non-newtonian laminar 2d swirl flow design by the topology optimization method. *Structural and multidisciplinary optimization 62*(1), 299–321.

Alonso, D. H. and E. C. N. Silva (2021). Topology optimization for blood flow considering a hemolysis model. *Structural and multidisciplinary optimization 63*(5), 2101–2123.

Barnes, H. A. (2002). *Viscosity*. Aberystwyth: Univ. of Wales, Inst. of Non-Newtonian Fluid Mechanics.

Bendsoe, M. P. and O. Sigmund (2004). *Topology Optimization: Theory, Methods, and Applications* (Second Edition, Corrected Printing. ed.). Berlin, Heidelberg: Springer Berlin Heidelberg.

Boyd, J., J. M. Buick, and S. Green (2007). Analysis of the casson and carreau-yasuda non-newtonian blood models in steady and oscillatory flows using the lattice boltzmann method. *Physics of fluids (1994) 19*(9), 093103–1 – 093103–14.

Cao, Y., S. Li, L. Petzold, and R. Serban (2003). Adjoint sensitivity analysis for differential-algebraic equations: The adjoint dae system and its numerical solution. *SIAM journal on scientific computing 24*(3), 1076–1089.

Chabannes, V., G. Pena, and C. Prud'homme (2013). High-order fluid–structure interaction in 2d and 3d application to blood flow in arteries. *Journal of computational and applied mathematics 246*, 1–9.

Chakravarty, S. and P. Mandal (1996). A nonlinear two-dimensional model of blood flow in an overlapping arterial stenosis subjected to body acceleration. *Mathematical and computer modelling 24*(1), 43–58.

Chapman, S. (1939). *The mathematical theory of non-uniform gases : an account of the kinetic theory of viscosity, thermal conduction, and diffusion in gases*. Cambridge: at the University Press.

Cheylan, I., G. Fritz, D. Ricot, and P. Sagaut (2019). Shape optimization using the adjoint lattice boltzmann method for aerodynamic applications. *AIAA journal 57*(7), 2758–2773.

Chow, K. and C. Mak (2006). A simple model for the two dimensional blood flow in the collapse of veins. *Journal of mathematical biology 52*(6), 733–744.

Chuanhu, Z., C. Songgui, S. Qicheng, and J. Feng (2016). Free-surface simulations of newtonian and non-newtonian fluids with the lattice boltzmann method. *Acta geologica Sinica (Beijing) 90*(3), 999–1010.

Conrad, D., A. Schneider, and M. Böhle (2015). Accuracy of non-newtonian lattice boltzmann simulations. *Journal of computational physics 301*, 218–229.

De Avila Belbute-Peres, F., T. Economon, and Z. Kolter (2020). Combining differentiable PDE solvers and graph neural networks for fluid flow prediction. In H. D. III and A. Singh (Eds.), *Proceedings of the 37th International Conference on Machine Learning*, Volume 119 of *Proceedings of Machine Learning Research*, pp. 2402–2411. PMLR.

Deng, Y., Y. Wu, and Z. Liu (2018). *Topology Optimization Theory for Laminar Flow: Applications in Inverse Design of Microfluidics*. Singapore: Springer Singapore.

Dugast, F., Y. Favennec, C. Josset, Y. Fan, and L. Luo (2018). Topology optimization of thermal fluid flows with an adjoint lattice boltzmann method. *Journal of computational physics 365*, 376–404.

Errico, R. M. (1997). What is an adjoint model? *Bulletin of the American Meteorological Society 78*(11), 2577–2591.

Evgrafov, A. (2005). The Limits of Porous Materials in the Topology Optimization of Stokes Flows. *Applied Mathematics and Optimization 52*(3), 263–277.

Evgrafov, A. (2006). Topology optimization of slightly compressible fluids. *ZAMM 86*(1), 46–62.

Farahbakhsh, I. (2020). *Krylov Subspace Methods with Application in Incompressible Fluid Flow Solvers*. Newark: John Wiley & Sons Ltd.

Formaggia, L., D. Lamponi, and A. Quarteroni (2003). One-dimensional models for blood flow in arteries. *Journal of engineering mathematics 47*(3), 251–276.

Ghigo, A. R., J.-M. Fullana, and P.-Y. Lagrée (2017). A 2d nonlinear multiring model for blood flow in large elastic arteries. *Journal of computational physics 350*, 136–165.

Guo, Z. and C. Shu (2013). *Lattice Boltzmann Method and Its Applications in Engineering*, Volume 3 of *Advances in Computational Fluid Dynamics*. Singapore: World Scientific.

Hamed, J. K. and K. M. Ramin (2020). A novel approach of unsteady adjoint lattice boltzmann method based on circular function scheme. *Journal of scientific computing 85*(2), 38–1 – 38–29.

He, X. and L.-S. Luo (1997). A priori derivation of the lattice boltzmann equation. *Physical review. E, Statistical physics, plasmas, fluids, and related interdisciplinary topics 55*(6), R6333–R6336.

Hekmat, M. H. and M. Mirzaei (2014). Continuous and discrete adjoint approach based on lattice boltzmann method in aerodynamic optimization part i: Mathematical derivation of adjoint lattice boltzmann equations. *Advances in applied mathematics and mechanics 6*(5), 570–589.

Huang, H., M. Sukop, and X. Lu (2015). *Multiphase Lattice Boltzmann Methods: Theory and Application*. New York: John Wiley & Sons Ltd.

Hwang, W. R. and S. G. Advani (2010). Numerical simulations of Stokes–Brinkman equations for permeability prediction of dual scale fibrous porous media. *Physics of Fluids 22*(11), 113101–1 – 113101–14.

James, K. A. and H. Waisman (2016). Layout design of a bi-stable cardiovascular stent using topology optimization. *Computer methods in applied mechanics and engineering 305*, 869–890.

Jiang, J. and C. Strother (2009). Computational fluid dynamics simulations of intracranial aneurysms at varying heart rates: A "patient-specific" study. *Journal of biomechanical engineering 131*(9), 091001–1 – 091001–11.

Karpouzas, G. K., E. M. Papoutsis-Kiachagias, T. Schumacher, E. de Villiers, K. C. Giannakoglou, and C. Othmer (2016). Adjoint optimization for vehicle external aerodynamics. *International Journal of Automotive Engineering 7*(1), 1–7.

Kim, N. H. and Y. Chang (2005). Eulerian shape design sensitivity analysis and optimization with a fixed grid. *Computer methods in applied mechanics and engineering 194*(30), 3291–3314.

Kreissl, S., G. Pingen, and K. Maute (2011). An explicit level set approach for generalized shape optimization of fluids with the lattice Boltzmann method. *International Journal for Numerical Methods in Fluids 65*(5), 496–519.

Krüger, T., H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen (2017). *The Lattice Boltzmann Method: Principles and Practice*. Graduate Texts in Physics. Cham: Springer International Publishing.

LaDisa Jr, J. F., I. Guler, L. E. Olson, D. A. Hettrick, J. R. Kersten, D. C. Warltier, and P. S. Pagel (2003). Three-dimensional computational fluid dynamics modeling of alterations in coronary wall shear stress produced by stent implantation. *Annals of biomedical engineering 31*(8), 972–980.

Li, X., L. Fang, and Y. Peng (2018). Airfoil design optimization based on lattice boltzmann method and adjoint approach. *Applied mathematics and mechanics 39*(6), 891–904.

Mattila, K. (2010). *Implementation Techniques for the Lattice Boltzmann Method*. Ph. D. thesis, Jyväskylän Yliopisto.

Mohamad, A. A. (2011). *Lattice Boltzmann Method*. London: Springer London.

Niccoli, G. and I. Eitel (2018). *Coronary Microvascular Obstruction in Acute Myocardial Infarction: From Mechanisms to Treatment*. San Diego: Elsevier Science and Technology.

Noble, D. R., S. Chen, J. G. Georgiadis, and R. O. Buckius (1995). A consistent hydrodynamic boundary condition for the lattice boltzmann method. *Physics of fluids (1994) 7*(1), 203–209.

Obiols-Sales, O., A. Vishnu, N. Malaya, and A. Chandramowlishwaran (2020). CFDNet: a deep learning-based accelerator for fluid simulations. *Proceedings of the 34th ACM International Conference on Supercomputing 1*, 1–12.

Pingen, G., A. Evgrafov, and K. Maute (2007). Topology optimization of flow domains using the lattice Boltzmann method. *Structural and Multidisciplinary Optimization 34*(6), 507–524.

Pingen, G., A. Evgrafov, and K. Maute (2009). Adjoint parameter sensitivity analysis for the hydrodynamic lattice Boltzmann method with applications to design optimization. *Computers & Fluids 38*(4), 910–923.

Pingen, G. and K. Maute (2010). Optimal design for non-newtonian flows using a topology optimization approach. *Computers and mathematics with applications (1987) 59*(7), 2340–2350.

Plessix, R.-E. (2006). A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. *Geophysical Journal International 167*(2), 495–503.

Quarteroni, A. and G. Rozza (2003). Optimal control and shape optimization of aorto-coronaric bypass anastomoses. *Mathematical models and methods in applied sciences 13*(12), 1801–1823.

Ribeiro, N. S., J. Folgado, and H. C. Rodrigues (2021). Surrogate-based multi-objective design optimization of a coronary stent: Altering geometry toward improved biomechanical performance. *International journal for numerical methods in biomedical engineering 37*(e3453), e3453–1 – e3453–24.

Romero, J. S. and E. C. N. Silva (2017). Non-newtonian laminar flow machine rotor design by using topology optimization. *Structural and multidisciplinary optimization 55*(5), 1711–1732.

Rozvany, G., M. Zhou, and T. Birker (1992). Generalized shape optimization without homogenization. *Structural optimization 4*(3-4), 250–252.

Rutkowski, M., W. Gryglas, J. Szumbarski, C. Leonardi, and L. Laniewski-Wollk (2020). Open-loop optimal control of a flapping wing using an adjoint lattice boltzmann method. *Computers and mathematics with applications (1987) 79*(12), 3547–3569.

Saito, N., T. Komatsu, and Y. Mori (2020). Influence of stent flexibility on artery wall stress and wall shear stress in bifurcation lesions. *Medical devices (Auckland, N.Z.) 13*(2), 365–375.

Sato, Y., K. Izui, T. Yamada, S. Nishiwaki, M. Ito, and N. Kogiso (2019). Reliability-based topology optimization under shape uncertainty modeled in eulerian description. *Structural and multidisciplinary optimization 59*(1), 75–91.

Sigmund, O. and K. Maute (2013). Topology optimization approaches: A comparative review. *Structural and Multidisciplinary Optimization 48*(6), 1031–1055.

Spaid, M. A. A. and F. R. Phelan (1997). Lattice Boltzmann methods for modeling microscale flow in fibrous porous media. *Physics of Fluids 9*(9), 2468–2474.

Strang, G. (2007). *Computational science and engineering*. Wellesley, Mass: Wellesley-Cambridge Press.

Succi, S. (2001). *The lattice Boltzmann equation for fluid dynamics and beyond*. Numerical mathematics and scientific computation. Oxford: Clarendon Press.

Suga, K. (2013). Lattice boltzmann methods for complex micro-flows: applicability and limitations for practical applications. *Fluid dynamics research 45*(3), 34501–1 – 34501–31.

Vergnault, E. and P. Sagaut (2014). An adjoint-based lattice boltzmann method for noise control problems. *Journal of computational physics 276*(276), 39–61.

Vikhansky, A. (2012). Construction of lattice-boltzmann schemes for non-newtonian and two-phase flows. *Canadian journal of chemical engineering 90*(5), 1081–1091.

Wang, X. (2014). *1D modeling of blood flow in networks: numerical computing and applications*. Ph. D. thesis, Université Pierre et Marie Curie - Paris.

Wentzel, J. J., D. M. Whelan, W. J. van der Giessen, H. M. van Beusekom, I. Andhyiswara, P. W. Serruys, C. J. Slager, and R. Krams (2000). Coronary stent implantation changes 3-d vessel geometry and 3-d shear stress distribution. *Journal of biomechanics 33*(10), 1287–1295.

Yu, D., R. Mei, and W. Shyy (2005). Improved treatment of the open boundary in the method of lattice boltzmann equation. *Progress in computational fluid dynamics 5*(1-2), 3–12.

Zhang, B. and X. Liu (2015). Topology optimization study of arterial bypass configurations using the level set method. *Structural and Multidisciplinary Optimization 51*(3), 773–798.

Zhang, B. C., S. X. Tu, A. Karanasos, R. v. Geuns, P. d. Jaegere, F. Zijlstra, and E. Regar (2018). Association of stent-induced changes in coronary geometry with late stent failure: Insights from three-dimensional quantitative coronary angiographic analysis. *Catheterization and cardiovascular interventions 92*(6), 1040–1048.

Zhu, J. and T. Gao (2016). *Topology Optimization in Engineering Structure Design*. San Diego, CA, USA: Elsevier Science.

Zou, Q. and X. He (1997). On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids 9*(6), 1591–1598.

Appendices

A. User Manual

This user manual gives an overview of the structure of the attached code beyond the comments contained in the code itself.

A.1 Lattice Boltzmann Method

The code of the LBM is divided into two parts:

- the file *myLBM.py*, where the simulation is run and
- the collection of functions *lbfuns.py*.

The former is used to run the provided and described test case and make changes on:

- the number of nodes in x- and y-direction `n`,
- the maximum number of time steps `nt`,
- the maximum velocity at the inlet `uxInMax`,
- the pressure at the outlet `pOut`,
- the viscosity `v`,
- the factor for polynomial scaling of the porosity `k`, described in 2.2 and
- the initial guess for the porosity matrix `beta`.

Additionally, the Boolean variable `plot` specifies if the domain is plotted during the simulation in addition to the final plots.

The latter provides the function `solveTestCase` that solves the described test case, the functions that have been described in the chapter 3.1 and functions for plotting. To set up a new test case, either a separate function can be created for the new setup, or the existing one can be modified.

A.2 Adjoint Method

The code of the adjoint method is also divided into two parts:

- the file *myAdjoint.py*, where the simulation is run and
- the collection of functions *adjointfuns.py*.

The former conducts the adjoint sensitivity analysis of the example described in chapter 2.3.2, where the following properties are designed for adjustment/can be adjusted:

- the number of grid points `n`,
- the maximum number of iterations `maxIt`,
- the velocity `u0` at the entrance of the tube ($x = 0$),
- the pressure `p0` at the exit of the tube ($x = 1$),
- the convergence criterion `conv`,
- the update factor `eta` of the steepest descent method,
- the initial guess for the design variables `kappa`.

The latter provides the functions used in the optimization loop of *myAdjoint.py* which are predominantly specific to the example and must be adjusted or recreated for a new problem. The function `pTarget`, defining the target pressure, was adjusted for the results presented in chapter 4.2

B. Python Code

B.1 Lattice Boltzmann Method

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib as mpl
import copy

def solveTestCase(n,nt,plot,uxInMax,pOut,beta,v,k):
    '''
    Lattice Boltzmann Method

                  5   1   6
                   \  |  /
                    \ | /
          D2Q9    4 --0-- 2
                    / | \
                   /  |  \
                  8   3   7

    Solve the test case with quadratic domain and velocity inlet at the
                                              left and pressure outlet at the
                                             bottom.
    General variables:
    n:      number of nodes in x and y direction
    nt:     number of time steps
```

```
22      plot:   enable real-time plotting
23
24      Boundary conditions:
25      uxInMax:    maximum velocity at the inlet
26      pOut:       pressure at the outlet
27
28      Variables for the porosity model:
29      beta:       guess for beta matrix
30      ktow:       tow permeability
31      k:          factor for polynomial scaling of the porosity model; k
                                > 1; Recommendation: 2 < k < 3
32      fluidLimit: value between 0 and 1 below which the node is treated
                                as bounce-back boundary; 0: solid;
                                1: fluid
33      '''
34      # Simulation parameters
35      nx = n # number of nodes in x-direction
36      ny = n # number of nodes in y-direction
37      rho0 = 1 # initial density
38      dt = 1 # size of a time step
39      cs = 1/np.sqrt(3) # lattice speed of sound
40      fluidLimit = 0.2 # when poro exceeds fluidLimit, the node is
                                considered solid
41      # Calculate variables related to porosity
42      tau, poro, fluid = porosityParam(beta, v, fluidLimit, k)
43
44      # Lattice parameters
45      X, Y = np.meshgrid(range(nx), range(ny))
46      nv, idxs, cxs, cys, wghts = D2Q9()
47
48      # Initial conditions
49      f = np.ones((ny,nx,nv)) # matrix for the results of the current
                                time step
50      fOld = np.copy(f)
```

```
51      feq = np.zeros((ny,nx,nv), dtype=float) # matrix for the collision
                                                  step (Equilibrium)
52      rho = np.sum(f,2) # sum up the velocities of each grid point
53      for i in idxs:
54          f[:,:,i] *= rho0 / rho
55
56      # Boundaries
57      # Solid boundary
58      bndry = np.full((ny,nx), False)
59      bndry[fluid==False] = True
60      bndry[ 0] = True
61      bndry[-1] = True
62      bndry[:, 0] = True
63      bndry[:,-1] = True
64      # pressure boundary
65      pbndry = np.full((ny,nx), False)
66      pbndry[0,int(ny*0.7):int(ny*0.9)] = True
67      bndry[pbndry] = False
68      # Velocity boundary
69      vbndry = np.full((ny,nx), False)
70      vbndry[int(ny*0.7):int(ny*0.9),0] = True
71      bndry[vbndry] = False
72
73      # Inlet Velocity
74      uxIn = parabolVelo(Y, uxInMax, vbndry)
75      uyIn = 0
76      # Outlet Pressure
77      rhoOut = pOut / (cs**2)
78      bndryCond = [uxIn, uyIn, rhoOut]
79
80      # Initial velocities
81      ux = 0 # velocity in x direction
82      uy = 0 # velocity in y direction
83      umaxOld = np.max(np.sqrt(ux**2+uy**2)) # save maximum velocity
```

```
    # Initiate plot
    if plot:
        fig, ax = plt.subplots()

    # for each time step
    for time in range(nt):
        #if time % 100 == 0:
        print('Time step: ', time)

        # Propagation (Streaming)
        f = propagation(f, idxs, cxs, cys)

        # Store bounce-back values
        if bndry.any:
            bb = f[bndry]
            bb  = bb[:,[0,3,4,1,2,7,8,5,6]]

        # Calculate macroscopic parameters
        rho, ux, uy = macroscopicParam(f, vbndry, pbndry, bndryCond,
                                            cxs,cys)

        # Calculate equilibrium distribution
        feq = equiPoro(feq, rho, ux, uy, idxs, cxs, cys, wghts, poro)

        # Calculate boundary distributions
        f = fcorrection(f, feq, rho, ux, uy, vbndry, pbndry)

        # Collision
        f = collision(f, feq, tau, dt, nv)

        # Collision for fullway bounce-back boundary
        if bndry.any:
            f[bndry] = bb
```

```
117
118         # Check convergence
119         criterion, fOld = convergenceF(f, fOld, ny, nx, nv)
120         if (criterion < 1e-16 and time > 10):
121             print('Converged.')
122             break
123
124         # plot in real time
125         if (plot and (time % 10) == 0) or (time == nt-1):
126             plotQuiverPoro(X,Y, ux, uy, bndry, ax, poro)
127             print(criterion)
128
129     # Calculate macroscopic parameters
130     rho, ux, uy = macroscopicParam(f, vbndry, pbndry, bndryCond, cxs,
                                        cys)
131
132     print('Pressure drop: ', ( np.sum(rho[vbndry]) - np.sum(rho[pbndry
                                        ]) )* (cs**2))
133     plotFinal(ux,uy,bndry,nx,ny,X,Y,rho,vbndry,pbndry,cs,poro)
134     return f, rho, ux, uy
135
136 def D2Q9():
137     ''' Velocity set D2Q9
138     nv:     number of velocities
139     idxs:   indices of the velocities
140     cxs:    x-coordinates of each velocity vector
141     cys:    y-coordinates of each velocity vector
142     wghts:  weights of the respective velocity vectors
143     '''
144     nv    = 9
145     idxs  = np.arange(nv)
146     cxs   = np.array([ 0, 0, 1, 0,-1, -1, 1, 1,-1])
147     cys   = np.array([ 0, 1, 0,-1, 0,  1, 1,-1,-1])
148     wghts = np.array([4/9,1/9,1/9,1/9,1/9,1/36,1/36,1/36,1/36])
```

```
149     return nv, idxs, cxs, cys, wghts
150 
151 def parabolVelo(Y, uxInMax, vbndry):
152     ''' Calculate inlet velocities with parabolic profile
153     ny:          number of nodes in y-direction
154     Y:           matrix with y-coordinates of the nodes
155     uxInMax:     maximum velocity in x-direction at the inlet
156     uxLen:       Size of the inlet
157     uxIn:        velocities in x-direction at the inlet
158     '''
159     inletLen = np.count_nonzero(vbndry)
160     uxLen = inletLen+2
161     a = -4*uxInMax/(uxLen-1)**2
162     uxIn = a*Y[:uxLen,0]**2 - a*Y[:uxLen,0]*(uxLen-1)
163     return uxIn[1:-1]
164 
165 def porosityParam(beta, v, fluidLimit, k):
166     ''' Calculate the parameters for the porosity model when the
                                        viscosity is fixed
167     beta:        porosity measure; 0: fluid, 1/tau: porous
168     v:           viscosity
169     fluidLimit: value between 0 and 1 below which the node is treated
                                        as bounce-back boundary; 0: solid;
                                        1: fluid
170     k:           factor for polynomial scaling
171     tau:         relaxation time; must be greater 0.5
172     poro:        porosity measure; 1: fluid; 0: porous
173     fluid:       boolean matrix that is True for fluid nodes
174     '''
175     tau = (6*v+1)/2 # relation with viscosity is defined by the
                                        velocity set D2Q9
176     poro = 1-(beta*tau)**k # 1: fluid; 0: porous
177     fluid = poro > fluidLimit
178     return tau, poro, fluid
```

```
179
180 def check(v,beta,k=2.5):
181     ''' Check the possible range of ktow, given a viscosity v
182     v:          viscosity
183     ktow:       permeability
184     k:          factor for polynomial scaling
185     tau:        relaxation time; must be greater 0.5
186     betamax:    maximum beta to respect the porosity model
187     ktowmin:    minimum ktow to respect the porosity model
188     beta:       porosity measure; 0: fluid, 1/tau: porous
189     poro:       porosity measure; 1: fluid; 0: porous
190     fluid:      boolean matrix that is True for fluid nodes
191     '''
192     tau = (6*v+1)/2
193     betamax = 1/tau
194     ktowmin = v/betamax
195     print('Tau: ', tau, '\nPossible range of beta: 0 to ', betamax, '\
                                    nPossible range of ktow: ', ktowmin
                                    , ' to infinity')
196     poro = 1-(beta*tau)**k
197     return poro
198
199 def propagation(f, idxs, cxs, cys):
200     ''' Propagation step of the Lattice Boltzmann Method
201     f:      discrete velocities
202     idxs:   indices of the velocity set
203     cxs:    x-coordinates of the velocity set
204     cys:    y-coordinates of the velocity set
205     '''
206     for i, cx, cy in zip(idxs, cxs, cys): # for each velocity
                                    discretization direction
207         f[:,:,i] = np.roll(f[:,:,i], cx, axis=1) # stream along x axis
208         f[:,:,i] = np.roll(f[:,:,i], cy, axis=0) # stream along y axis
209     return f
```

```
210
211 def macroscopicParam(f, vbndry, pbndry, bndryCond, cxs,cys):
212     ''' Calculate the macroscopic parameters of the
213     f:          discrete velocities
214     vbndry:     boolean matrix that is True for the velocity boundary
215     pbndry:     boolean matrix that is True for the pressure boundary
216     bndryCond:  list of variables of the boundary conditions
217     cxs:        x-coordinates of the velocity set
218     cys:        y-coordinates of the velocity set
219     rho:        densities
220     ux:         velocities in x-direction
221     uy:         velocities in y-direction
222     '''
223     uxIn = bndryCond[0]
224     uyIn = bndryCond[1]
225     rhoOut = bndryCond[2]
226
227     # Calculate macroscopic parameters
228     rho = np.sum(f,2)
229     ux  = np.sum(f*cxs,2)/rho
230     uy  = np.sum(f*cys,2)/rho
231
232     # At the boundary (velocity, inlet)
233     if vbndry.any:
234         ux[vbndry] = uxIn
235         uy[vbndry] = uyIn
236         rho[vbndry] = 1/(1-ux[vbndry]) * (f[vbndry,0]+f[vbndry,1]+f[
                                              vbndry,3]+2*(f[vbndry,4]+f[vbndry
                                              ,5]+f[vbndry,8]))
237
238     # At the boundary (pressure, outlet)
239     if pbndry.any:
240         rho[pbndry] = rhoOut
241         ux[pbndry] = 0
```

```
242         uy[pbndry] = 1-((f[pbndry,0] + f[pbndry,2] + f[pbndry,4] + 2*(f
                                            [pbndry,3] + f[pbndry,7] + f[pbndry
                                            ,8]))) /rho[pbndry]
243
244     return rho, ux, uy
245
246 def equi(feq, rho, ux, uy, idxs, cxs, cys, wghts):
247     ''' Calculate the equilibrium distribution
248     feq:    equilibrium distribution
249     rho:    density
250     ux:     velocities in x-direction
251     uy:     velocities in y-direction
252     idxs, cxs, cys, wghts: variables of the D2Q9 velocity set
253     '''
254     for i, cx, cy, w in zip(idxs, cxs, cys, wghts):
255         feq[:,:,i] = rho * w * ( 1 + 3*(cx*ux+cy*uy) + 9/2*(cx*ux+cy*uy
                                            )**2 - 3/2*(ux**2+uy**2) )
256     return feq
257
258 def equiPoro(feq, rho, ux, uy, idxs, cxs, cys, wghts, poro):
259     ''' Calculate the equilibrium distribution
260     feq:    equilibrium distribution
261     rho:    density
262     ux:     velocities in x-direction
263     uy:     velocities in y-direction
264     idxs, cxs, cys, wghts: variables of the D2Q9 velocity set
265     poro:   porosity measure; 1: fluid; 0: porous
266     '''
267     for i, cx, cy, w in zip(idxs, cxs, cys, wghts):
268         feq[:,:,i] = rho * w * ( 1 + 3*(cx*ux*poro+cy*uy*poro) + 9/2*(
                                            cx*ux*poro+cy*uy*poro)**2 - 3/2*((
                                            ux*poro)**2+(uy*poro)**2) )
269     return feq
270
```

```python
def fcorrection(f, feq, rho, ux, uy, vbndry, pbndry):
    ''' Correct the discrete velocities at the open boundaries
    f:          discrete velocities
    feq:    equilibrium distribution
    rho:        densities
    ux:         velocities in x-direction
    uy:         velocities in y-direction
    vbndry:     boolean matrix that is True for the velocity boundary
    pbndry:     boolean matrix that is True for the pressure boundary
    bndryCond:  list of variables of the boundary conditions
    '''
    # At the boundary (velocity, inlet)
    if vbndry.any:
        #f[vbndry] = np.copy(feq[vbndry])

        f[vbndry,2]=feq[vbndry,2]+f[vbndry,4]-feq[vbndry,4]
        f[vbndry,6]=0.5*(rho[vbndry]*(ux[vbndry]+uy[vbndry])-f[vbndry
                                    ,1]-f[vbndry,2]+f[vbndry,3]+2*f[
                                    vbndry,8]+f[vbndry,4])
        f[vbndry,7]=0.5*(rho[vbndry]*(ux[vbndry]-uy[vbndry])+f[vbndry
                                    ,1]-f[vbndry,2]-f[vbndry,3]+f[
                                    vbndry,4]+2*f[vbndry,5])

    # At the boundary (pressure, outlet)
    if pbndry.any:
        f[pbndry,1] = feq[pbndry,1] + (f[pbndry,3] - feq[pbndry,3])
        f[pbndry,5] = 1/2 * (rho[pbndry]*(uy[pbndry]-ux[pbndry]) - f[
                                    pbndry,1] + f[pbndry,2] + f[pbndry
                                    ,3] - f[pbndry,4] + 2*f[pbndry,7])
        f[pbndry,6] = 1/2 * (rho[pbndry]*(ux[pbndry]+uy[pbndry]) - f[
                                    pbndry,1] - f[pbndry,2] + f[pbndry
                                    ,3] + f[pbndry,4] + 2*f[pbndry,8])

    return f
```

```
297
298 def collision(f, feq, tau, dt, nv):
299     ''' Collision step for the Lattice Boltzmann Method
300     f:      discrete velocities
301     feq:    equilibrium distribution
302     tau:    relaxation time
303     dt:     time step
304     nv:     number of discrete velocities
305     '''
306     for i in range(nv): # Apply collision for all the nodes
307         f[:,:,i] -= (dt/tau) * (f[:,:,i]-feq[:,:,i])
308     return f
309
310 def convergenceVelo(ux,uy,umaxOld):
311     # Check convergence with the max velocity
312     umaxNew = np.max(np.sqrt(ux**2+uy**2)) # save maximum velocity
313     criterion = np.abs((umaxOld-umaxNew)/umaxNew)
314     umaxOld = umaxNew
315     return criterion, umaxOld
316
317 def convergenceF(f, fOld, ny, nx, nv):
318     f = np.array(f)
319     criterion = abs(f - fOld).reshape(ny*nx*nv)
320     fOld = np.copy(f)
321     return max(criterion), fOld
322
323 def plotQuiverPoro(X,Y, ux, uy, bndry,ax,poro):
324     ''' Plot the computational domain with velocity arrows and porosity
                                              '''
325     plt.cla() # clear the current axes
326     ux[bndry] = 0
327     uy[bndry] = 0
328
```

```
329     q = ax.quiver(X[bndry==False], Y[bndry==False], ux[bndry==False],
                                        uy[bndry==False], np.hypot(ux[bndry
                                        ==False], uy[bndry==False]) , pivot
                                        ='mid')
330
331     vorticity = (np.roll(ux, -1, axis=0) - np.roll(ux, 1, axis=0)) - (
                                        np.roll(uy, -1, axis=1) - np.roll(
                                        uy, 1, axis=1))
332     vorticity[bndry] = np.nan
333     cmap = copy.copy(mpl.cm.get_cmap("bwr"))
334     cmap.set_bad('black')
335     plt.imshow(vorticity, cmap=cmap)
336
337     # Show porosity
338     cmapPoro = copy.copy(mpl.cm.get_cmap('Greens'))
339     cmapPoro.set_bad(color='black')
340     norm = mpl.colors.Normalize(vmin=0, vmax=1)
341     poro[bndry] = np.nan
342     plt.imshow(1-poro, cmap=cmapPoro, norm=norm, alpha=0.5)
343     ax.invert_yaxis()
344     ax = plt.gca()
345     ax.get_xaxis().set_visible(False)
346     ax.get_yaxis().set_visible(False)
347     ax.set_aspect('equal')
348     plt.pause(0.001)
349     return
350
351 def plotFinal(ux,uy,bndry,nx,ny,X,Y,rho,vbndry,pbndry,cs,poro):
352     ''' Final figure and axes '''
353     f = plt.figure(figsize=(10,10))
354     ax = [f.add_subplot(221),f.add_subplot(222),f.add_subplot(223),f.
                                        add_subplot(224)]
355     ax[0].set_title('Velocity profile')
356     ax[0].set_xlabel('x')
```

```
357     ax[1].set_title('Pressure drop')
358     ax[2].set_title('Velocity and vorticity')
359     ax[2].set_xlabel('x')
360     ax[2].set_ylabel('y')
361     ax[3].set_title('Porosity')
362     ax[3].set_xlabel('x')
363     ax[3].set_ylabel('y')
364 
365     ux[bndry] = 0
366     uy[bndry] = 0
367 
368     # Plot velocity profile
369     varx = np.arange(nx-2) +1
370     vary = np.arange(ny-2) +1
371     ax[0].plot(Y[vary,varx],np.sqrt(ux[vary,varx]**2+uy[vary,varx]**2),
                                    '-o',color='red', label='middle')
372     ax[0].plot(Y[:,0],ux[:,0],'-o',label='inlet')
373     ax[0].plot(X[0,:],-uy[0,:],'-o',label='outlet')
374     ax[0].set_xlabel("x/y-coordinate")
375     ax[0].grid()
376     ax[0].legend()
377 
378     # Plot pressure drop
379     x = [0,nx/2,nx]
380     p = [np.sum(rho[vbndry]) * (cs**2), np.trace(rho[int(ny*0.4):int(ny
                                    *0.6),int(nx*0.4):int(nx*0.6)]) * (
                                    cs**2), np.sum(rho[pbndry]) * (cs
                                    **2)]
381     ax[1].plot(x,p,'-o', color='green')
382     ax[1].set_xticks(x)
383     ax[1].set_xticklabels(['inlet', 'middle', 'outlet'])
384     ax[1].grid()
385 
386     # Plot velocity and vorticity
```

```
    q = ax[2].quiver(X[bndry==False], Y[bndry==False], ux[bndry==False
                                    ], uy[bndry==False], np.hypot(ux[
                                    bndry==False], uy[bndry==False]) ,
                                    pivot='mid')
    vorticity = (np.roll(ux, -1, axis=0) - np.roll(ux, 1, axis=0)) - (
                                    np.roll(uy, -1, axis=1) - np.roll(
                                    uy, 1, axis=1))
    vorticity[bndry] = np.nan
    cmap = copy.copy(mpl.cm.get_cmap("bwr"))
    cmap.set_bad('black')
    ax[2].imshow(vorticity, cmap=cmap)
    ax[2].invert_yaxis()
    ax[2].set_aspect('equal')
    ax[2].set_xticks([])
    ax[2].set_yticks([])

    # Show porosity
    cmapPoro = copy.copy(mpl.cm.get_cmap('Greys'))
    cmapPoro.set_bad(color='black')
    norm = mpl.colors.Normalize(vmin=0, vmax=1)
    poro[bndry] = np.nan
    ax[3].imshow(1-poro, cmap=cmapPoro, norm=norm)
    ax[3].invert_yaxis()
    ax[3].set_aspect('equal')
    ax[3].set_xticks([])
    ax[3].set_yticks([])

    plt.show()
    return
```

Code Listing B.1: lbfuns

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
from lbfuns import *

'''
Lattice Boltzmann Method

            5   1   6
             \  |  /
              \ | /
    D2Q9    4 --0-- 2
              / | \
             /  |  \
            8   3   7
'''

n = 27 # Number of nodes on the x and y axis
nt = 100000 # Max number of time steps
plot = True # Enable or disable plotting during the simulation
uxInMax = 0.025 # Velocity boundary condition; max velocity of the parabolic profile at the inlet
pOut = 0.33 # Pressure boundary condition; pressure at the outlet

# initial guess for the porosity (beta * tau should be in between 0 to 1)
beta0 = 0.45 # 0: fluid; 1/tau: porous
beta1 = 0
beta = np.full((n,n),beta0)
#beta[np.arange(n-1),np.arange(n-2,-1,-1)] = 0.1
beta[np.arange(n-2),np.arange(n-3,-1,-1)] = beta1
beta[np.arange(n-3),np.arange(n-4,-1,-1)] = beta1
beta[np.arange(n-4),np.arange(n-5,-1,-1)] = beta1
beta[np.arange(n-5),np.arange(n-6,-1,-1)] = beta1
beta[np.arange(n-6),np.arange(n-7,-1,-1)] = beta1
beta[np.arange(n-7),np.arange(n-8,-1,-1)] = beta1
beta[np.arange(n-8),np.arange(n-9,-1,-1)] = beta1
```

```
36
37 v = 0.4 # viscosity
38 k = 2.5 # factor for polynomial scaling of the porosity
39
40 poro = check(v,beta,k)
41 print(poro) # 1: fluid; 0: porous
42
43 f, rho, ux, uy = solveTestCase(n,nt,plot,uxInMax,pOut,beta,v,k)
44
45 # Save results
46 dfrho = pd.DataFrame(rho)
47 dfrho.to_csv('./rho.csv', index=False)
48 dfux = pd.DataFrame(ux)
49 dfux.to_csv('./ux.csv', index=False)
50 dfuy = pd.DataFrame(uy)
51 dfuy.to_csv('./uy.csv', index=False)
```

Code Listing B.2: myLBM

B.2 Adjoint Method

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.integrate as integrate
4 import pandas as pd
5
6 def pTarget(x):
7     return  -x + 1 #1.8*x**2 + 1.2 #1-x**4
8
9 def area(x,kappa):
10     ''' Calculate the area of the tube
11     x:      x-coordinates
12     kappa:  design variabels
```

```
13      '''
14      return kappa[0]*(-x**3+3*x**2-3*x+1) + kappa[1]*(3*x**3-6*x**2+3*x)
                                                + kappa[2]*(-3*x**3+3*x**2) +
                                               kappa[3] *x**3
15
16  def gradient(var,dx):
17      ''' Calculate the change per step dx of the variable var(x) '''
18      grad = np.zeros(len(var))
19      grad[1:-1] = (var[2:] - var[:-2])/(2*dx)
20      grad[0] = (var[1] - var[0])/dx
21      grad[-1] = (var[-1] - var[-2])/dx
22      return grad
23
24  def properties(a,u0,p0,dx,n):
25      ''' Calculate the domain properties
26      a:  area of the tube
27      u0: velocity at the boundary/entrance of the tube
28      p0: pressure at the boundary/exit of the tube
29      dx: distance between the grid points
30      n:  number of grid points
31      u:  velocity along the tube
32      p:  pressure along the tube
33      '''
34      # Calculate velocity; Continuity equation: A1*u1=A2*u2
35      c = u0*a[0]
36      u = c/a
37      # Calculate pressure; Momentum conservation
38      dudx = gradient(u,dx)
39      p = np.zeros(n)
40      p[-1] = p0
41      for i in range(n-2,-1,-1):
42          p[i] = p[i+1] + u[i]*dudx[i]*dx
43      return u,p
44
```

```
45 def lm(a,u,dp,dx,n):
46     ''' Calculate Lagrange multipliers
47     a:  area of the tube
48     u:  velocity along the tube
49     dp: difference between the actual pressure and the target pressure
                                         along the tube
50     dx: distance between the grid points
51     n:  number of grid points
52     ua: adjoint velocity along the tube
53     pa: adjoint pressure along the tube
54     '''
55     # Calculate adjoint velocity by adjoint equation
56     ua = np.zeros(n)
57     for i in range(1,n):
58         ua[i] = ua[i-1]+dp[i]*dx
59     # Calculate adjoint pressure by adjoint equation
60     duadx = gradient(ua, dx)
61     pa = np.zeros(n)
62     pa[-1] = - (u[-1]*ua[-1])/a[-1]
63     for i in range(n-2,-1,-1):
64         pa[i] = pa[i+1] + u[i]/a[i] * duadx[i] * dx
65     return ua,pa
66
67 def sensitivities(x,u,pa,dx,kappa):
68     ''' Calculate the sensitivities '''
69     dpadx = gradient(pa,dx)
70     # Solve sensitivity equations
71     dFdkappa = np.zeros(len(kappa))
72     dFdkappa[0] = - np.trapz(u*dpadx*(-x**3+3*x**2-3*x+1),x,dx) - pa
                                         [0]*u[0]
73     dFdkappa[1] = - np.trapz(u*dpadx*(3*x**3-6*x**2+3*x),x,dx)
74     dFdkappa[2] = - np.trapz(u*dpadx*(-3*x**3+3*x**2),x,dx)
75     dFdkappa[3] = - np.trapz(u*dpadx*x**3,x,dx) + pa[-1]*u[-1]
76     return dFdkappa
```

```
77
78 def plotInit():
79     ''' Initialize figure and axes '''
80     f = plt.figure(figsize=(10,4))
81     ax = [f.add_subplot(141),f.add_subplot(142),f.add_subplot(143),f.
                                          add_subplot(144)]
82     ax[0].set_title('Shape of the Tube')
83     ax[0].set_xlabel('x')
84     ax[1].set_title('Pressure')
85     ax[1].set_xlabel('x')
86     ax[2].set_title('Velocity')
87     ax[2].set_xlabel('x')
88     ax[3].set_title('Objective')
89     ax[3].set_xlabel('Iteration')
90     return f, ax
91
92 def plot(ax,a,p,u,x,it,c):
93     ''' Plot current parameters during optimization '''
94     r = a/np.pi**2
95     ax[0].plot(x,r,color=c,alpha=0.1)
96     ax[0].plot(x,-r,color=c,alpha=0.1)
97     ax[1].plot(x,p,color=c,alpha=0.1)
98     ax[2].plot(x,u,color=c,alpha=0.1)
99     return
100
101 def plotLegend(ax,a,p,u,x,it,c,label):
102     ''' Plot current parameters with label to be shown in the legend
                                          '''
103     r = a/np.pi**2
104     ax[0].plot(x,r,color=c)
105     ax[0].plot(x,-r,color=c)
106     pl = ax[1].plot(x,p,label=label,color=c)
107     ax[2].plot(x,u,color=c) #label=f'{it}'
108     return pl
```

Code Listing B.3: adjointfuns

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import cm
import scipy.integrate as integrate
import pandas as pd
from adjointfuns import *


# Define Grid
n = 50 # number of grid points
x = np.linspace(0,1,n)
dx = 1/n # distance between grid points

# Boundary conditions
p0 = 0 # pressure at x=1
u0 = 1 # # Flow velocity at the entrance of the tube (x=0)

# Optimization parameters
eta = 0.02 # Update
maxIt = 2000 # Max iterations
conv = 0.01 # Convergence criterion
# Initial guess for design variables
kappa = [0.6,0.6,1,0.5]

# Store optimization steps
results = []
# Plot results
f, ax = plotInit()
itPlot = 200000
color = iter(cm.rainbow(np.linspace(0,1,int(maxIt/itPlot))))

```

```
32 for it in range(maxIt):
33     # Calculate domain properties
34     a = area(x,kappa)
35     u,p = properties(a,u0,p0,dx,n)
36 
37     # Plot starting properties
38     if it == 0:
39         p1, = plotLegend(ax,a,p,u,x,it,'g','Start')
40 
41     # Calculate objective
42     pt = pTarget(x)
43     dp = p-pt
44     F = 0.5*np.sum(dp**2)
45 
46     # Check for convergence
47     if F < conv:
48         break
49 
50     # Calculate Lagrange multipliers
51     ua, pa = lm(a,u,dp,dx,n)
52 
53     # Calculate sensitivities of the design variables
54     dFdkappa = sensitivities(x,u,pa,dx,kappa)
55     # Update the design variables
56     kappa -= eta*dFdkappa
57 
58     # Store results
59     results.append([it,F,kappa[0],kappa[1],kappa[2],kappa[3]])
60 
61     if (it % itPlot == 0 and it != 0):
62         c=next(color)
63         plot(ax,a,p,u,x,it,c)
64 
65 # Visualize results
```

```
66 tbl = pd.DataFrame(results, columns=['it', 'F', 'kappa1', 'kappa2', '
                                        kappa3', 'kappa4'])
67 tbl = tbl.set_index('it')
68 print(tbl)
69 p2, = plotLegend(ax,a,p,u,x,it,'r','Result')
70 p3, = ax[1].plot(x,pt,label='Target',c='black')
71 plt.legend(handles=[p1,p2,p3], title='Legend', bbox_to_anchor=(-1.25,
                                        1), loc='upper left')
72 ax[3].plot(tbl['F'])
73 plt.show()
```

Code Listing B.4: myAdjoint