NTNU

Norwegian University of
Science and Technology

Numerical Simulations of Flow Around a Bluff Body, Using Multigrid and an Immersed Boundary Method

Marie Flø Aarsnes

Master of Science in Engineering and ICT
Submission date: June 2017
Supervisor: Håvard Holm, IMT

Norwegian University of Science and Technology
Department of Marine Technology

NTNU Trondheim
Norwegian University of Science and Technology
Department of Marine Technology

MASTER THESIS IN MARINE TECHNOLOGY

SPRING 20017

FOR

STUD. TECHN.

Numerical simulations of flow around a bluff body, using multigrid and a immersed boundary method.

In the thesis the candidate shall present his personal contribution to the resolution of problem within the scope of the thesis work.

A simplified CFD solver should be developed, based on the SOLA code. The candidate shall familiarize herself with theory and numerical approximations in the code.

The candidate shall pay special attention to calculation efficiency. It will be important that the code runs fast for simulations of large models with many grid points. Different alternatives for speeding up the code should be considered.

If time permits, the candidate should also implement an immersed boundary method. This method should be used for making the code calculate flow around complex geometries. Special care should be given to how the force on the body is implemented.

As a test case for the code, the student shall calculate flow around a circular cylinder.

Theories and conclusions should be based on mathematical derivations and/or logic reasoning identifying the various steps in the deduction.

The candidate should utilize the existing possibilities for obtaining relevant literature.

The thesis should be organized in a rational manner to give a clear exposition of results, assessments, and conclusions. The text should be brief and to the point, with a clear language. Telegraphic language should be avoided.

The thesis shall contain the following elements: A text defining the scope, preface, list of contents, summary, main body of thesis, conclusions with recommendations for further work, list of symbols and acronyms, reference and (optional) appendices. All figures, tables and equations shall be numerated.

The supervisor may require that the candidate, in an early stage of the work, present a written plan for the completion of the work. The plan should include a budget for the use of computer and laboratory resources that will be charged to the department. Overruns shall be reported to the supervisor.

NTNU Trondheim
Norwegian University of Science and Technology
Department of Marine Technology

The original contribution of the candidate and material taken from other sources shall be clearly defined. Work from other sources shall be properly referenced using an acknowledged referencing system.

The thesis shall be submitted in two copies:

- Signed by the candidate
- The text defining the scope included
- In bound volume(s)
- Drawings and/or computer prints that cannot be bound should be organized in a separate folder.

Supervisor : Associate Professor Håvard Holm
Start : 15.01.2017
Deadline : 18.06.2017

Trondheim, date 07/6-2017

Preface

This is a master thesis on CFD, which is my last contribution to my master degree at Norwegian University of Science and Technology (NTNU) as a part of the study program engineering and ICT. The work was carried out during the spring 2017 and is an extension of my project thesis "Navier Stokes Solver".

I would first like to thank my supervisor Håvard Holm for the time, valuable input and support during this semester. Furthremore I would like to thank Jon Coll Mossige for sharing his knowledge and work on the immersed boundary method.

Finally I would like to thank my family and friends for being helpful and supportive during all my five years at NTNU.

Summary

Increase in computational efficiency is one of the most prominent factors for successful applications of large CFD models with many grid points. Consequences of the increase are reduction in CPU-time and memory usage, so better mesh refinement could be used. Hence, the CFD-solver will improve the resolution in the computational domain.

This report provides an introduction to the background theory in CFD with basis on methods used to develop a simple incompressible Navier Stokes solver. Different multigrid algorithms and an immersed boundary method are discussed, where the solver exploit the cause of increased computational efficiency, especially by using multigrid. Validation is done based on efficiency and accuracy for the multigrid algorithms compared to Gauss Seidel method and SOR method. Force calculations around a submerged bluff cylinder in a 2D flow are used to validate the immersed boundary method implemented in the solver together with a full multigrid algorithm.

By implementing a full multigrid method, the time used to solve the Poisson equation was reduced significantly and the accuracy of the resolution is kept. The validation tests of a solver combining an immersed boundary method and the full multigrid algorithm was successfully carried out except for too low coefficients at Re = 100 in the final test case. The accuracy of flow resolution was specially affected by the time refinement and the width of the computational domain.

Sammendrag

Økning i beregnings effektivitet er en av hoved faktorene for å oppnå suksessfulle CFD applikasjoner for store modeller med mange grid-punkter. Konsekvensene av økningen er reduksjon i CPU-tid og minne bruk, så finere grid kan bli brukt og CFD-løseren kan dermed oppnå enda bedre numeriske resultater i domenet.

Denne rapporten gir en introduksjon til bakgrunnsteori til CFD basert på de metoder som blir brukt til å utvikle en enkel inkompressibel Navier Stokes løser. Forskjellige multigrid algoritmer og en immersed boundary methode er diskutert, hvor løseren utnytter den resulterende økningen av beregnings effektivitet, særlig på vegne av multigird metoden. Valideringen er basert på effektivitet og nøyaktighet for multigrid algoritmene sammenlignet med Gauss Seidel metoden og SOR metoden. Krefter rundt en nedsenket fast sylinder i en 2D strøm er beregnet og brukt til å validere immersed boundary metoden som er implementert i løseren sammen med en full multigrid metode.

Ved å implementere en full multigrid metode, er tiden brukt på å løse Poissons likning redusert betraktelig samtidig som nøyaktigheten i resultatet er beholdt. Gjennomføringen av validerings testene for en løser som kombinerer en immersed boundary methode og en full multigrid methode var vellykket, bortsett fra for lave koeffisienter i siste test ved Re = 100. Nøyaktigheten i strømningsfeltet var særlig påvirket av valget av tids-steg og bredden på domenet som ble brukt til simuleringene.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

To get essential engineering data for complex problems, *Computational Fluid Dynamics* (CFD) is a useful tool to provide data in an inexpensive way. By numerical simulations, a CFD-solver studies the physical laws under viscous conditions by solving the governing equations numerically. At a current state, it has been developed numbers of CFD-solvers. Some are commercial, e.g. Fluent, and other are open source, e.g. OpenFoam. The solvers are build up by using different approaches based on old methods. The development of the solvers are not yet at a level where the user can work with the tool uncritically, so knowledge of numerics involved are essential.

CFD is a powerful tool and can be used to visualize complex problems outside our range of vision, like Peskin (1972) used CFD with an *Immersed Boundary Method* (IBM) to simulate blood-flow trough a human heart valve, figure 1.1. Different areas of the solution domain can be observed without disturbing the flow field around. Perhaps the simulations simulate a flow around an complex rigid boundaries like Lai and Peskin (2000) presented in their studies. The

results may supplement experimental tests as a qualitative tool to decide e.g. design before the tests are preformed.

Because of computational limitations, CFD applications are not at a level where it can be used for real time computations. Increase in computational power and efficiency is one of the most prominent factors. When focusing on efficiency in a Navier Stokes solver, the Poisson solver should be evaluated. The process solving the Poisson equation usually takes the vast majority of the runtime. To affect the overall performance, an efficient method should be used on the Poisson solver. At a current state, one of the most efficient methods is still the *multigrid method* introduced by (Brandt, 1977).

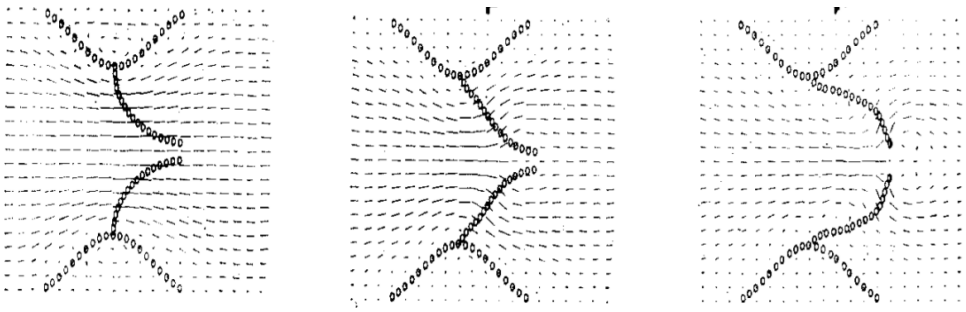

Figure 1.1: Blodflow trough a human heart valve (Peskin, 1972).

1.2 Objectives

The main objective of this thesis is to develop a *CFD-solver* with special attention paid to calculation efficiency, so the code runs fast for simulations of large models with many grid points. As a test case, flow around a cylinder is tested with spacial attention on the forces. Therefore, a method for simulations of *fluid structure interactions* (FSI) is developed. The main objective is future divided into:

1. Modeling the Navier Stokes equation in a proper way based on old methods.
2. Develop *multigrid methods* to reach convergence faster than iterative schemes.
3. Implement *immersed boundary method* (IBM) for simulations of fluid-structure problem.

1.3 Approach

The basis of the Navier Stokes solver is formed by "SOLA - Solution Algorithm for 2D incompressible laminar transient flow", an excerpt from "A Navier Stokes Solver using the Multigrid Method" by Reidar Kristoffersen and based on Hirt et al. (1975). My project thesis "Navier Stokes Solver" (Aarsnes) consists of a preliminary layout and development of the solver, and this master thesis is an extension for that.

The following limitations are used in the solver:

1 Constant density, ρ = constant

2 Constant viscosity, ν = constant

3 Newtonian fluid

4 Laminar flow

5 Cartesian coordinates, $(x_i) = (x, y)^T$ and $(u_i) = (u, v)^T$

Hence, the governing equations to be solved simplifies to a simple two-dimensional incompressible Navier Stokes system. The temporal-derivatives are discretized by a forward Euler scheme and the spatial-derivatives are discretized by a second order central differences scheme on an equidistant staggered grid. This is known as the FTCS-scheme (Forward-Time Central-Space). For the iterative-scheme used to solve the Poisson equation, Gauss Seidel, successive over-relaxation (SOR) or different multigrid methods may be chosen. In a test case with flow around a cylinder, the FSI problem is solved by IBM.

1.4 Outline

Chapter 2 provides an overview of the fundamental theory in a CFD-solver. The simplified governing equations, discretization methods to solve the derivatives, different way of structuring the computational nodes in the domain and some numerical methods are presented. In Chapter 3, the efficient multigrid method is described in three different algorithms. Chapter 4 presents how to solve the FSI problem with IBM. The code layout and some comments to the development are given in Chapter 5. Validation of the code and a test case of flow around a cylinder is presenter in Chapter 6, and concluded in Chapter 7 together with some recommendations for future work.

Chapter 2

Fundamental Theory

In this chapter some fundamental theory used for developing a Navier-Stokes solver are introduced. This is an extension from my project thesis "Navier Stokes Solver" (Aarsnes), where the relevant parts from the project are included here.

2.1 Navier Stokes Equation

For many viscous flow problems, it is not possible to get an accurate solution using simple equations. The solutions of the Navier-Stokes equations can display very fine details of the flow structure, such as strong viscous-inviscid interactions with large separated flow regions. The unsteady compressible Navier-Stokes equations are a mixed set of hyperbolic-parabolic equations in time, and the incompressible Navier-Stokes equations are a mixed set of elliptic-parabolic equations. Thus, the inviscid and viscous equations require significantly different solution strategies.

2.1.1 Incompressible Navier Stokes equation

When heat transfer or significant property variations are not present, the incompressible Navier-Stokes equation is the best choice. The governing equations for the two-dimensional incompressible Navier-Stokes system with a constant property flow and without body forces or external heat can be written in conservative form:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) \tag{2.1}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}) \tag{2.2}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{2.3}$$

Here, u and v represents the velocity components in x- and y-direction, respectively, p is the pressure, ρ is the density and ν is the kinematic viscosity. Equation 2.1 is the *x-momentum equation* and 2.2 is the *y-momentum equation*, which are parabolic PDEs. Equation 2.3 is the *continuity equation*, which is an elliptic PDE. In the momentum equations there is a *convective term*, $\mathbf{u}\nabla\mathbf{u}$, and a *diffusive term*, $\nu\nabla^2\mathbf{u}$, hence the Navier-Stokes equation belongs to the class of convection-diffusion equations.

2.1.2 Dimensionless approach of variables

A dimensionless approach generalize the problem and reduces the number of parameters in the equation. This is possible because it is easier to see how to treat the parameters, i.e. possibilities for neglecting or approximating them. The purpose of making Navier Stokes equation dimensionless is to reduce the number of times the equation needs to be solved numerically.

Table 2.1: Non-dimensional parameters (Richard H. Pletcher, 2013).

Name	Reference parameter	Non-dimensional parameter
Velocity	U	$\mathbf{u}^* = \frac{\mathbf{u}}{U}$
Length	L	$\mathbf{r}^* = \frac{\mathbf{r}}{L}$
Time	-	$t^* = \frac{tU}{L}$
Pressure	-	$p^* = \frac{p}{\rho U^2}$
Reynolds number	-	$Re = \frac{UL}{\nu}$

For the case of flow without heat transfer, the non-dimensional equations only depend on the Reynolds number, which is a ratio between the viscous and the advective term. This means that the non-dimensional parameters will have the same value at same Reynolds number. The non-dimensional primitive-variables for a Cartesian coordinate system are introduced in table 2.1.

By substituting the non-dimensional parameters from table 2.1 into the Navier Stokes equations 2.1, 2.2 and 2.3, the non-dimensional x-momentum equation 2.5, y-momentum equation 2.6 and continuity equation 2.4 are obtained.

$$\frac{\partial u^*}{\partial x^*} + \frac{\partial v^*}{\partial y^*} = 0 \tag{2.4}$$

$$\frac{\partial u^*}{\partial t^*} + u^* \frac{\partial u^*}{\partial x^*} + v^* \frac{\partial u^*}{\partial y^*} = -\frac{\partial p^*}{\partial x^*} + \frac{1}{Re}\left(\frac{\partial^2 u^*}{\partial x^{*2}} + \frac{\partial^2 u^*}{\partial y^{*2}}\right) \tag{2.5}$$

$$\frac{\partial v^*}{\partial t^*} + u^* \frac{\partial v^*}{\partial x^*} + v^* \frac{\partial v^*}{\partial y^*} = -\frac{\partial p^*}{\partial y^*} + \frac{1}{Re}\left(\frac{\partial^2 v^*}{\partial x^{*2}} + \frac{\partial^2 v^*}{\partial y^{*2}}\right) \tag{2.6}$$

2.1.3 The projection method

For an incompressible fluid, the pressure and the velocity are independent of time.

The projection method, also called the method of fractional steps, is a procedure used to decouple the pressure and velocity and was first presented by Chorin (1968) on a regular grid. The pressure gradient terms are omitted from the momentum equation (2.1 and 2.2) in the frist step and advanced in time. By using the Helmholtz decomposition, resolved into a divergence-free term and a curl-free term, equation 2.7 is preformed.

$$\frac{\mathbf{u}^{k+1}-\mathbf{u}^{*}}{\Delta t}+\nabla p^{k+1}=0 \tag{2.7}$$

Where $\frac{\mathbf{u}^{k+1}-\mathbf{u}^{*}}{\Delta t}$ has zero divergence and ∇p^{k+1} has zero curl. $\mathbf{u}^{*}$ denotes the predicted velocity, $\mathbf{u}^{k+1}$ is the velocity field at the new time-step and p^{k+1} is the new pressure. Now, there are two unknowns, $\mathbf{u}^{k+1}$ and p^{k+1}, that must be solved to find the solution. The continuity equation 2.3 can be differentiated to yield, thus the pressure field needs to satisfy the incompressibility constraint 2.8.

$$\nabla\cdot\mathbf{u}^{k+1}=0 \tag{2.8}$$

Hence, the Poisson equation 2.9 will be derived with a source term equal to the divergence of the predicted velocity field and the pressure difference as an potential function .

$$\nabla^{2}p^{k+1}=\frac{1}{\Delta t}\nabla\mathbf{u}^{*} \tag{2.9}$$

Then follow this solution procedure:

1 Calculate the predicted velocity $\mathbf{u}^{*}$ from momentum equation while neglecting the pressure gradient terms.

2 Solve the Poisson equation 2.9 for the corrected pressure p^{n+1}.

3 Calculate the corrected velocity $\mathbf{u}^{n+1}$ from 2.7.

The projection method is possible to implement on both regular and staggered grids, where explicit and implicit methods are employed. Using an explicit-first-order Euler scheme for time-derivatives and staggered grid Peyret and Taylor demonstrated that the scheme is closely related to marker and cell (MAC) method by Harlow et al. (1965).

2.2 Grid

Grid, also called mesh, is a necessary tool when working with computational simulations. To approximate numerical solutions to a PDE, a discretization of the equation needs to be done to find a system of algebraic differential equations. Therefor the discretization is accomplished by placing discrete nodes over the solution field, where the discrete nodes can be connected in different ways to form discrete cells. The differential equations can be represented over discrete nodes or over discrete cells. This is just a short introduction to some types of grids, for future reading see Thompson et al. (1998).

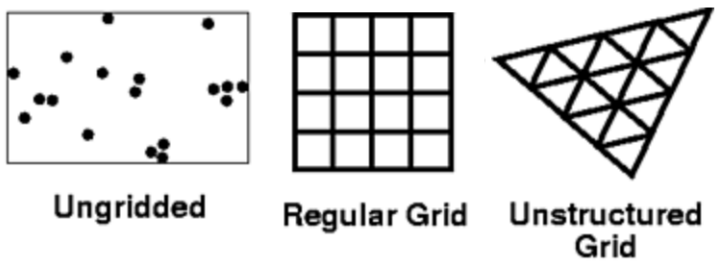

Figure 2.1: Different kinds of grids, a modification from Djeddi et al. (2013).

When coupling grid and solution processes together several choices based on underlying principles and mathematics can be made to optimize the process. For the finite difference method (FDM), the derivatives of field variables are easily expressed, thus structured grids are usually used. For the finite element method (FEM) and the finite volume method (FVM), unstructured grids are more convenient because of the flexibility these methods offers.

2.2.1 Staggered grid

Staggered grid is a structured grid, i.e. with uniform cell size, where the field variables are represented at different locations (Harlow et al., 1965). The pressure is represented at the center of the cell and the velocities at the cell edges in each direction, see figure 2.2. Staggered grids are more accurate than non-staggered (collocated) grids and can therefore use coarser grids, sometimes twice as coarse grid, to obtain the same accuracy (Mccormick, 1988). For pressure calculations, like the Poisson equation, staggered grids has an advantage because of a more direct coupling between velocity and pressure, thus avoiding unphysical pressure oscillations.

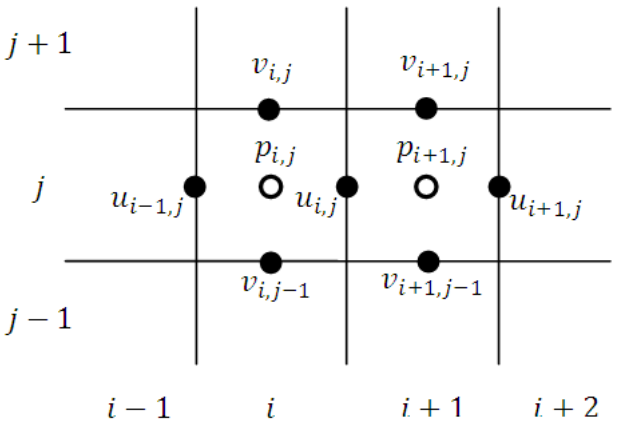

Figure 2.2: Staggered grid (Djeddi et al., 2013).

2.2.2 Adaptive grid

When using structured grid on a flow filed, unnecessary computational effort are done in some parts of the domain. Usually, a flow field is not uniform, hence different grid treatment should be used for different areas in the domain, see figure 2.3.

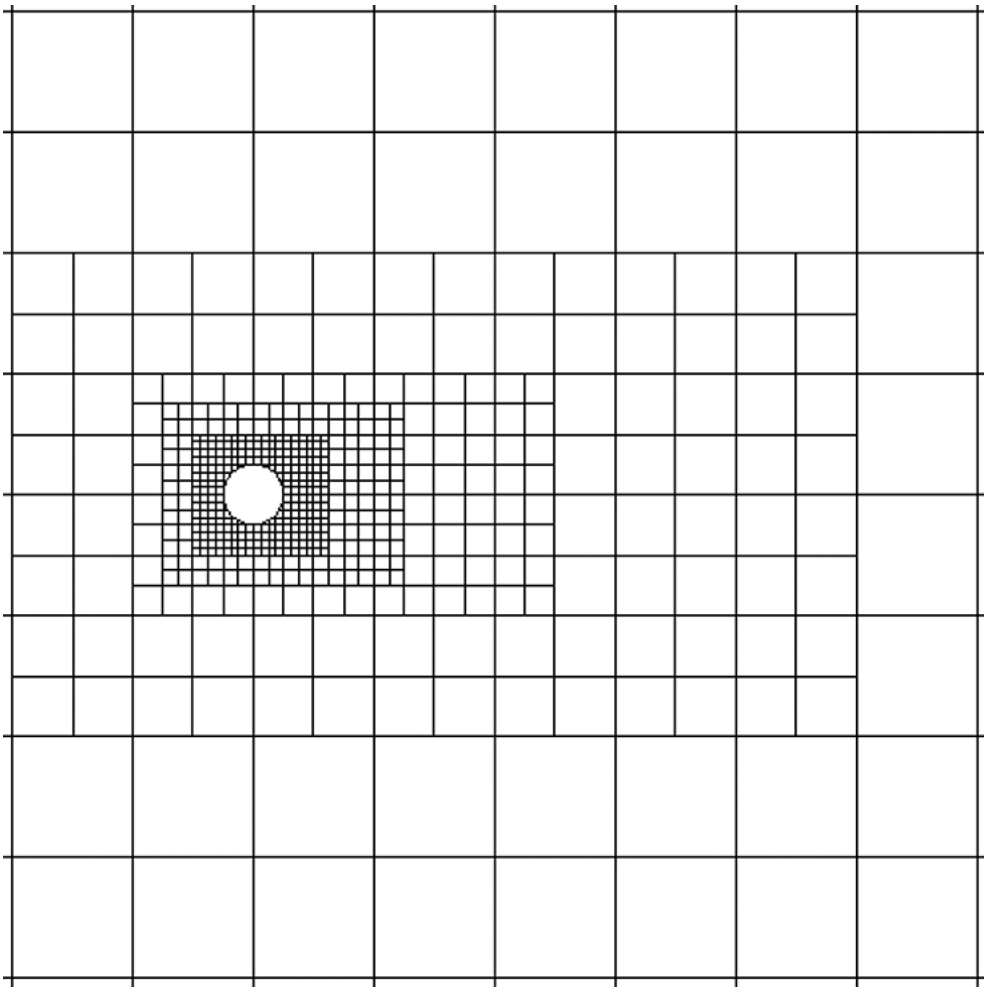

Figure 2.3: Adaptive grid around a cylinder (Vanella et al., 2014)

Adaptive grids follows gradients in the physical solution, hence the areas with higher gradients, f.ex high velocity and pressure gradients, are better represented with a refined grid due to the rest of the domain. High gradients usually occurs around boundary layers and flow fields past a body is a critical area where higher resolutions are need. Structured grid will in these cases need high resolution in the whole domain to handle the high gradients around the body, which is very inefficient.

2.3 Discretization

As mentioned on section 2.2, the discrete operators depends on the chosen grid, but also on a discretization method. A discretization method approximates the first and second derivatives in the Navier Stokes system, equations 2.5, 2.6 and 2.4. The momentum equations are re-ranged to make the discretization easier with the temporal derivatives on the left hand side and the spatial derivatives on the right hand side. see equation 2.10. i and j denotes x- and y- direction, respectively, for velocity and in space.

$$\frac{\partial u_i}{\partial t} = -\frac{\partial u_i u_j}{\partial x_j} - \frac{\partial p}{\partial x_i} + \frac{1}{Re}\frac{\partial^2 u_i}{\partial x_j} \tag{2.10}$$

2.3.1 Spatial

Spatial discretization methods are used to represent and evaluate PDE's. The most common ones are the FDM, the FVM and the FEM. This subsection will give a short introduction of each.

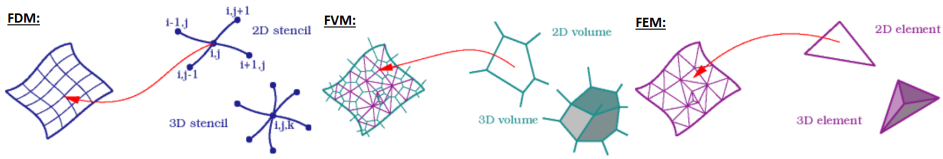

Figure 2.4: Primary behavior of FDM, FVM and FEM

Finite Difference Method

The FDM is working on stencils, see figure 2.4. Thus, the method is approximating the node derivatives for a problem and leads to sparse matrices. It is limited for use on uniform-structured grid and is therefore simple to implement and derive.

There are different approaches of FDM, which differ in how the stencil is built up. One approach is the central FDM, see figure 2.5, a five point stencil scheme which approximates the derivatives with the closest neighboring nodes.

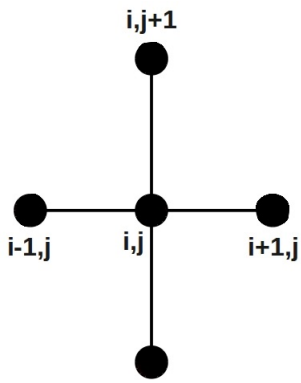

Figure 2.5: 5-point stencil for central FDM.

Finite Element Method

The FEM is working on elements, see figure 2.4. Thus, the method uses basis functions over a finite region which leads to large, banded matrices. The advantages of the method are that it can be used on unstructured grids and that it is general. Irregular geometries require more computer storage, making FEM more time consuming in these situations.

Finite Volume Method

The FVM is working with volumes, see figure 2.4. Thus, the method is approximating integrals to find the solution. Divergence terms are converted to surface integrals using the divergence theorem and evaluated as fluxes at the surface of the finite volumes. The advantages of the method are that it can be used on unstructured grids and that it is conservative.

2.3.2 Temporal

In time, it is useful to consider the discretization process in two stages (LeVeque, 1992):

1. Discretize the problem in space and leave it continuous in time. Then the PDE become a system of ordinary differential equations (ODE's), also called *semi discrete equations*. Then the right hand side can be solved as a *boundary value problem*.

2. Discretize in time using a numerical method that solves a system of ODE's.

Two different explicit numerical methods for solving the system of ODE's are described briefly.

Forward Euler

Explicit Euler scheme is named after and first introduced by Euler (1768), see equation 2.11, is first-order accurate in time, $\mathcal{O}(\Delta t)$. Hence, small time-step Δt is decreed because on the lack of stability and accuracy.

$$\frac{u^{n+1} - u^n}{\Delta t} = R(u^n) \tag{2.11}$$

$R(u^n)$ is the right hand side of the equation, n is the current time-step and $n+1$ is the new time-step.

Runge-Kutta

Solving the system of ODE's by a first-order method may entail high computational costs. By choosing a higher-order method like Runge-Kutta (Kutta, 1901) is recommended if high accuracy is desired. Runge-Kutta schemes can have different orders of accuracy depending on the order of the applied Runge-Kutta. For a second-order Runge-Kutta method, see equation 2.12, the order of accuracy is two, $\mathcal{O}(\Delta t^2)$.

$$u^{(1)} = u^n + \Delta t R(u^n) \qquad\qquad u^{n+1} = u^n + \frac{\Delta t}{2}\Big(R(u^n) + R(u^{(1)})\Big) \tag{2.12}$$

$R(u^n)$ is the right hand side of the forward Euler equation 2.11, n is the current time-step and $n+1$ is the new time-step.

2.4 Boundary Conditions

A *boundary value problem* is a differential equation together with some constraints, *boundary conditions* (BC). The technique used for implementing BC's have great influence on the stability and convergence of the numerical solution. For flow field problems, just a limited domain of the problem will be evaluated. Therefore, BC's need to be specified in a finite distance from e.g. the geometry in the flow. The two main BC's are:

- *Dirichlet boundary condition*, or fixed boundary condition, specifies the value at the boundary, $f(x)$, and can be expressed in a general form as

$$u(x) = f(x) \qquad \forall_x \in \partial\Omega, \quad \Omega \subset \mathbb{R}^n$$

- *Neumann boundary conditions* specifies what the value of derivatives over the boundary, $f(x)$, is and can be expressed in a general form as

$$\frac{\partial u(x)}{\partial \mathbf{n}} = f(x) \qquad \forall_x \in \partial\Omega, \quad \Omega \subset \mathbb{R}^n$$

  Thus, the boundary needs to be sufficiently smooth so a derivative in the same direction exist. Neumann boundary conditions are not well defined at corners.

Where $\mathbf{n}$ denotes the normal vector to the boundary $\partial\Omega$ of the domain Ω and u is the velocity of the fluid.

According to way of representing the calculation values on a staggered grid, ghost-cells outside of the domain (i.e. exterior cells) are being used to specify BC's. For a geometry, the cell right inside of the geometry-wall can be seen as ghost-cells. BC's in a staggered grid are located at the boundary of a calculation

cell. Therefore a BC at a vertical cell-wall will pass through a horizontal-velocity node and a horizontal-wall will pass trough a vertical-velocity node. Some BC's for a 2D staggered grid are future discussed.

A rigid impermeable wall may be either *no-slip* or *free-slip*, considered as a plane of symmetry (Harlow et al., 1965). The velocity nodes on the wall will vanish at all times for either wall types. Suppose a impermeable wall along the *y-axis*, see figure 2.6, then the velocities will have the constraints:

- *No-slip:* $u' = 0$ and $v' = -v$
- *Free-slip:* $u' = 0$ and $v' = v$

Where u' and v' are velocities in the ghost-cells and u and v are the velocities in the fluid. Analogous boundary conditions are applied on a wall along the *x-axis*. In general, there should be no change in pressure over a boundary if there are no body-forces, hence $p' = p$. p' is the pressure in the ghost-cell and p is the pressure on the fluid side.

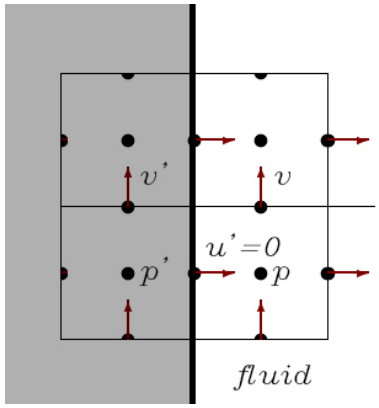

Figure 2.6: Boundary conditions on a vertical wall

Velocity and pressure needs to have at least one Dirichlet condition, i.e. be stated at least once in the domain, otherwise the numerical system could turn out unsolvable.

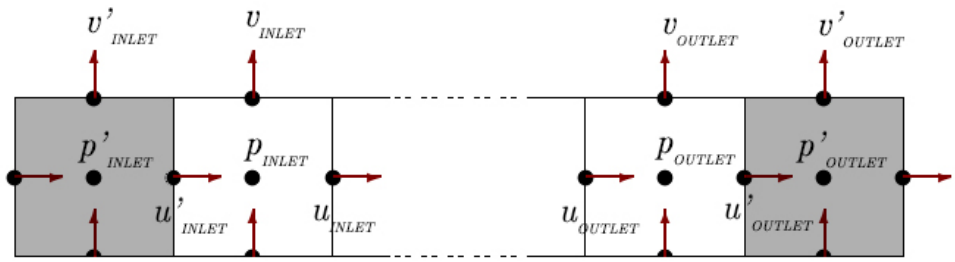

Figure 2.7: How to apply periodic boundary condition

To apply *periodic* boundary conditions (Patankar et al., 1977) an extra source term need to be attached to the momentum equation for the stream direction. Periodic conditions are pressure driven, so the source term will be equivalent to the pressure drop per unit periodic length. The boundary conditions for velocity and pressure at inlet cells are equal to the ghost-cells at outlet face and vice versa for the outlet cells:

- Inlet: $u_{inlet} = u'_{outlet}$ $v_{inlet} = v'_{outlet}$ $p_{inlet} = p'_{outlet}$
- Outlet: $u_{outlet} = u'_{inlet}$ $v_{outlet} = v'_{inlet}$ $p_{outlet} = p'_{inlet}$

A visualization is given in figure 2.7.

2.5 Methods to Solve Linear Systems

The Poisson equation 2.9 is a linear system of equations and can be written as $A\mathbf{u} = \mathbf{b}$. A is the coefficient matrix, $\mathbf{u}$ is a vector of unknowns and $\mathbf{b}$ is the source term. To solve it, an iterative or a direct scheme is used.

Direct schemes, e.g. Gaussian elimination, is solving the systems of equations exactly. Direct schemes are efficient on systems with a limited number

of unknowns. However, when the number of unknowns increase the computational work increases considerably too and a iterative scheme should be considered.

Iterative methods is based on relaxing the solution at each node by starting with an initial guess and modifying it until a limit of convergence is reached. Each modifying is an iteration called relaxing sweep and the final solution would become an approximation of the exact solution. Gauss Seidel and SOR are two iterative schemes that may be used. Gauss Seidel is a simple method which requires only one storage vector, hence it is easy to implement. SOR method may be a refinement of the Gauss Seidel method but provides faster convergence by choosing a optimal damping coefficient ω_{opt}. As Kahan (1958) presented $\omega \in (0,2)$ for SOR to converge, however ω_{opt} may be hard to find. For $\omega = 1$, SOR is the same as Gauss Seidel method.

In chapter 3 a more complex method for solving the Poisson equation is introduced, the *multigrid method*. Equations for implementation of Gauss Seidel and SOR on a 2D central FDM scheme are prescribed in section 6.1 together with a limit for convergence.

2.6 Stability Analysis

Stability criteria are taken into account to ensure that there will be no numerical uncertainties in the simulation. By *Lax's equivalence theorem* provided by Richtmyer and Morton (1967) this is true if the scheme is consistent.

Lax's equivalence theorem: Given a properly posed initial value problem and a finite-difference approximation that satisfies the consistency condition, stability is the necessary and sufficient condition for convergence.

The Navier Stokes equation is a complex non-linear PDE, therefore it is hard to apply von Neumann's analysis. The normal procedure is to linearize the equation and investigate one term at a time. In subsection 2.1, the different terms are pointed out for the Navier -Stokes equation.

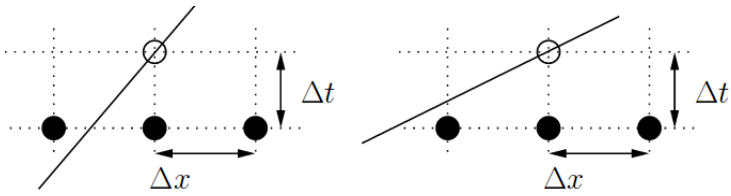

Figure 2.8: Stability, CFL-condition. Left figure: stable. Right figure: unstable.

From the *convection* term, the *Courant-Friedrichs-Lewy condition* (*CFL-condition*) 2.13 is derived, see figure 2.8. The non-dimensional number C is called the *Courant number*, which represent how many cell-sizes Δx_i a fluid particle passes trough a time-step Δt with velocity u_{x_i}. The condition must be consistent in all dimensions and at the whole domain at each time-step.

$$C_i = \frac{u_{x_i}\Delta t}{\Delta x_i} \leq C_{max}, \qquad i = 1,2,3 \tag{2.13}$$

C_{max} is changing with the discretization method. Deciding whether the time-discretization method are implicit or explicit has a great impact. For explicit methods C_{max} is usually equals to one. Implicit methods are usually less sensitive for numerical instability, so C_{max} can be larger.

From the *diffusion* term, a stability criteria to ensure that the viscous diffusion of momentum is limited to one cell per time-step, see equation 2.14, is derived.

$$\Delta t \leq \frac{Re}{2}\left(\frac{\Delta x^2 \cdot \Delta y^2}{\Delta x^2 + \Delta y^2}\right) \tag{2.14}$$

The last stability criteria ensures that the stability is sustained for the computation. The equation 2.15 provides positive diffusion by stabilizing the numerically unstable convection term. The condition must be consistent in all dimensions and on the whole domain at each time-step, since it build on the *CFL-condition*.

$$\Delta t \leq Min\Big(\frac{2}{Re(u_{i,j}^n)^2}, \frac{2}{Re(v_{i,j}^n)^2}\Big) \tag{2.15}$$

Equations 2.14 and 2.15 are here given for a 2D problem and based on $C_{max} = 1$, i.e. adjusted for explicit methods. For other C_{max} they will change accordingly.

Stability of Explicit Euler

Linear stability of explicit Euler is found from $|1 + h\lambda| \leq 1$, where λ is the eigenvalue of the linear test equation. The stability domain is given in figure 2.9, where the axes are the real and imaginary of $h\lambda$. The explicit Euler method demands small time-steps if the solution is fast decaying or have highly oscillating modes.

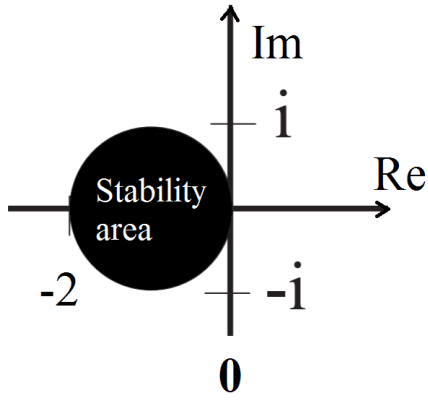

Figure 2.9: The stability area for a explicit Euler scheme, the axis are scaled for $h\lambda$.

Chapter 3

Multigrid

The multigrid method for solving elliptic PDE introduced by Brandt (1977) accelerate the convergence of an iterative method, like Jacobi method or Gauss Seidel method, from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, when the PDE is discretized on n grid points. This is done by combining coarse and fine grids, where the error is transferred from fine to coarser grids. By analyzing the error or difference in the iterative scheme with Fourier analysis, it can be seen that the error components have different frequencies. Iterative methods like Jacobi and Gauss Seidel are efficient for reducing high frequencies, because the new solution value is dependent on the previous one. Therefore, in a multigrid algorithm the coarse grid is used to reduce the low frequency components (which becomes high frequency components on coarse grids) of the error and the fine grid improves the accuracy by reducing the high frequencies. Hence, the coarse grids are only used to obtain correction, never to seek solutions of the original problem.

In a multigrid method, several grids which are typically increased by a factor of 2 can be used. Set the chosen difference representation on the left side of the equation and the discretized iterative scheme on the right side, then equation 3.1 represents the linear system to be solved.

$$A^h \mathbf{u}^h = \mathbf{f}^h \tag{3.1}$$

Where h is the grid spacing on the corresponding equidistant grid. See subsection 2.5 for an introduction on methods to solve the linear system of equations. The method could be applied to the linear system directly or to the error equation, which is called *the residual equation.*

The residual equation

Let $\mathbf{u}^{*(k)}$ be an approximation of $\mathbf{u}$ at iteration k, then the error $\mathbf{e}^{(k)}$ = $\mathbf{u}$ - $\mathbf{u}^{*(k)}$ satisfies the residual equation 3.2.

$$A\mathbf{e}^{(k)} = \mathbf{f} - A\mathbf{u}^{*(k)} =: \mathbf{r}^{(k)} \tag{3.2}$$

In multigrid methods, the residual equation is used to update the current $\mathbf{u}^{(k)}$. An approximation of $\mathbf{e}^{(k)}$, $\mathbf{e}^{*(k)}$, is computed and used to find the new iteration $\mathbf{u}^{(k+1)}$ = $\mathbf{u}^{(k)}$ + $\mathbf{e}^{*(k)}$. Close to the solution the error $\mathbf{e}^{(k)}$ is small, therefore zero is a good initial guess for 3.2.

There are two ways of constructing multigrid methods, the standard *geometric multigrid* (GMG) and the *algebraic multigrid* (AMG). GMG, which is the multigrid design used in this chapter, have some limitations regarding hierarchy of grids. Sometimes the hierarchy dependency on the underlying geometry is hard to handle, then AMG can be used. In contrast to GMG, AMG do not exploit the underlying geometry and work directly on the linear system. For future reading on AMG, see Stüben (1983) and Xu and Zikatanov (2017).

3.1 Grid Transfer

When grids are transferred, they are either *restricted*, i.e. transfer error from fine to coarse grid, or *prolongated* (also called *interpolation*), i.e. transfer initial solution from coarse to fine grid.

When transposing from a fine to a coarser grid, the error (residual) is transfered from Ω^h to Ω^{2h}. By a discrete Fourier analysis of the modes on different grids (Wienands and Oosterlee, 2001) can prove that the k-th Fourier mode on Ω^h is the k-th Fourier mode on Ω^{2h}, see equation 3.3 (Strang, 2006). A transformation from the fine to the coarse grid provides the k-th mode with higher frequency when $1 < k < \frac{n}{2}$.

$$w^h_{k,2j} = sin\left(\frac{2jk\pi}{n}\right) = sin\left(\frac{jk\pi}{\frac{n}{2}}\right) = w^{2h}_{k,j}, \qquad for\ 1 < k < \frac{n}{2} \tag{3.3}$$

$$w^h_{k,2j} = -w^{2h}_{n-k,j}, \qquad for\ \frac{n}{2} < k < n \tag{3.4}$$

For $\frac{n}{2} < k < n$ the k-th mode becomes negative, see equation 3.4, which means that the modes on Ω^{2h} is represented as relatively smooth compared to the oscillating modes on Ω^h. Therefore, it is necessary to reduce the oscillating error on modes on Ω^h before transferring (restricting) it to Ω^{2h}. Usually, 1-2 iterations

is enough for reducing the high frequency errors with an iterative scheme.

To improve the behavior of an iterative method, the initial guess is important. In a multigrid scheme, this is done by solving the problem approximately on the coarsest grid, Ω^{xh}, and transferring (prolongate) the solution to the next finer grid, $\Omega^{(x+1)h}$. The transferred solution can be used as a initial guess on the fine grid since the oscillating error modes are damped on the coarsest grid.

3.1.1 Restriction

The transformation from fine to coarse grid is called *restriction*. A restriction procedure can restrict solutions or residual dependent on the multigrid algorithm. Restriction operator, I_h^{2h}, can be preformed in different ways, in this subsection two different methods is taken into consideration, *injection* and *weighted restriction*.

$$I_h^{2h} : \mathbb{R}^n \rightarrow \mathbb{R}^{n/2}$$
$$\mathbf{u}^{2h} = I_h^{2h}\mathbf{u}^h$$

The fine grid, Ω^h, consists of n grid cells and the coarse grid, Ω^{2h}, consists of $\frac{n}{2}$ grid cells. Node numbering i and j on the coarse grid corresponds to node numbering $2i$ and $2j$ on the fine grid.

Injection

The simplest restriction method is restriction by injection. A injection matrix, I_h^{2h}, for a FDM consists of only one and zeros, i.e. ignores the odd-numbered fine grid values and directly adopt the even-numbered values to the coarse grid. With this method, the error will not be corrected on the odd-numbered fine nodes. This can cause a reduction of the efficiency of the multigrid method.

Weighted restriction

Weighted restriction uses all the nodes on the fine grid in the transformation and can be preformed in several ways, full-weighted, half-weighted etc. The injection matrix I_h^{2h} for *full-weighted* restriction has the relation to the linear interpolation matrix equals to $I_h^{2h} = \frac{1}{2}\left(I_{2h}^h\right)^T$ (Strang, 2006). In 2D, the full- weighted stencil will include nine neighboring points.

3.1.2 Prolongation

The transformation from coarse grid to fine grid is called *prolongation* (or *interpolation*). This is done by a prolongation matrix, I_{2h}^h, which is defined by a local averaging for FDMs. The simplest averaging method is *linear interpolation*, see figure 3.1.

$$I_{2h}^h : \mathbb{R}^{n/2} \rightarrow \mathbb{R}^n$$
$$\mathbf{u}^h = I_{2h}^h \mathbf{u}^{2h}$$

The coarse grid, Ω^{2d}, consists of $\frac{n}{2}$ grid cells and the fine grid, Ω^d, consists of n grid cells. Node numbering *2i* and *2j* on the fine grid corresponds to node numbering *i* and *j* on the coarse grid.

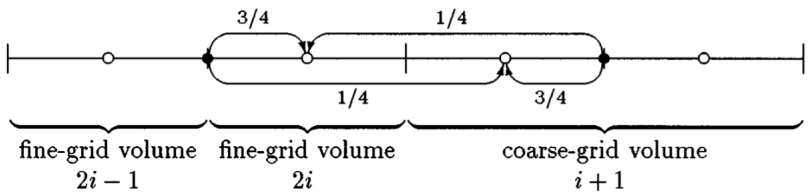

Figure 3.1: Linear interpolation in 1D (Drikakis et al., 1998).

3.2 Different algorithms

There are many possibilities for preforming a multigrid method. In this section algorithms for a *two-step, v-cycle* and *full multigrid* is focused on. For future explanation or reading on other algorithms see Wesseling (1995) and Stüben and Trottenberg (1982).

The new variables in the algorithms is explained here, where v_1 is the *pre-smoothing*, v_2 is the *coarse-smoothing* and v_3 is the *post-smoothing*, see section 3.3 for more information on the smoothing parameters. The residual is denoted as $\mathbf{r}$ and the solution-error as $\mathbf{e}$.

It is important that the right-hand side of the linear system, $\mathbf{f}$, is constant trough the methods, i.e. $\mathbf{f}$ is equal on a downward stoke and an upward stoke for the respective grid depth.

3.2.1 The Two-Level Method

Two-step multigrid can be used to test the *residual equation* 3.2 in a multigrid method. A schematic configuration is given in figure 3.2 and the algorithm is:

1 Iterate $A^h\mathbf{u}^h = \mathbf{f}^h$, v_1-times.

2 Compute the residual $\mathbf{r}^h = \mathbf{f}^h - A^h\mathbf{u}^h$ on Ω^h and restrict it to Ω^{2h}, $\mathbf{f}^{2h} = I_h^{2h}\mathbf{r}^h$.

3 Iterate $A^{2h}\mathbf{e}^{2h} = \mathbf{f}^{2h}$ v_2-times or solve directly. Start with the initial guess $\mathbf{e}^{2h} = 0$.

4 Prolongate $\mathbf{e}^{2h}$ to Ω^{2h}, $\mathbf{e}^h = I_{2h}^h\mathbf{e}^{2h}$ and update the solution $\mathbf{u}^h = \mathbf{u}^h + \mathbf{e}^h$.

5 Iterate $A^h\mathbf{u}^h = \mathbf{f}^h$, v_3-times.

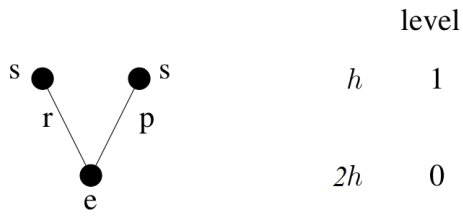

Figure 3.2: Schematic of two-step multigrid, modification from Strang (2006).

3.2.2 V-Cycle Multigrid Method

A V-cycle multigrid method starts similar as a two-step multigrid method, but the depth of the cycle can vary, see figure 3.3. The V-cycle multigrid method is the simplest one and the algorithm is:

1 Iterate $A^h\mathbf{u}^h = \mathbf{f}^h$, ν_1-times. Store $\mathbf{u}^h$ and $\mathbf{f}^h$.

2 Compute the residual $\mathbf{r}^h = \mathbf{f}^h - A^h\mathbf{u}^h$ on Ω^h and restrict it to Ω^{2h}, $\mathbf{f}^{2h} = I_h^{2h}\mathbf{r}^h$. Store $\mathbf{f}^{2h}$.

3 Iterate $A^{2h}\mathbf{e}^{2h} = \mathbf{f}^{2h}$ ν_2-times and store $\mathbf{e}^{2h}$. Start with the initial guess $\mathbf{e}^{2h} = 0$.

4 Compute the residual $\mathbf{r}^{2h} = \mathbf{f}^{2h} - A^{2h}\mathbf{u}^{2h}$ on Ω^{2h} and restrict it to Ω^{4h}, $\mathbf{f}^{4h} = I_{2h}^{4h}\mathbf{r}^{2h}$. Store $\mathbf{f}^{4h}$.

- Continue until coarsest grid is reached ...

5 Solve coarsest grid $A^{xh}\mathbf{e}^{xh} = \mathbf{f}^{xh}$ by iterating ν_3-times or solve directly. Start with the initial guess $\mathbf{e}^{xh} = 0$.

6 Prolongate $\mathbf{e}^{xh}$ to $\Omega^{(x+1)h}$, $\mathbf{e}^{(x+1)h} = I_{xh}^{(x+1)h}\mathbf{e}^{xh}$ and update the solution $\mathbf{e}^{(x+1)h} = \mathbf{e}_{old}^{(x+1)h} + \mathbf{e}_{new}^{(x+1)h}$.

7 Iterate $A^{(x+1)h}\mathbf{e}^{(x+1)h} = \mathbf{f}^{(x+1)h}$, ν_3-times.

- Continue until finest grid is reached ...

8 Prolongate $\mathbf{e}^{2h}$ to Ω^h, $\mathbf{e}^h = I_{2h}^h \mathbf{e}^{2h}$ and update the solution $\mathbf{u}^h = \mathbf{u}^h + \mathbf{e}^h$.

9 Iterate $A^h \mathbf{u}^h = \mathbf{f}^h$, ν_3-times.

The V-cycle algorithm can be preformed in different number of cycles, γ, by using a for-loop over step 2-9.

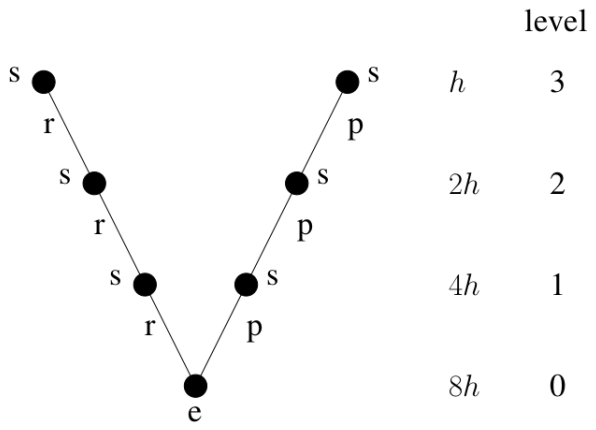

Figure 3.3: Schematic of V-cycle multigrid, modification from Strang (2006).

3.2.3 The Full Multigrid Method

The full multigrid method start at the coarsest grid and prolongate up to the next finer grid to provide a good initial guess. Then V-cycles of each depth are preformed to improve the solution until the finest grid is reached, see figure 3.4. The full multigrid algorithm is:

1 Store $\mathbf{f}^h$.

2 Restrict $\mathbf{f}^h$ to Ω^{2h}, $\mathbf{f}^{2h} = I_h^{2h} \mathbf{f}^h$ and store $\mathbf{f}^{2h}$.

- Continue until the coarsest grid is reached ...

3 Restrict $\mathbf{f}^{(x-1)h}$ to Ω^{xh}, $\mathbf{f}^{xh} = I^{xh}_{(x-1)h}\mathbf{f}^{(x-1)h}$.

4 Solve coarsest grid $A^{xh}\mathbf{e}^{xh} = \mathbf{f}^{xh}$ by iterating ν_3-times or solve directly and store $\mathbf{e}^{xh}$. Start with the initial guess $\mathbf{e}^{xh} = 0$.

5 Prolongate $\mathbf{e}^{xh}$ to $\Omega^{(x+1)h}$, $\mathbf{e}^{(x+1)h} = I^{(x+1)h}_{xh}\mathbf{e}^{xh}$ and store $\mathbf{e}^{(x+1)h}$.

- Run V-cycles, see subsection 3.2.2, for each depth until the finest grid is reached ...

6 Prolongate $\mathbf{e}^{2h}$ to Ω^h, $\mathbf{e}^h = I^h_{2h}\mathbf{e}^{2h}$ and update the solution $\mathbf{u}^h = \mathbf{u}^h + \mathbf{e}^h$.

7 Iterate $A^h\mathbf{u}^h = \mathbf{f}^h$, ν_3-times.

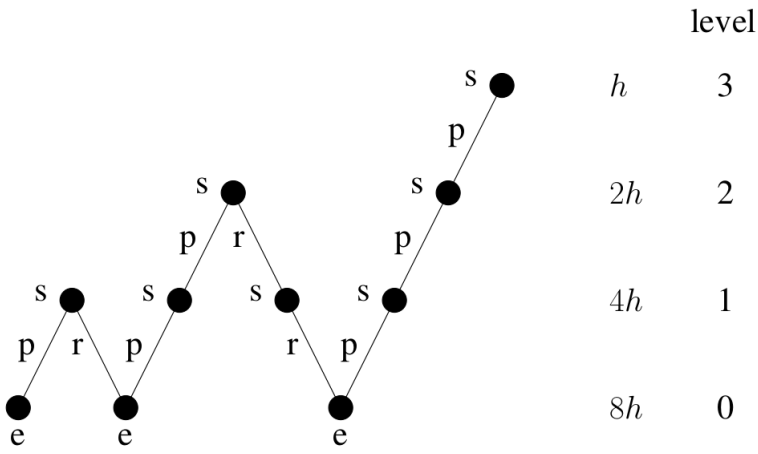

Figure 3.4: Schematic of full multigrid, modification from Strang (2006).

3.3 Smoothening Parameters

The purpose of a multigrid method is to reach convergence more efficiently then an iterative method. For a PDE problem, the accuracy is solved to $\mathcal{O}(n^2)$ in 5-12 iterations (Mccormick, 1988).

During a multigrid scheme, there are stated three different values for iterations, *pre-smoothing*, *coarse-smoothing* and *post-smoothing*. This is done to solve the multigrid method as efficient as possible.

For *pre-smoothing*, which is used in a downward stoke, there only needs to be preformed a few iterations, 1-3, depending on the problem to be solved. The purpose (goal) here is not to reach convergence, but to reduce the high frequencies which an iterative scheme do quite fast, see section 3.1 for more details.

For *coarse-smoothing*, which is used to smooth the coarsest grids, there can be preformed a higher number of iterations. Usually, the cost and time spent on iterating on the coarsest grids is quite low compared to iterating on the finest grid. Therefore it is expedient (appropriate) to find a optimal number for *coarse-smoothing* so the initial guess for the next coarsest grid is good. This number will make a change in the computational work and the running time for a multigrid method.

For *post-smoothing*, which is used in an upward stoke, number of iterations can vary depending on the the problem to be solved. There should not be necessary to iterate to many times. Usually, it is better to change the *coarse-smoothing* or number of cycles instead of increasing the number of *post-smoothing* iterations, to reach convergence.

3.4 Convergence and Computational Work

It is possible to find a convergence factor β, which is the spectral radius of the overall iteration matrix, by numerical analysis Strang (2006). β is a constant factor less than one, $\beta < 1$, and independent of the grid size h. Hence, theoretically the number of iterations is bounded independent of the grid level on the linear system and only dependent of the accuracy.

To achieve an optimal multigrid algorithm, the number of floating point operations (flops) per multigrid cycle should behave like $\mathcal{O}(n)$, where n is the number of degrees of freedom (unknowns). Number of flops for a deep v-cycle or a full multigrid is greater than for a two-step multigrid Strang (2006), the relation is given in equation 3.5.

$$\textbf{Flops in full MG} < \frac{2^d}{2^d-1}(\textbf{Flops in V}-\textbf{cycle}) < \left(\frac{2^d}{2^d-1}\right)^2(\textbf{Flops in two}-\textbf{step}) \tag{3.5}$$

d is the dimension in space and the power of 2 is related to the transpose relation between a fine and a coarse grid. Hence, the total number of flops are asymptotically optimal for multigrid methods.

Good programing is decisive to optimize the multigrid algorithm together with the chosen transposing methods. Regardless, the multigrid method is one of the most efficient iterative methods known today (Hackbusch, 1985).

Chapter 4

Immersed Boundary Method

The governing equations for the fluid are easily adopted to *fluid-structure interaction* problems. This is done by an approach to model the coupling between the structure and the fluid. Hence, the immersed boundary method (IBM) can be used to solve the coupling.

The IBM was first proposed and implemented by Peskin (1972). He inserted an immersed boundary of the flexible leaflet of a human heart valve and presented it as a method for solving the Navier-Stokes equation on a rectangular domain accomplished by replacing the boundary by a field of force defined on the mesh points of the rectangular domain. Later, the method was adopted by Lai and Peskin (2000) to simulate rigid boundaries.

In IBM, the immersed boundary points on a geometry does not need to conform with the Eulerian grid points in the computational fluid domain. The points are applied to a Lagrangian grid, but by imposing the immersed boundary conditions on the geometry the immersed boundary can be handled in the Eulerian grid by modifying the governing equations.

There are several advantages by using the IBM to apply a geometry in a fluid:

- The geometry does not need to fit the computational grid.
- It can handle moving boundaries.
- Grid complexity and quality are not significantly affected.
- The computational cost of each grid node is generally less expensive, i.e. less memory and CPU are used.
- IBM solved by explicit time stepping schemes require solvers only for the Eulerian equations (Guy et al., 2015).
- Works together with multigrid methods.

4.1 Imposing of Immersed Boundary Conditions

Imposing of the immersed boundary conditions is the key factor in an immersed boundary algorithm. The method used to impose these boundary conditions is what distinguishes the algorithms, i.e. how to modify the governing equations. The domain occupied by the geometry is denoted by Ω_b and the immersed boundary by Γ_b, then the immersed boundary conditions is specified as equation 4.1.

$$\mathbf{u} = \mathbf{u}_\Gamma \qquad and \qquad \frac{\partial p}{\partial \mathbf{x}} = 0 \qquad on \ \Gamma_b \tag{4.1}$$

The governing equations (2.5, 2.6 and 2.4) for the domain Ω is discretized without taking the immersed boundary into account. The immersed boundary condition is imposed indirectly by modifying 2.5 and 2.6 with a forcing function $\mathbf{f_b}$, see 4.2, representing the effect of the immersed boundary.

$$\mathbf{u}_t + \mathbf{u}\cdot\nabla\mathbf{u} = -\nabla p + \frac{1}{Re}\nabla^2\mathbf{u} + \mathbf{f_b} \quad on\ \ \Omega \tag{4.2}$$

$$\nabla\cdot\mathbf{u} = 0 \tag{4.3}$$

The forcing function can be implemented in two different ways:

1. *Continuous forcing method* - $\mathbf{f_b}$ is applied to the entire domain, i.e. 4.2 is solved on $(\Omega + \Omega_b)$.

2. *Discrete forcing method* - first, $\mathbf{f_b}$ = 0 and 4.2 is discretized without the immersed boundary, then the cell-velocities near the immersed boundary are adjusted to account for the immersed boundary.

When a continuous forcing method is used, elastic and rigid boundaries needs different treatment. This method will not be described here, for future discussions around elastic boundaries read Peskin (1972) and for rigid boundaries read Goldstein et al. (1993).

4.2 Discrete Forcing Methods

Discrete forcing methods are divided into *indirect* and *direct* forcing approaches. Indirect approaches are imposing the immersed BC's indirectly on the immersed boundary. Otherwise, the direct forcing approaches are imposing the immersed BC's directly on the immersed boundary, which will be future discussed here.

4.2.1 Direct Forcing approach

The direct forcing approach was derived by Mohd-Yusof (1997). He provided a method where the velocity value is imposed directly on the boundary and the implementation do not require additional CPU time. To demonstrate the main

steps in the approach assume a 2D immersed boundary with the Eulerian grid points (i, j). Considering the equation 4.2 discretized in time by the explicit Euler method, equation 4.4 arise.

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = RHS_{i,j} + f_{i,j} \tag{4.4}$$

$n+1$ and n are respectively the next and the present time-step, i, j is the location on the Eulerian grid. For $u_{i,j}^{n+1} = U_{i,j}$, where $U_{i,j}$ is the fluid velocity next to the immersed boundary, then equation 4.5 for $f_{i,j}$ yields.

$$f_{i,j} = \frac{U_{i,j} - u_{i,j}^{n}}{\Delta t} - RHS_{i,j} \qquad near\ \Gamma_b \tag{4.5}$$

Since the immersed boundary does usually not coincide with the underlying grid, an interpolation procedure needs to be used to determine $U_{i,j}$ close to the intersection points.

When the projection method from subsection 2.1.3 is used to decouple the velocities and the pressure, the immersed BC's are forced on the predicted velocities, $u_{i,j}^{*}$ and $v_{i,j}^{*}$, together with the interpolation procedure. Balaras (2004) prescribe this simplification without reducing the temporal accuracy of the method.

4.2.2 Interpolation

Different interpolation schemes can be used to find the velocities close to the immersed boundary. Fadlun et al. (2000) presents three different schemes on a staggered grid (see figure 4.1), *no interpolation*, *volume fraction weighting* and *velocity interpolation*, and tested the results they provided. The *velocity interpolation* gave the best results and will be described here.

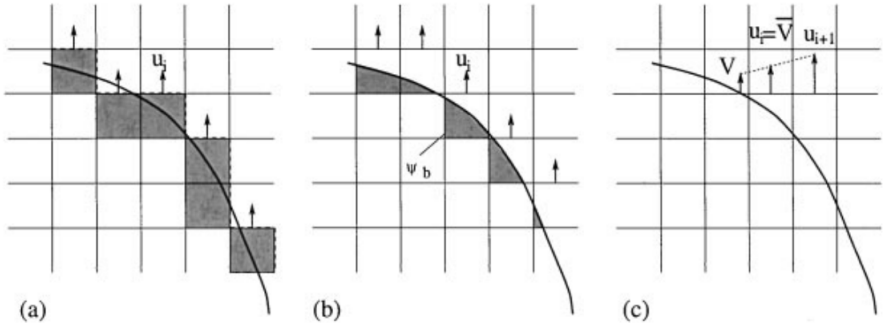

Figure 4.1: From Fadlun et al. (2000). (a) no interpolation,(b) volume fraction weighting,(c) linear interpolation.

The interpolation scheme presented by Fadlun et al. (2000) is different and more accurate than the interpolation scheme presented by Peskin (1972). The interpolation is done by a linear approximation of the velocities close to the immersed boundary wall. The velocity profiles are assumed to be approximately linear from the wall if the grid around the immersed boundary is fine enough.

Chapter 5

Code layout

A Navier Stokes solver is developed to preform case studies with multigrid and IBM. The incompressible Navier Stokes equations for a 2-dimensional flow is solved on an equidistant grid and the following sections will give a short introduction on the code layout, methods used, assumptions taken and limitations on the code. The respective code is found in Appendix B and a flow chart can be seen in figure 5.1.

The programming language used to develop the program is C with compiler **gcc** and flags **-W** and **-o**. Double precisions is used to ensure convergence of the simulations. A vtf-file is written in the end of each time-step (or at chosen time-steps) to visualize the result in GLview. The program is ran on **Ubuntu 16.04** with 4 (64-bit, 1.8GHz) CPUs.

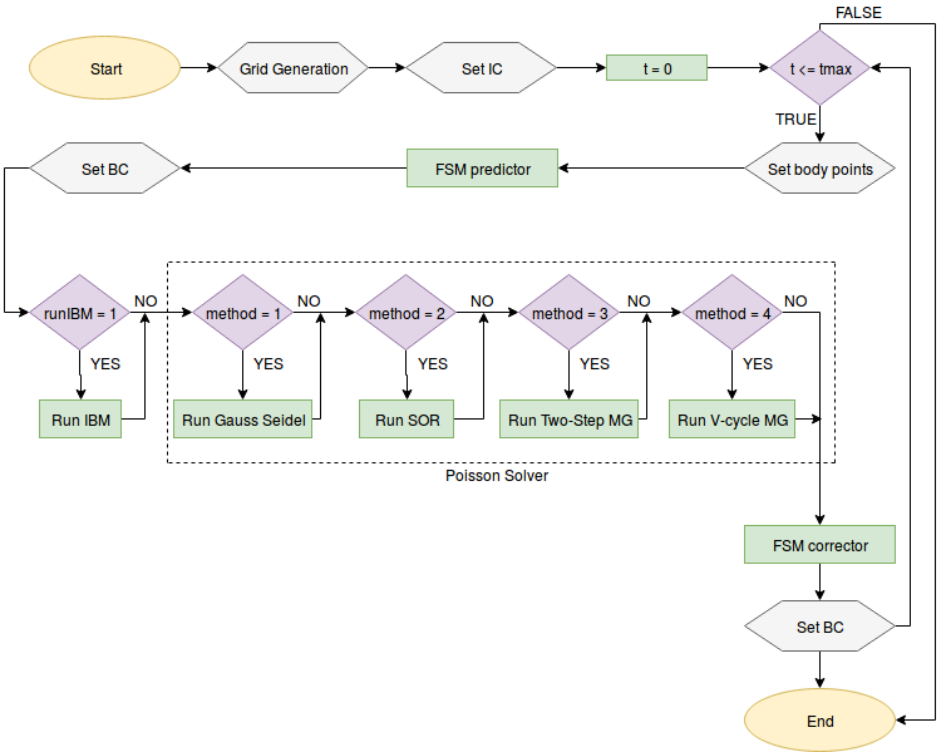

Figure 5.1: Flow chart of the Navier Stokes solver

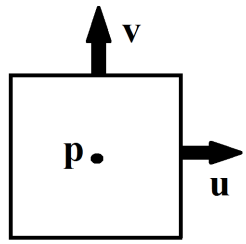

Figure 5.2: A staggered grid cell at position i, j.

5.1 Staggered Grid Generation

The solver uses an equidistant staggered grid in the computational domain. When refering to u, v and p in the computational node, the definitions given in figure 5.3, 5.4 and 5.5 are the current one. By using these definitions, half-integer velocity adresses are not needed and a cell i defined as in figure 5.2.

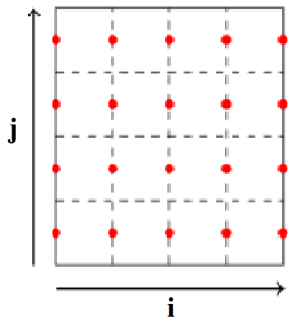

Figure 5.3: The representation of u-velocity in the staggered grid.

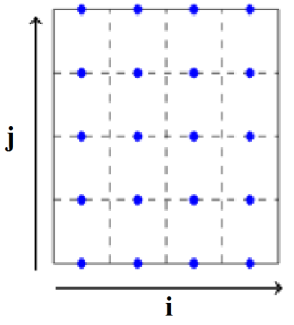

Figure 5.4: The representation of v-velocity in the staggered grid.

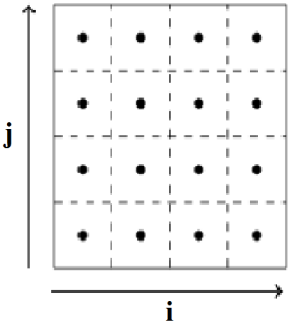

Figure 5.5: The representation of the pressure in the staggered grid.

5.2 Finite Difference Scheme

A FDM scheme, represented by the nodes in figure 5.6, 5.7 and 5.8, is used to solve the spatially differential terms in equation 2.10 as $RHS(u,v,p)$ in 5.1. The FDM-stencil presented by Harlow and Welch (Harlow et al., 1965) is used with the following equations for the viscous terms 5.3, the advective terms 5.2 and the pressure terms 5.4. $h = \Delta x = \Delta y$ since an equidistant grid is used and u.

$$\frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} = RHS(u,v,p) \tag{5.1}$$

Where $\mathbf{u} = \{u,v\}$ is the velocities in 2D.

$$
\begin{aligned}
A_x = \frac{0.25}{h}\Big(&\big(u_{i+1,j}+u_{i,j}\big)^2-\big(u_{i,j}+u_{i-1,j}\big)^2 \\
&+\big(u_{i,j+1}+u_{i,j}\big)\big(v_{i+1,j}+v_{i,j}\big)-\big(u_{i,j}+u_{i,j-1}\big)\big(v_{i+1,j-1}+v_{i,j-1}\big)\Big) \\
A_y = \frac{0.25}{h}\Big(&\big(u_{i,j+1}+u_{i,j}\big)\big(v_{i+1,j}+v_{i,j}\big)-\big(u_{i-1,j}+u_{i-1,j-1}\big)\big(v_{i-1,j}+v_{i,j}\big) \\
&+\big(v_{i,j}+v_{i,j+1}\big)^2-\big(v_{i,j-1}+v_{i,j}\big)^2\Big)
\end{aligned}
\tag{5.2}
$$

$$
\begin{aligned}
V_x &= \frac{1}{Re}\Big(\frac{u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}-4u_{i,j}}{h^2}\Big) \\
V_y &= \frac{1}{Re}\Big(\frac{v_{i+1,j}+v_{i-1,j}+v_{i,j+1}+v_{i,j-1}-4v_{i,j}}{h^2}\Big)
\end{aligned}
\tag{5.3}
$$

$$
\begin{aligned}
P_x &= \frac{p_{i,j}-p_{i+1,j}}{h} \\
P_y &= \frac{p_{i,j}-p_{i,j+1}}{h}
\end{aligned}
\tag{5.4}
$$

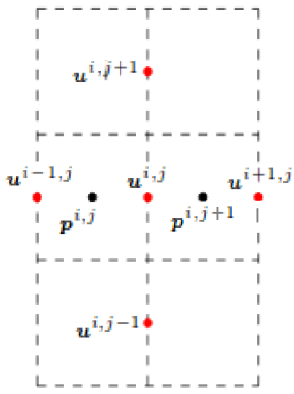

Figure 5.6: The representation of pressure and velocities used for x-direction calculations.

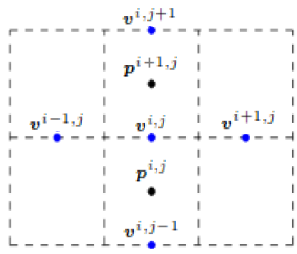

Figure 5.7: The representation of pressure and velocities used for y-direction calculations.

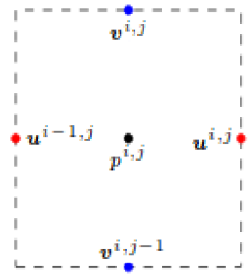

Figure 5.8: Velocities used to calculate $p_{i,j}$.

5.3 Explicit Euler Scheme

An explicit Euler scheme 2.11 is used to solve the temporal discretization. Hence, the Navier Stokes equations will be solved for each Δt until *tmax* is reached, starting from $t = 0$, see 5.1. The current spatial values are used in the right-hand side solution according to the explicit Euler scheme, see equation 5.5.

$$\frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} = RHS(u^n, v^n, p^{n+1}) \tag{5.5}$$

To solve the pressure-velocity coupling by the *projection method* a predictor, PM-predictor, is stated from the euler scheme. The output from the PM-predictor results in the predicted velocities, equation 5.6, which are used to form the right-hand side of the Poisson equation. After the Poisson equation is solved, a projection method corrector, PM-corrector, will update the non-solenoidal velocities with the pressure correction found in the Poisson solver, see equation 5.7.

$$\begin{aligned} u^* &= u^n + \Delta t RHS(u^n, v^n, p^n) \\ v^* &= v^n + \Delta t RHS(u^n, v^n, p^n) \end{aligned} \tag{5.6}$$

$$\begin{aligned} u^n &= u^* - \Delta t \frac{\Delta p_{i+1,j} - \Delta p_{i,j}}{h} \\ v^n &= v^* - \Delta t \frac{\Delta p_{i,j+1} - \Delta p_{i,j}}{h} \end{aligned} \tag{5.7}$$

$$p^{n+1} = p^n + \Delta p \tag{5.8}$$

5.4 Poisson Solver

The Poisson equation can be solved with different equation solvers. In the present code, Gauss Seidel, SOR, two-step multigrid and V-cycle multigrid are compared. The right-hand side, f, is stated as the divergence of the predicted velocities, u^* and v^*, divided by Δt.

$$-\Delta p = f \qquad in \quad \Omega = (0,1)^2 \tag{5.9}$$

5.4.1 Gauss Seidel

Gauss Seidel method is implemented straight forward. Equation 5.10 presents the scheme to be iterated for each iteration k.

$$\Delta p_{i,j}^k = \frac{1}{4}\Big(\Delta p_{i+1,j}^{k-1} + \Delta p_{i-1,j}^k \, \Delta p_{i,j+1}^{k-1} + \Delta p_{i,j-1}^k - h^2 f_{i,j}\Big) \tag{5.10}$$

5.4.2 Successive Over Relaxation

For SOR method it is reasonable to find the optimal omega, ω_{opt}, for the currant grid size. ω_{opt} is found by linear interpolation between the grid size and interpolation values found from experimental results. The iterative scheme solved is presented in equation 5.11 and iterated for each iteration k.

$$\Delta p_{i,j}^k = \Big(1-\omega_{opt}\Big)\Delta p_{i,j}^{k-1} + \frac{\omega_{opt}}{4}\Big(\Delta p_{i+1,j}^{k-1} + \Delta p_{i-1,j}^k \, \Delta p_{i,j+1}^{k-1} + \Delta p_{i,j-1}^k - h^2 f_{i,j}\Big) \tag{5.11}$$

5.4.3 Multigrid

The multigrid method is implemented to accelerate the rate of convergence of the Poisson solver. Three different multigrid algorithms are implemented, two-step multigrid (see algorithm in subsection 3.2.1), v-cycle multigrid (see algorithm in subsection 3.2.2) and full multigrid(see algorithm in subsection 3.2.3). All the algorithms are using Gauss-Seidel, presented in subsection 5.4.1, to find an approximate solution. The number of interior points in x- and y-*direction* is 2^d in a [0:1]x[0:1] domain, where d is the number of multigrid levels.

In 2D, the *injection* done by the *restriction* matrix I_h^{2h} in the present code is presented in equation 5.12. n is the number of fine grid points, h represents the fine grid and $2h$ the coarse grid.

$$\Delta p_{i,j}^{2h} = \Delta p_{2i,2j}^{h}, \quad i = 1,...,\frac{n}{2} \; and \; j = 1,...,\frac{n}{2} \tag{5.12}$$

The *prolongation* matrix I_{2h}^{h} for linear interpolation will in 2D be represented as *bilinear interpolation*, see figure 5.9 with corresponding equation 5.13, where n is the number of coarse grid points.

$$\begin{aligned}
e_{2i,2j}^{h} &= \frac{9}{16}e_{i,j}^{2h} + \frac{3}{16}(e_{i+1,j}^{2h} + e_{i,j+1}^{2h}) + \frac{1}{16}e_{i+1,j+1}^{2h}, \\
e_{2i+1,2j}^{h} &= \frac{9}{16}e_{i+1,j}^{2h} + \frac{3}{16}(e_{i+1,j+1}^{2h} + e_{i,j}^{2h}) + \frac{1}{16}e_{i,j+1}^{2h}, \\
e_{2i,2j+1}^{h} &= \frac{9}{16}e_{i,j+1}^{2h} + \frac{3}{16}(e_{i+1,j+1}^{2h} + e_{i,j}^{2h}) + \frac{1}{16}e_{i+1,j}^{2h}, \\
e_{2i+1,2j+1}^{h} &= \frac{9}{16}e_{i+1,j+1}^{2h} + \frac{3}{16}(e_{i+1,j}^{2h} + e_{i,j+1}^{2h}) + \frac{1}{16}e_{i,j}^{2h}, \\
for \quad i &= 0,...,n \quad and \quad j = 0,...,n.
\end{aligned} \tag{5.13}$$

To improve the *prolongation*, *mixed interpolation*, see figure 5.10, are implemented at the boundaries on the fine grid. *Mixed interpolation* is prescribed under section 3.1.2 and developed in the present code as equation 5.14. $boundary$ represents a boundary point in the fine grid for e^h and in the coarse grid for e^{2h}. n is the number of coarse grid points.

$$
\begin{aligned}
e^h_{boundary,2j} &= \frac{3}{4}e^{2h}_{boundary,j} + \frac{1}{4}e^{2h}_{boundary,j+1},\\
e^h_{boundary,2j+1} &= \frac{3}{4}e^{2h}_{boundary,j+1} + \frac{1}{4}e^{2h}_{boundary,j},\\
e^h_{2i,boundary} &= \frac{3}{4}e^{2h}_{i,boundary} + \frac{1}{4}e^{2h}_{i+1,boundary},\\
e^h_{2i+1,boundary} &= \frac{3}{4}e^{2h}_{i+1,boundary} + \frac{1}{4}e^{2h}_{i,boundary},\\
for \quad i = 0,...,n \quad and \quad j = 0,...,n.
\end{aligned}
\tag{5.14}
$$

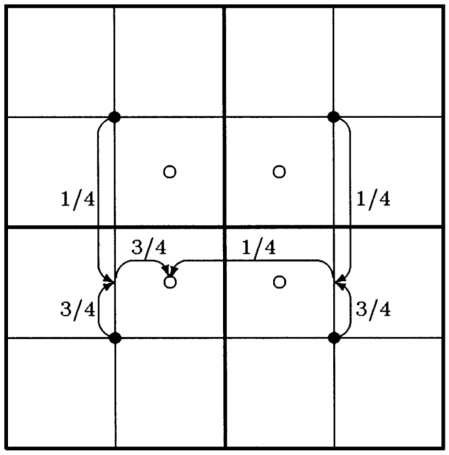

Figure 5.9: Bilinear interpolation (Drikakis et al., 1998).

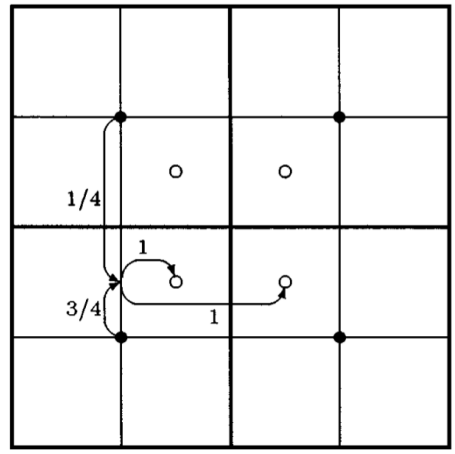

Figure 5.10: Mixed interpolation (Drikakis et al., 1998).

5.5 IBM

To insert a geometry in the fluid flow, IBM is used. The IBM developed in the present code is based on Mossige (2017) and measures the force function by *direct force approach*, which is a *discrete force method*.

Coinciding with theory prescribed in chapter 4, the velocities close to the immersed boundary should be modified. In order to find the positions relative to the underlying computational grid, the staggered grid configurations needs to be taken into consideration. To interpolate parallel to the velocity directions, which is done in the present code, a grid shifting of the velocity is accomplished. To save computational time, a domain surrounding the immersed boundary is defined, see fiure 5.11. A representation of parallel linear interpolation and the grid shift are given in figure 5.13 and 5.12.

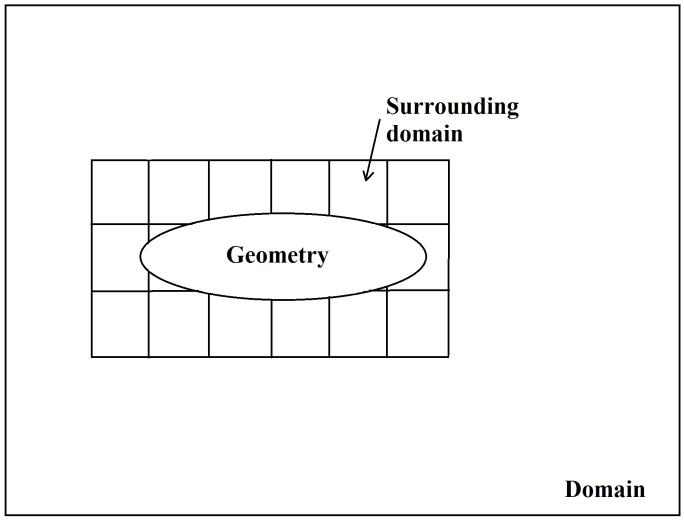

Figure 5.11: A demonstration of the surrounding domain for an immersed boundary.

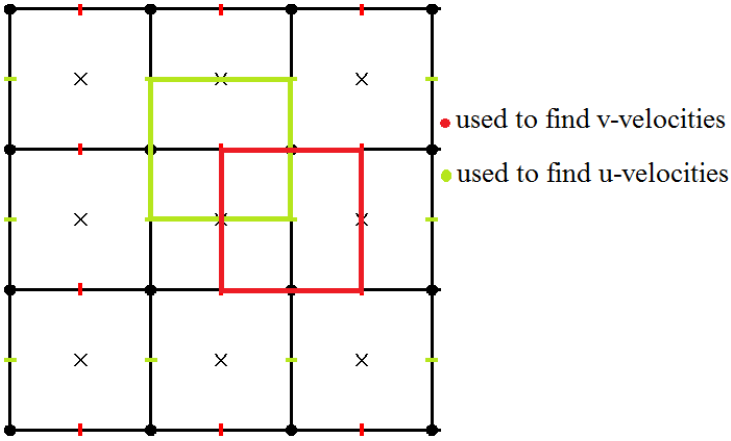

Figure 5.12: How to apply grid shift on the surrounding domain. The black grid is the original staggered grid, red is v-velocities and green is u- velocities.

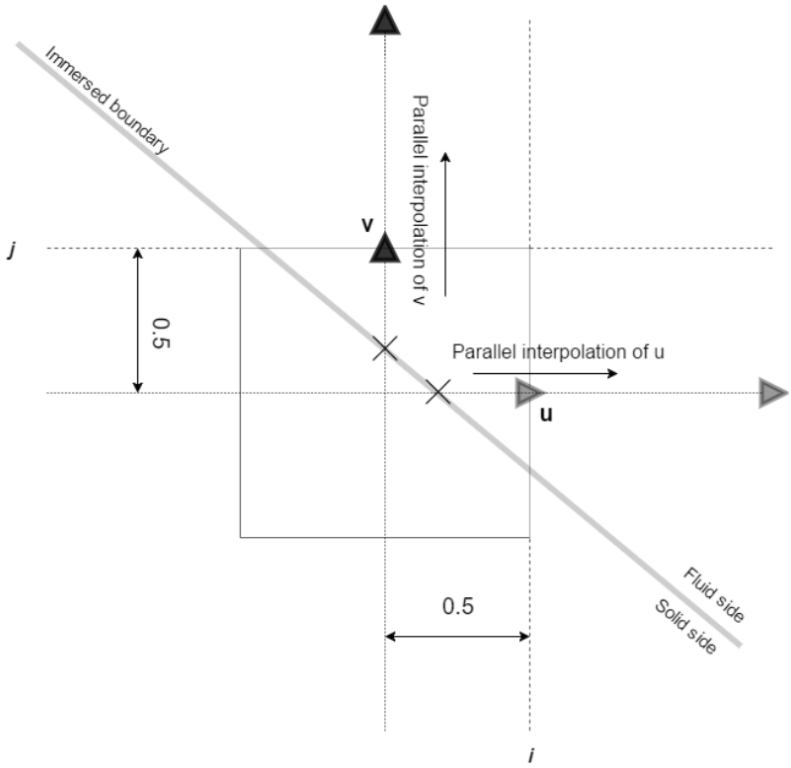

Figure 5.13: Parallel interpolation after $\frac{1}{2}$-gridshift, from Mossige (2017).

The direction into the fluid from the immersed boundary is detected for each intersection point and the two first fluid node velocities are used in the linear interpolation together with the immersed boundary velocity. The purpose of the interpolation is to modify the first fluid node with respect to the two other nodes, i.e. the effect of the boundary is stated indirectly. Since the predicted velocities, see equation 5.6, are used in the IBM, the interpolated velocities are flagged to not be corrected by the PM-corrector. Pressure boundary conditions are not imposed according to Fadlun et al. (2000) and the internal flow flied is a freely developed.

Force coefficients for lift and drag are calculated from the lift and drag forces, see equation 5.15. For each intersection point, i, a contribution to the forces are measured by Newton's second law and third law, see equation 5.16, where the change in velocity in x-direction will give a contribution to the drag force and the change in velocity in y-direction will give a contribution to the lift force. Hence, the forces are equal but has the opposite direction of the influence the immersed boundary has on the fluid. u_d and v_d denotes the difference between the interpolated and the predicted velocity, A is the areal of the geometry, u_∞ is the inflow velocity, ρ is the density and ρh^2 represents the mass.

$$\begin{aligned} C_D &= 2\frac{F_D}{\rho u_\infty^2 A} \\ C_L &= 2\frac{F_L}{\rho u_\infty^2 A} \end{aligned} \tag{5.15}$$

$$\begin{aligned} F_D &= \sum_i -\frac{u_d}{\Delta t}\rho h^2 \\ F_L &= \sum_i -\frac{v_d}{\Delta t}\rho h^2 \end{aligned} \tag{5.16}$$

When a fluid flows over a geometry, the geometry exerts a (pressure) force, and if a geometry surface is no-slip, the fluid exerts a share force too. Conceptually, the geometry exerts a force with opposite direction on the fluid (Goldstein et al., 1993).

5.5.1 Limitations on the Immersed Boundary

How to define the immersed boundary geometry has certain limitations, which needs to be taken into consideration:

- The geometry should be located $2h$ from the boundary cells because of the choice of interpolation nodes described earlier.
- The geometry should be a closed polygon.
- Grid generation around the immersed boundary have to be fine enough to assume linear velocity profiles.
- The immersed boundary cannot be too close to another part of the immersed boundary, then an interpolated velocity can be used to interpolate a new velocity.
- The forced motion procedure on a body is not tested, hence may not be perfectly developed.

The IBM-module is not complete due to the fluid structure interaction problem. It still reminds to implement the deformation and the velocity response of the immersed boundary. At this point, the boundaries can only move by a pre-described motion as a closed rigid geometry.

5.6 Convergence criteria

For the Poisson solver, the convergence criteria is 10^{-6} due to the infinity-norm, see equation 5.17. Δp_d is the change in Δp per iteration.

$$\|\Delta p_d\|_\infty \leq 10^{-6} \tag{5.17}$$

5.7 General Limitations

To define the domain in the refinement test, a change where made to handle rectangular domains. This includes the *dimX* and *dimY* in the code, which are factors multiplied by the number of cells in the original domain [0:1]x[0:1]. Be aware of that *dimX* and *dimY* needs to be equals to or larger than 1.0. To get right post-processing in GLview, *dimX* needs to be larger or equals to *dimY*, but this has no impact on the computations done before the post-processing stage. The IBM is not affected by the multiplication, hence the immersed boundary need to be placed in the domain [0:1]x[0:1] with the same restrictions as described earlier. This modification of the code is not optimal, but are done in the late stage of the implementation process due to efficiency of the computational work.

During the implementation some changes were made in the code, which have impact on the code setup. Static allocation of memory where implemented first, but during the implementation of multigrid methods, too much memory where occupied for larger problems. Therefore, a change to dynamic allocation where made. By lack of time, just the necessary modifications for this change where implemented.

5.8 Post-processing

In the post processing stage, an averaging of the velocities and pressure is done to be represented in the same node as specified for visualization in GLview. The averaging is only for the visualization and will not affect the previous calculations steps. When using IBM, the interpolated velocities close to the immersed boundary are already specified, but this is not taken into consideration at this stage. This means that the visualization of velocities closest to the boundary will be affected by the velocities inside of the immersed boundary and the BC's at the wall are not achieved in the visualization. This should be taken into consideration when analyzing the figures.

Chapter 6

Validation

The developed solver is validated to check accuracy, efficiency and force coefficients, a final test case is done in the end to collect it all. First, the Poisson solver is validated and the most efficient method with acceptable accuracy is used in the following. Further, the chosen method is tested together with the IBM in a final case. Refinement tests are preformed before the final test case to optimize the simulations. An efficiency overview for one time-step of the final test case is presented at the end.

The fluid solver without IBM is assumed to work OK and is not validated here. The fluid solver is previously validated in my project thesis (Aarsnes), where the solver is based on the same methods as the present solver and SOR method is used to solve the Poisson equation. Some modifications are done in the code, but it is still reasonably to assume that the validations are valid.

6.1 Poisson Solver

In the Navier-Stokes solver, there are several possible methods defined to solve the Poisson equation 2.9. Only iterative methods has been developed according to the large model problem and memory limitations. Two simple methods, *Gauss Seidel* and *SOR*, are tested against the more complex *multigrid* methods. In subsection 6.1.1 the Poisson solver is run with Gauss Seidel method and SOR method, in subsection 6.1.2 it is run with different multigrid methods. First, a test problem is defined.

Test Problem

A test problem is defined and used in all validations under this section. The domain, Ω = [0,1]x[0,1], have *periodic boundary conditions* at $d\Omega$ and a non-zero forcing function 6.1 of the Poisson equation. The analytical solution of the problem is given in equation 6.2. Mesh resolution in the tests for the Poisson solver is 256x256 with h = $\frac{1}{256}$. x and y refers to the position on the 2D domain.

$$\mathbf{f}(x,y) = cos(2\pi x)cos(2\pi y) \tag{6.1}$$

$$p_{exact} = \frac{1}{(2\pi)^2} cos(2\pi x)cos(2\pi y) \tag{6.2}$$

6.1.1 Gauss Seidel and SOR

Figure 6.1 and figure 6.2 shows the analytical solution, equation 6.2, plotted against the numerical solutions preformed by Gauss Seidel and SOR. The solution values lie along a line in the middle of the domain, horizontally (y = 0.5). In both cases, the numerical results are following the analytical solution in an acceptable way by crossing it, hence the curvature line of the difference given in figure 6.3 and figure 6.4. The difference between the analytical solution and the numerical solution turns out asymmetric with the left slope slower than the right slope. Thereby, there is a small displacement in the Poisson solver preformed by the Gauss Seidel method. This could arise from how the solver sweep trough the domain in x- and y-direction. Here, the Gauss Seidel method sweep from the lower left corner to the upper right corner of the domain. Since y is constant trough the domain, the solution passes from left to right and can permit the largest errors to accumulate at the high values of x (Kahan, 1958). When this displacement occurs in the Gauss Seidel method, it will also occur in the other methods which are based on the Gauss Seidel algorithm. Figure 6.4 is a plot of the difference when using SOR method to solve the Poisson equation numerically. SOR can be seen as a method who applies a correction to the value already obtained from the Gauss Seidel method based on extrapolation from previous iterations (Richard H. Pletcher, 2013) and therefor provide a more asymmetric but accurate solution.

From Kahan (1958), SOR should smooth down the error to the convergence criteria (see subsection 5.6) faster then Gauss Seidel if the damping factor ω is optimal. Figures 6.5 and 6.6 shows how the residual, Δp^k - Δp^{k-1}, propagate during the iterations, k, for the two methods. The Gauss Seidel method smooths the Poisson equation properly in 3431 iterations and SOR in 571 iterations. Hence, SOR method are proven to be faster then Gauss Seidel.

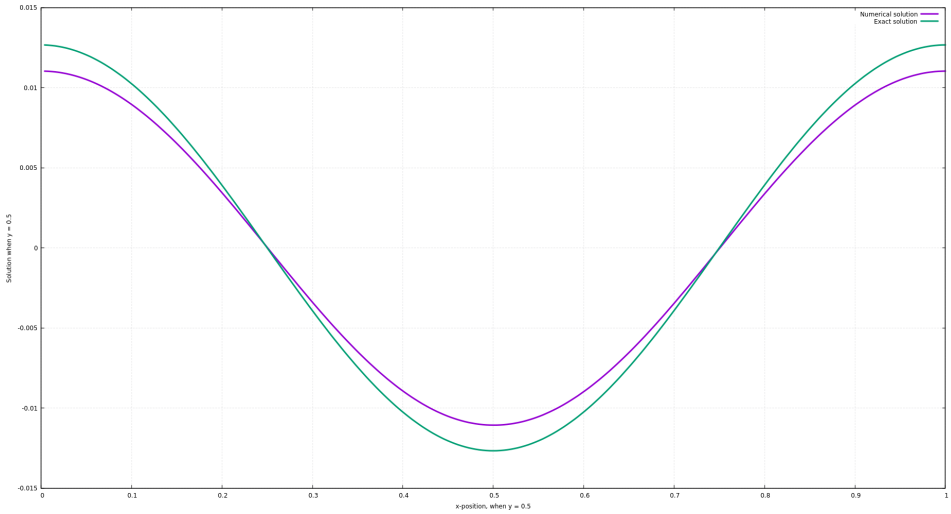

Figure 6.1: Analytical solution (green) vs numerical solution (purple) by Gauss Seidel.

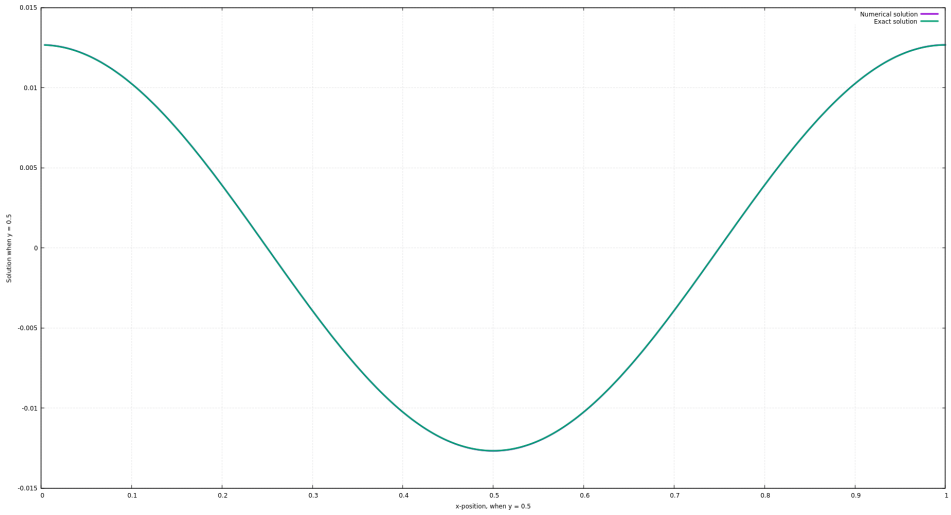

Figure 6.2: Analytical solution (green) vs numerical solution (purple) by SOR.

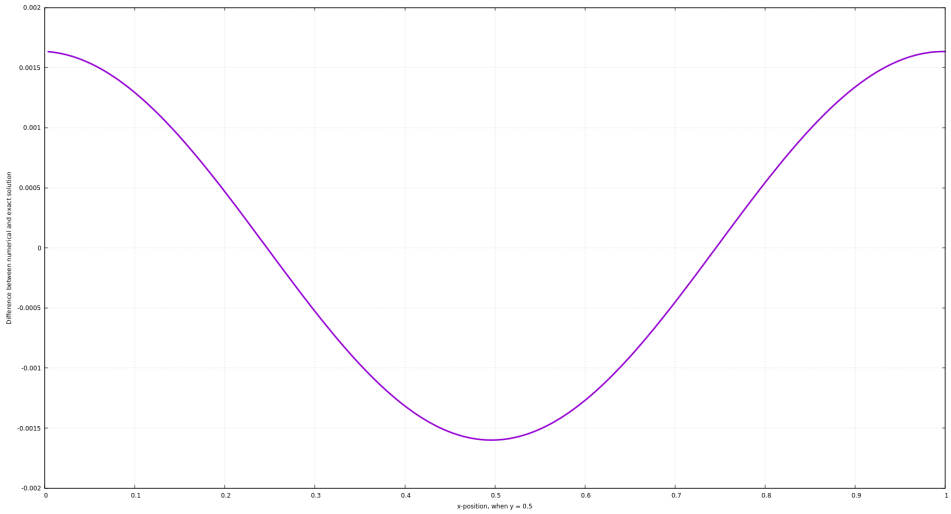

Figure 6.3: Difference between the analytical solution and Gauss Seidel.

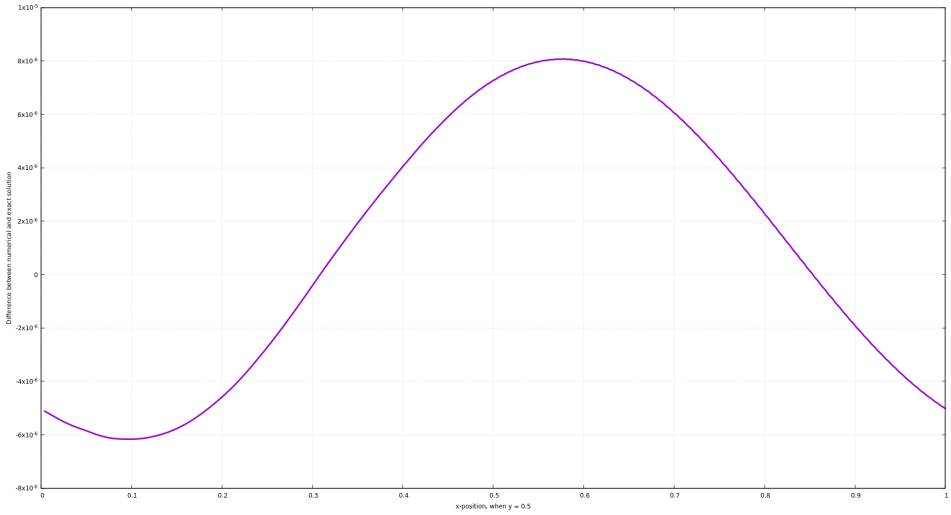

Figure 6.4: Difference between the analytical solution and SOR.

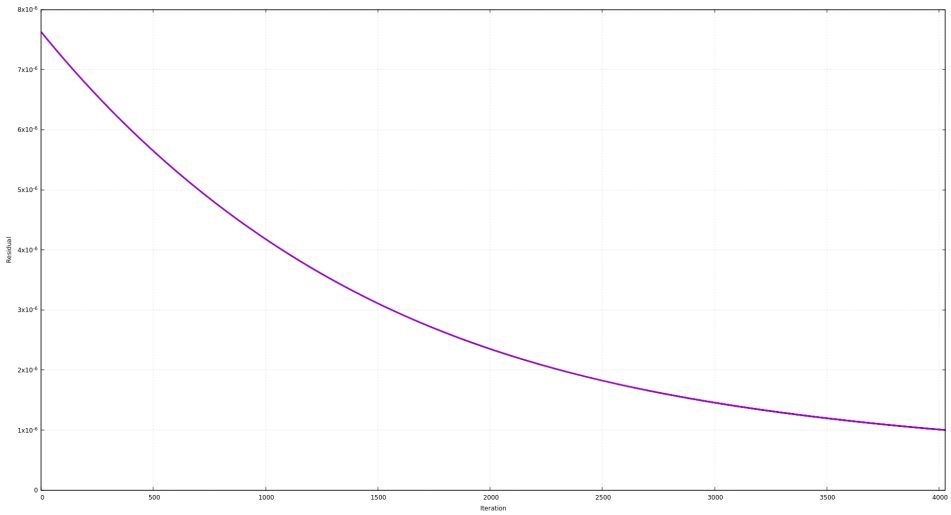

Figure 6.5: Change in residual per iteration for Gauss Seidel method.

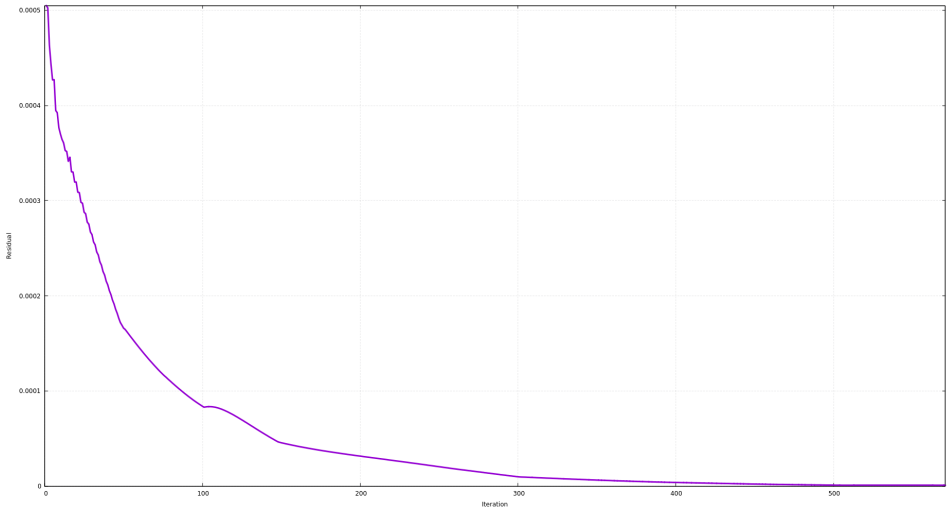

Figure 6.6: Change in residual per iteration for SOR.

6.1.2 Multigrid

A way to test whether a multigrid algorithm works properly or not, is to measure the residual before and after the algorithm goes to a coarser grid (Wesseling, 1995). A significant reduction in the residual should be found.

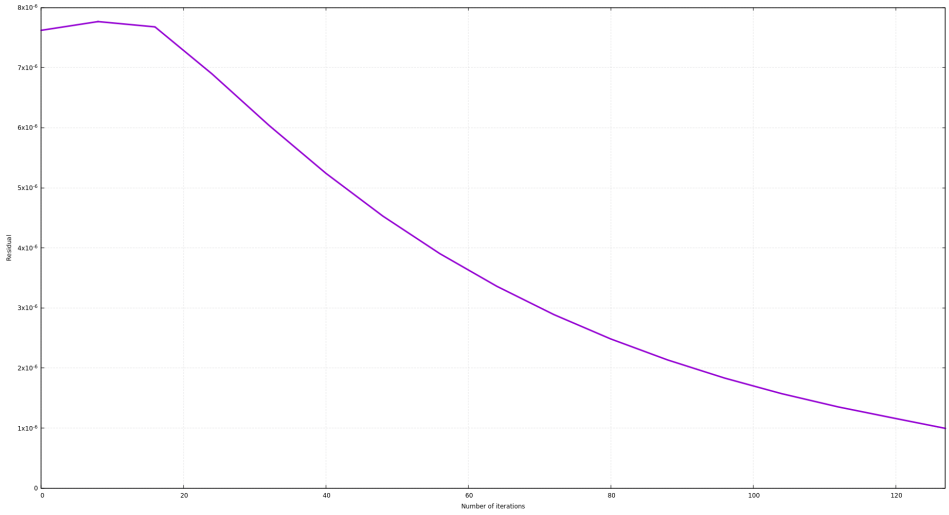

Figure 6.7: Change in residual per iteration for two-step MG.

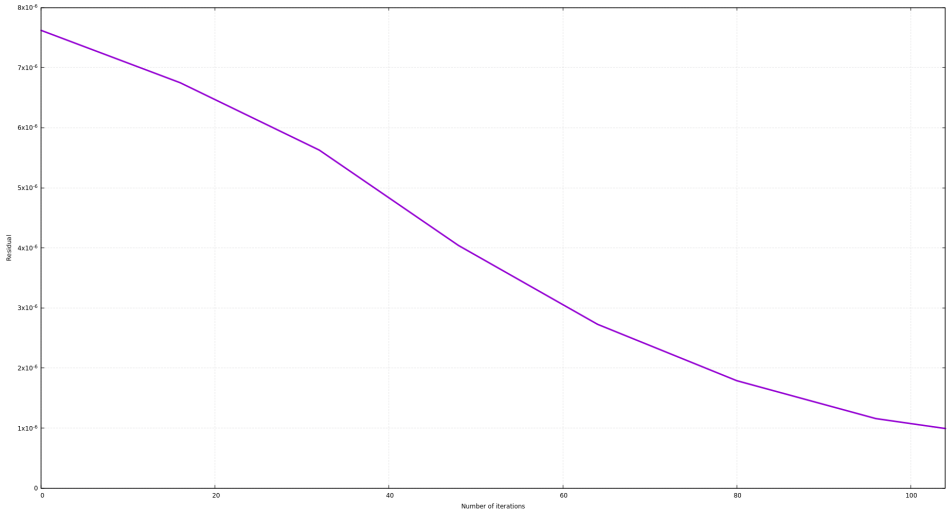

Figure 6.8: Change in residual per iteration for v-cycle MG when d = 3.

Figures 6.7, 6.8, 6.9 and 6.10 consists of plots of the residual reduction before an after the algorithm goes to a coarser grid, i.e. change in residual over a v-cycle. The number of iterations is measured depending on the depth in the v-cycle. The results coincides with the theory presented in chapter 3.

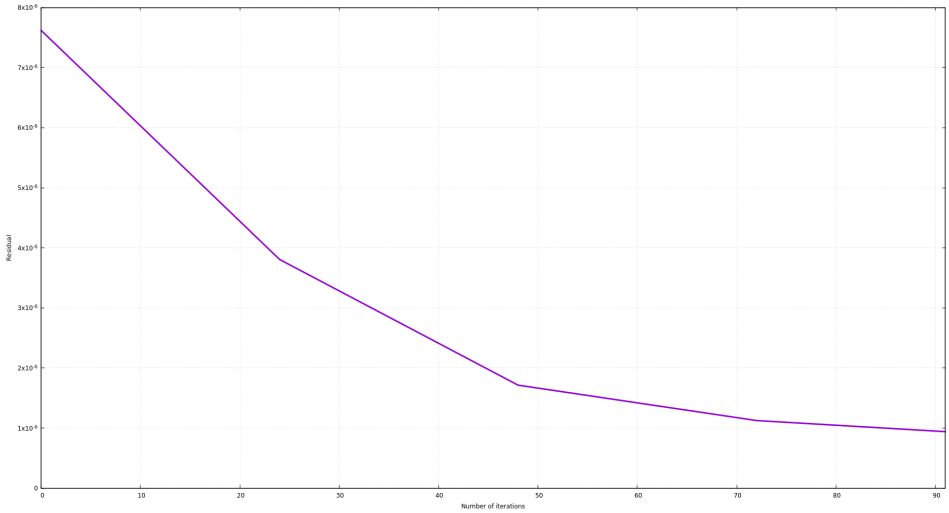

Figure 6.9: Change in residual per iteration for v-cycle MG when d = 4.

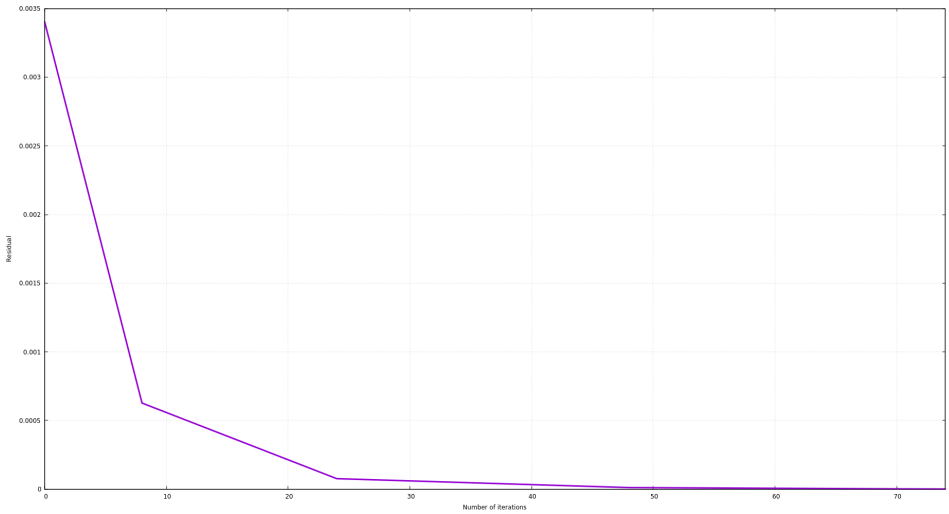

Figure 6.10: Change in residual per iteration for full MG.

It is possible to see that the change in residual deceases faster in the beginning of the iteration scheme when the depth in the v-cycle increases. This coincides with the reduction in high frequencies error for each grid step, i.e. transfered to lower frequencies on the fine grid, and deeper v-cycles will reach the convergence limit faster.

For the full multigrid algorithm, which starts on the coarsest grid, the reduction in residual is significantly higher in the beginning then for the v-cycle, but it also starts with a higher residual. By the algorithm, the method will continuously improve the initial guess at each depth, and quickly reduce the slow low frequent components of the fine grid. Independently of when the convergence criteria is reached, the algorithm applies v-cycles at each depth until the solution is back on the fine grid.

In a multigrid algorithm, the *projection* and *restriction* are sensitive parts. If the reduction in the residual is not found, these parts are not functioning properly and should be investigated (Wesseling, 1995).

To decide the number of pre-, coarse- and post-iterations, different iteration testes are made. The optimal iteration number is stated from the run-time for a converged solution. As a default case, pre-iteration $\nu_1 = 2$, coarse-iteration $\nu_2 = 50$ and post-iteration $\nu_3 = 8$. Figure 6.11 is a plot of the change in residual over number of cycles when changing ν_1. The optimal choice for ν_1 is 2. Figure 6.12 is a plot of the CPU-time used for different number of iterations where the optimal choice of ν_3 is 8. For v-cycle multigrid and full multigrid, different number of coarse-iterations are optimal. From plots on figures 6.13 and 6.14, it can be seen that ν_2 for v-cycle multigrid should be 35 and for the full multigrid ν_2 should be 15, with respect to the CPU-time. These iteration values are used in all future work.

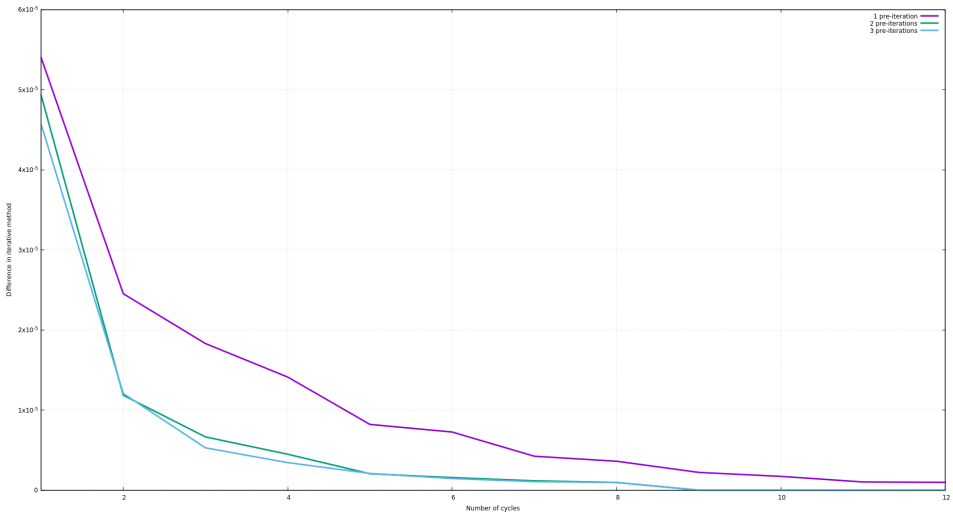

Figure 6.11: Pre-iteration test, iterations - 1:purple, 2:green, 3:blue.

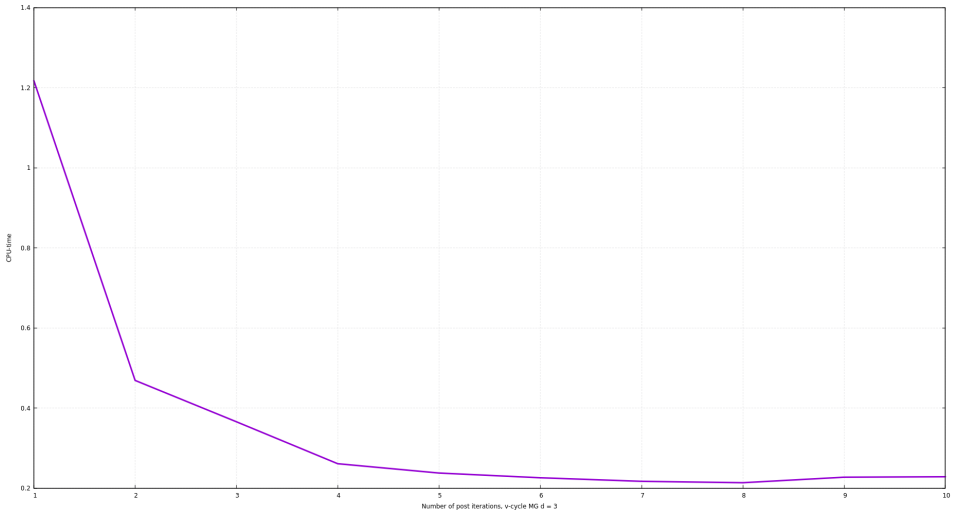

Figure 6.12: Post-iteration test.

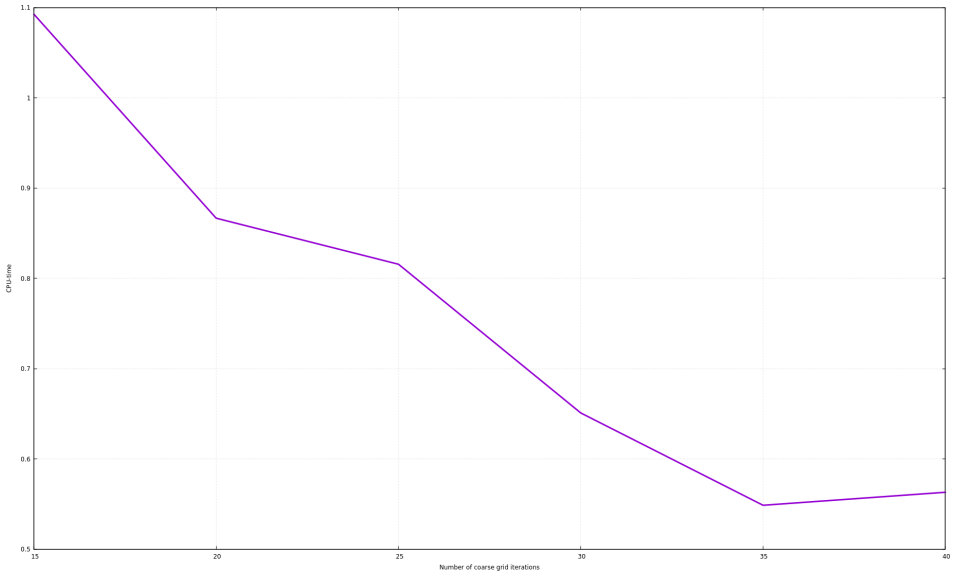

Figure 6.13: CPU-time vs number of coarse iterations, v-cycle MG.

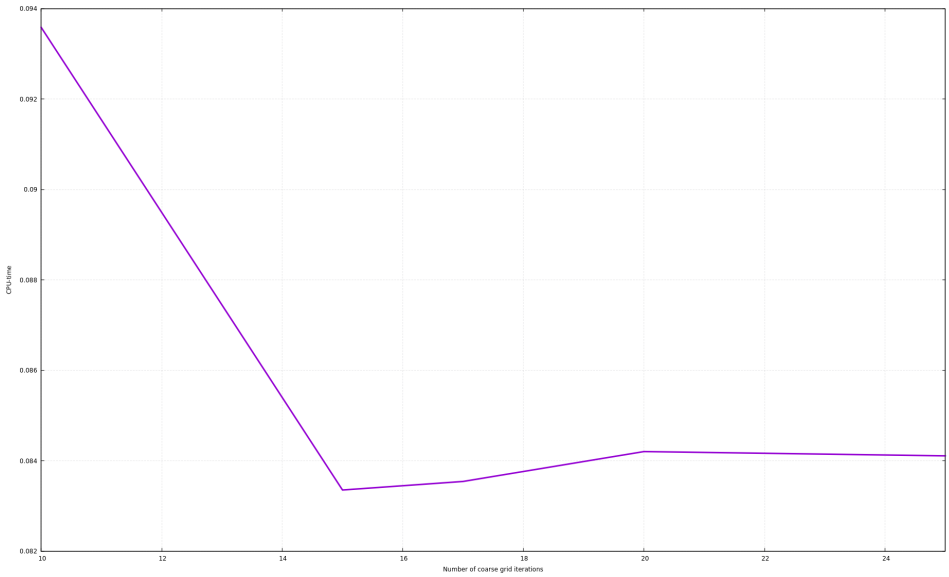

Figure 6.14: CPU-time vs number of coarse iterations, full MG.

The multigrid methods, figure 6.17 and 6.15, are less accurate then the SOR method, but more accurate then the Gauss Seidel method. The difference from the analytical solution, equation 6.2, to the numerical solutions are still propagating asymmetrically trough the domain, figure 6.18 and 6.16, see the accuracy validation of the Gauss Seidel method for more details. The same way of finding solution values and difference as prescribed under 6.1.1 yields for these plots.

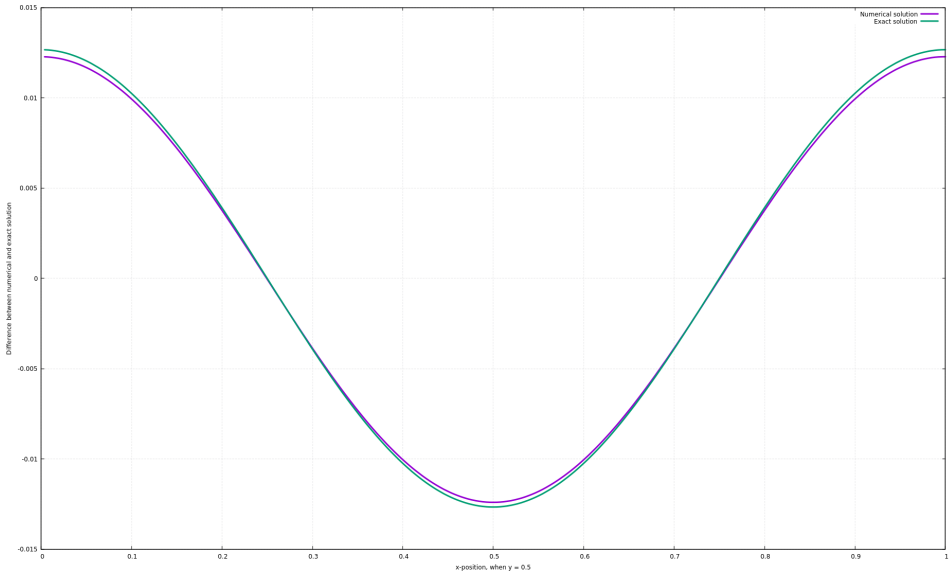

Figure 6.15: Analytical solution (green) vs numerical solution (purple) by v-cycle MG with d = 3.

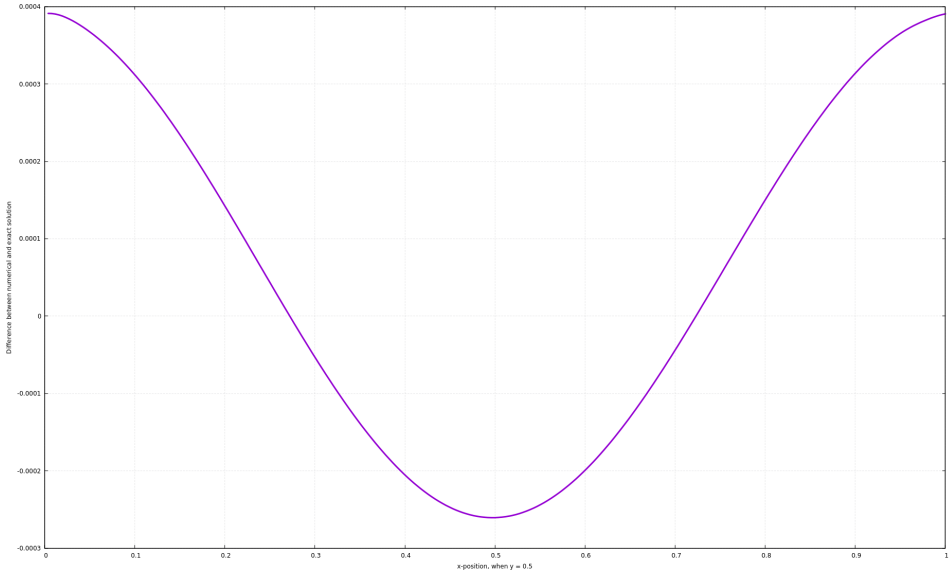

Figure 6.16: Difference between the analytical solution and numerical solution of v-cycle MG with d = 3.

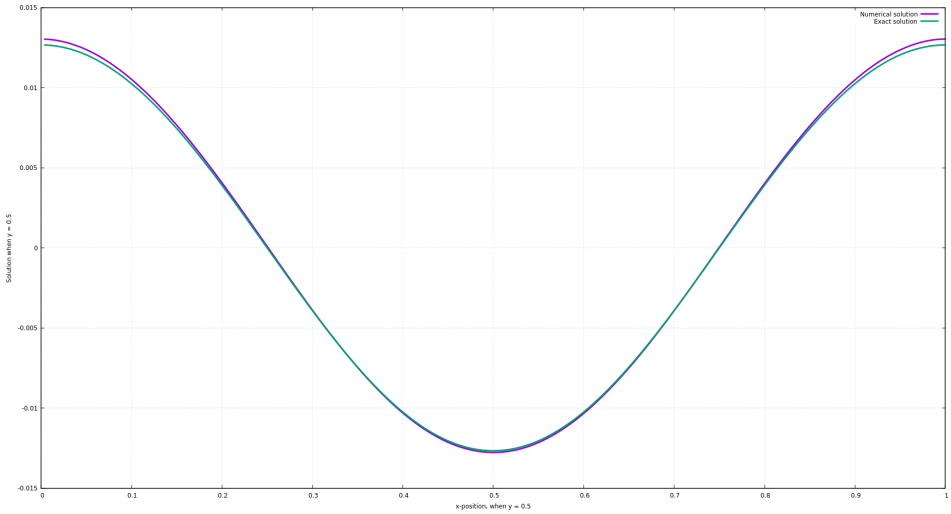

Figure 6.17: Analytical solution (green) vs numerical solution (purple) by full MG.

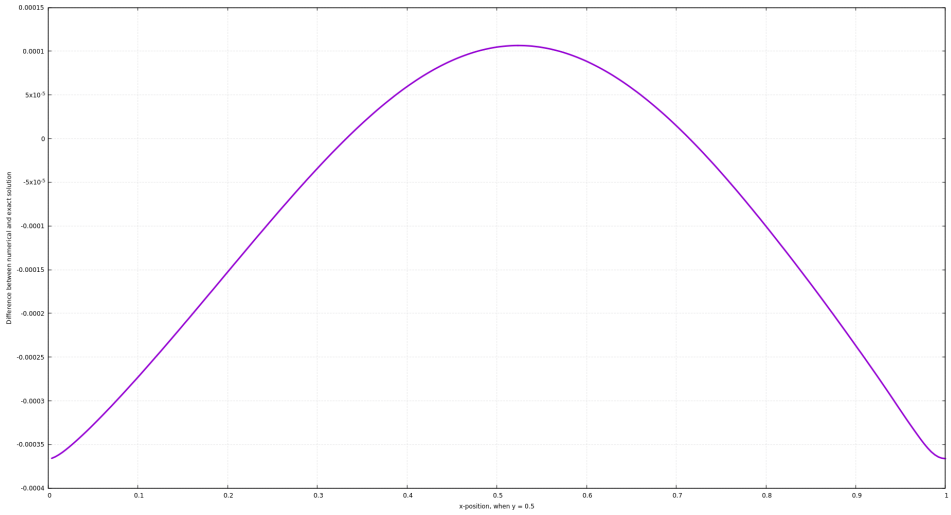

Figure 6.18: Difference between the analytical solution and numerical solution of full MG.

6.1.3 Comparison

To compare the different methods used to solve the Poisson equation, number of iterations and the speed-up factor is presented. The speed-up factor is calculated basted on the measured CPU-time for a method with respect to the measured CPU-time for the Gauss Seidel method, see equation 6.3.

$$S = \frac{(CPU - time)_{GS}}{(CPU - time)_{method}} \tag{6.3}$$

For a multigrid algorithm, the iterations is counted only for post-iterations on a upward stoke. This is an approximation based on the cost done to preform an iteration on a coarser grid compared to the finest grid. Since an upward stoke consists of coarser grids too, all iteration counts are merged together to count on just one stoke (upward, because it have most iterations). Therefore, the full multigrid iterations will count less then the iterations done on a v-cycle, i.e. the full multigrid has maximum 8 iterations on the finest grid per time-step. Together with the speed-up factor, this will give a good enough result of the efficiency relation between the different methods.

The plot of the speed-up factor, 6.19, and the plot of iterations, 6.20, have coinciding but opposite (ranging of the methods) results.

The Gauss Seidel method is clearly the slowest method, as expected. SOR will generally speed up the run-time 2-3 times and the two-step multigrid approximately 5 times, independent of number of grid points in the domain. V-cycle multigrid with depth more then two and the full multigrid method are dependent on number of grid points in the domain. On very coarse grids, they are not working optimal, therefore $h > \frac{1}{64}$ when using multigrid. The speed-up factor for the v-cycle algorithm will stagnate at approximately 15 for depth equals to three and 33 for depth equals to four. In a v-cycle it is not adequate to go

deeper then four depths. Therefor, in this solver it is possible to conclude with a maximum speed-up factor on 33 for the v-cycle algorithm.

For the full multigrid algorithm, the speed-up factor vary even more depending on number of grid cells. The reason for this is that the number of iterations in a full multigrid algorithm is independent of number of grid cells in the domain, distinct from the other methods (especially Gauss Seidel). Hence, it is clearly the fastest method to solve the Poisson equation. For the accuracy in the full multigrid algorithm, the RMS of the error in the solution domain will stabilize on an acceptable level when number of grid cells in the domain increases, according to plot 6.21.

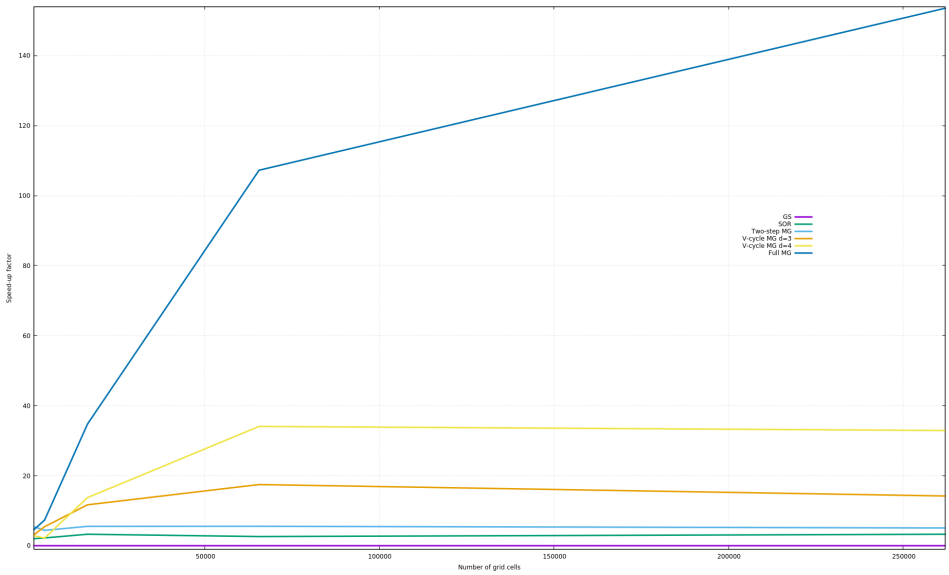

Figure 6.19: The speed-up factor with respect to Gauss Seidel method. Purple: GS, green: SOR, light blue: two-step MG, orange: v-cycle d=3 MG, yellow: v-cycle d=4 MG and dark blue: full MG.

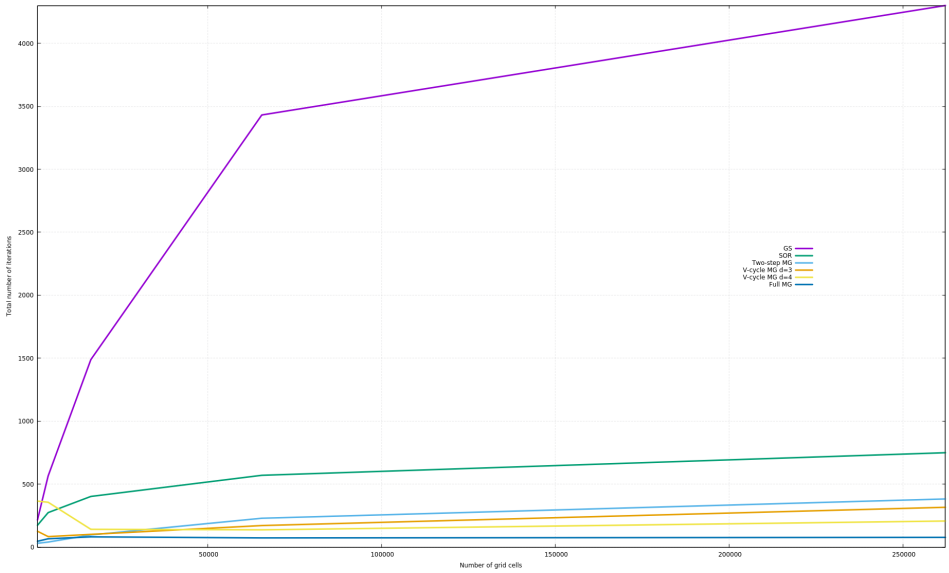

Figure 6.20: Total number of iterations. Purple: GS, green: SOR, light blue: two-step MG, orange: v-cycle d=3 MG, yellow: v-cycle d=4 MG and dark blue: full MG.

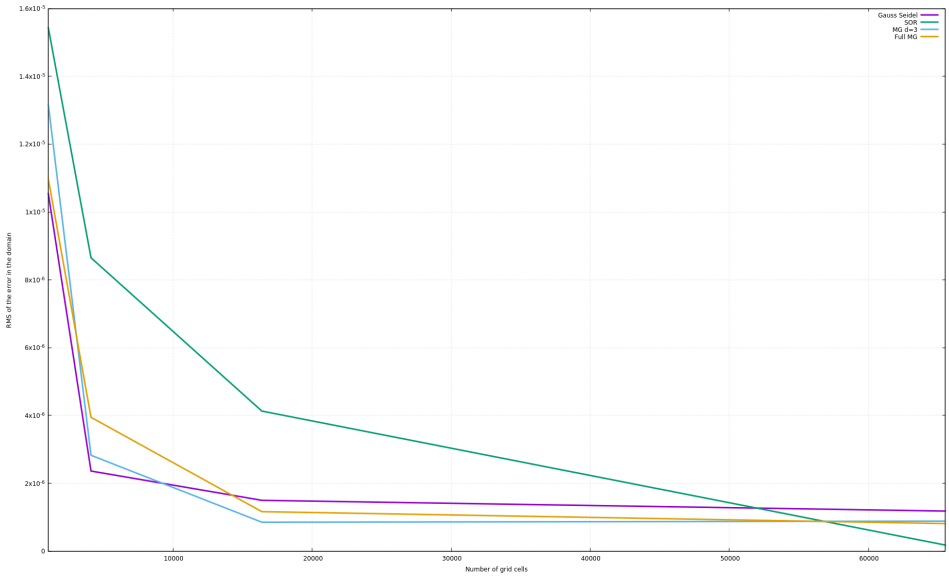

Figure 6.21: RMS of the error in the domain for different number of grid cells. Purple: GS, green: SOR, light blue: v-cycle d=3 MG and orange: full MG.

6.2 Navier Stokes Solver

Several refinement tests are done to optimize the numerical result on the Navier-Stokes solver. The model in the refinement tests should be as similar to the actual test case as possible i.e. same geometry, fluid properties etc. Force coefficients for lift and drag are measured to compare the test results and to chose the the most proper refinement. The lift coefficient is measured as the maximum value over 10 periods at the end of the time-period. The drag coefficient is measured at the arithmetic average over 10 periods at the end of the time-period. Flow past a cylinder is a traditional fluid dynamic problem, and is therefore chosen as a test case. It is easy to find published experiments to compare result values with. The von Karmen vertex street in the wake of a cylinder has some characteristics which are useful to validate together with the forces and Strouhals number when vary the Reynolds number.

Default case

The default test case is a channel flow with slip conditions at the walls, Dirichlet condition on the pressure at outflow and inflow velocity equals to 1.0 m/s, see figure 6.22 for more details. The domain Ω is [0:2]x[0:1]. The immersed boundary is stated by 50 points to create a proper circle and the diameter in the circle is equal to 0.1 and consists of approximately 12 cells. Reynolds number 20 and 100 are used to test the flow-field. The time-step Δt is set to meet the stability criteria, i.e. $\Delta t_{Re=100} = 0.001$ and $\Delta t_{Re=20} = 0.0003$, and a time-series lasts to 30 sec ($tmax = 30.0$). All tests are done with the default test case when nothing else is specified. If any of the variables are specified later, the new value is the valid one.

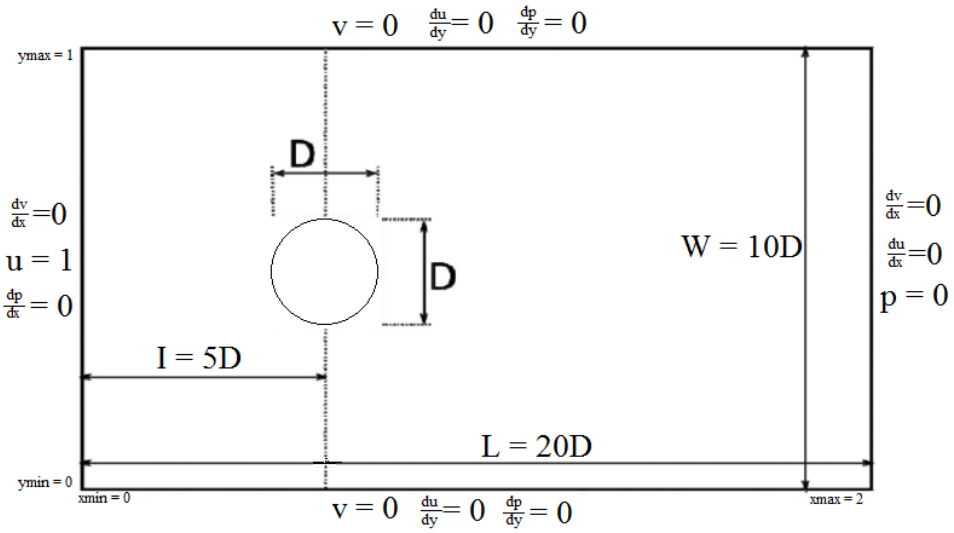

Figure 6.22: The computational domain of the default case with BC's and domain size.

6.2.1 Time Refinement Test

A time refinement test is done for a time dependent problem to check the dependency between the time resolution and accuracy. Several tests are presented in table 6.1.

Table 6.1: Refinement of time.

Re	Δt	C_L	C_D
20	0.001	-	unstable
	0.0003	-	2.281875
	0.0001	-	2.206216
	0.00007	-	2.188227
	0.00005	-	2.193227
100	0.001	0.249477	1.444527
	0.0003	0.16241	1.378774
	0.0001	0.141969	1.361831
	0.00007	0.139043	1.359352
	0.00005	0.137111	1.357709

The accuracy of the result is dependent of the time resolution in the solver. An explicit Euler scheme has a small stability area, hence considered as a less stable and more time-step restricted, i.e. need smaller time-steps, then other higher-order methods. All the tested Δt for Re = 100 in table 6.1 are in the stable area for explicit Euler according to the stability analysis, but other factors may also influence the accuracy. Explicit Euler is a first-order accuracy method, thereby the result was expected and Δt needs to be taken into consideration and may never converge. The optimal Δt for the final test case should be chosen according to previous published experimental results and closes to the limit set by the stability criteria in the stable area.

6.2.2 Mesh Refinement Test

When using the FDM method, the accuracy of the solution is linked to the mesh size in the domain. When the mesh size decreases towards zero, the solution is moving toward the exact solution of the equation. Some limitations here are the time and limited computational resources. Therefore, it is important to find the optimal mesh size with an mesh refinement test where the goal is to minimize the error to a acceptable level depending on the analyzing goals.

In the mesh refinement tests, the mesh size is tested against the diameter of the cylinder, $\frac{D}{h}$, where $h = \Delta x = \Delta y$, i.e. an equidistant grid is kept. The test case setup is similar to the default case, but some changes are done for Δt. Only Re = 100 is tested for because of the high rate of changes in the velocity and pressure gradients trough the domain.

Table 6.2: Refinement of number of cells in cylinder diameter.

Re	h	Δt	$\frac{D}{h}$, cells	C_L	C_D
100	$\frac{1}{64}$	0.006	6	0.119433	1.765178
	$\frac{1}{128}$	0.003	12	0.260477	1.498631
	$\frac{1}{256}$	0.0001	24	0.265365	1.497785

Table 6.3: Accuracy of the Poisson solver on domain [0:2]x[0:1].

Number of cells (2nxn)	RMS of domain
8192 (n=64)	$1.8068 \cdot 10^{-6}$
32768 (n=128)	$7.33725 \cdot 10^{-7}$
131072 (n=256)	$5.35976 \cdot 10^{-7}$

Another test is done to check the accuracy against an analytical solution, equation 6.2. To establish this test, the test problem from section 6.1 is used and the Poisson solver is tested. The difference between the numerical result and the analytical solution is calculated by a L2-norm (RMS) of the domain [0:2]x[0:1]. The results are presented in table 6.3.

Table 6.2 shows that the numerical results are dependent of the mesh resolution. The RMS of the solution differences in the domain are acceptable for all the tests. From table 6.2, $\frac{D}{h} = 12$ is has two decimals equals to $\frac{D}{h} = 24$ for the force coefficients. Due to run-time of the time-series and the small difference in the solution coefficients, $\frac{D}{h} = 12$ will be used in the final test case.

6.2.3 Domain Refinement Test

Different domain refinement tests are preformed to find the optimal refinement of the domain length (L), domain width (W) and the cylinder distance from the inflow (I). The results of the lift and drag coefficients for different L and W are presented in table 6.4 and table 6.5.

The force coefficients are not affected significantly by changing in the domain length. For Re = 20, this is as expected since no vortex shedding occur. For Re = 100, this is more unexpected because of the large vortex velocity gradients that follows the von Karmen street downstream in the domain. The resolution of the vortex shedding is important, otherwise an unstable von Karmen street may distract the pressure distribution around the cylinder and effect oscillation of the drag force. L = 20D is the longest possible construction of the domain and will be used for the final test case. It may turn out to not be long enough.

Table 6.4: Refinement of domin length.

Re	L (length)	C_L	C_D
20	12.5D	-	2.605942
	15D	-	2.605528
	17.5D	-	2.605466
	20D	-	2.605461
100	12.5D	0.255942	1.444761
	15D	0.25048	1.444593
	17.5D	0.249475	1.444526
	20D	0.248281	1.444507

In the domain refinement tests, the force coefficients are significantly affected by the change in the domain width. The coefficients are not converged properly, but W = 17.5D will be used in the final test case and is the larges possible construction of the domain in the CFD-solver together with L = 20D.

Table 6.5: Refinement of domin width.

Re	W (width)	C_L	C_D
20	10D	-	2.605461
	12.5D	-	2.515492
	15D	-	2.461674
	17.5D	-	2.426802
100	10D	0.248281	1.444507
	12.5D	0.244023	1.407576
	15D	0.240576	1.386462
	17.5D	0.238610	1.373361

For the inflow refinement test, the length of the domain, L, is changed to 20D. This is done with respect to the length refinement test and are constant during domain inflow refinement test. The x-position in the cylinder center will vary depending on I and the length behind the cylinder will change between 13D-16D. If the cylinder is placed to close the inflow, the numerical solutions will be affected by the constant inflow velocity. The converged inflow coefficients are found for I = 6D, and will be used in the final test case.

Table 6.6: Refinement of domin inflow.

Re	I (inflow)	C_L	C_D
20	4D	-	2.443131
	5D	-	2.426646
	6D	-	2.425538
	7D	-	2.425098
100	4D	0.310410	1.490689
	5D	0.240358	1.386235
	6D	0.238520	1.377452
	7D	0.238758	1.376718

The domain refinement test are strongly dependent of the Reynolds number

because of the von Karmen vortex street. For higher Reynolds numbers, the test should be reverse.

6.2.4 Multigrid together with IBM

A final test case is made to test and analyses the results provided by a solver which used a multigrid method together with an IBM. To summarize the final test setup:

$\mathbf{L}$ = 20D $\quad$ $\mathbf{W}$ = 17.5D $\quad$ $\mathbf{I}$ = 6D

$\Delta\mathbf{t_{Re=100}}$ = 0.001 $\quad$ $\frac{\mathbf{D}}{\mathbf{h}}$ = 12 $\quad$ $\mathbf{h} = \frac{1}{128}$

$\Delta\mathbf{t_{Re=20}}$ = 0.0003

The Poisson equation is solved with a full multigrid algorithm.

Strouhals number is a dimensionless frequency of the vortices which are shed from the body, given in equation 6.4. $f_v = \frac{1}{T_v}$ is the vortex shedding frequency, T_v is a dimensionless time period, D is the diameter of the body and U_∞ is the inflow velocity.

$$St = \frac{D f_v}{U_\infty} \tag{6.4}$$

Table 6.7: Result from the final test case at Re = 20.

Result source	C_D
Present	2.123974
Taira and Colonius (2007)	2.07
Park et al. (1998)	2.01
Xu and Wang (2006)	2.23

Table 6.8: Result from the final test case at Re = 100.

Result source	C_L	C_D	Strouhals number
Present	0.171374	1.281555	0.182
Lai and Peskin (2000) scheme 1	0.3290	1.4630	0.144
Lai and Peskin (2000) scheme 2	0.3299	1.4473	0.165
Park et al. (1998)	0.3321	1.33	-
Xu and Wang (2006)	0.34	1.423	0.171

Table 6.7 present the drag-coefficient at Re = 20. The coefficient for the present CFD-solver is lying in between the results from previous studies. Visualizations associated with the test are as expected and presented in figure 6.23 and 6.24.

For the test with Re=100, several parameters are checked and presented in table 6.8. The Strouhals number is larger then from the other studies, but in an acceptable range. For the lift- and drag- coefficients, the previous studies presents much higher coefficients. As seen for the refinement tests, the coefficient values are more similar to those from previous studies. From the default case, only the domain size has been changed and the domain width had large impact on the coefficients. More thorough analyses should therefor be done on the domain before any conclusions are made. The visualizations of the pressure, velocities and streamlines, see figures 6.25 and 6.26, are behaving as expected during the time-series.

The visualization defects of the velocities done in the post-processing stage should be taken into consideration when analyzing figures 6.24 and 6.26. This is described in subsection 5.8.

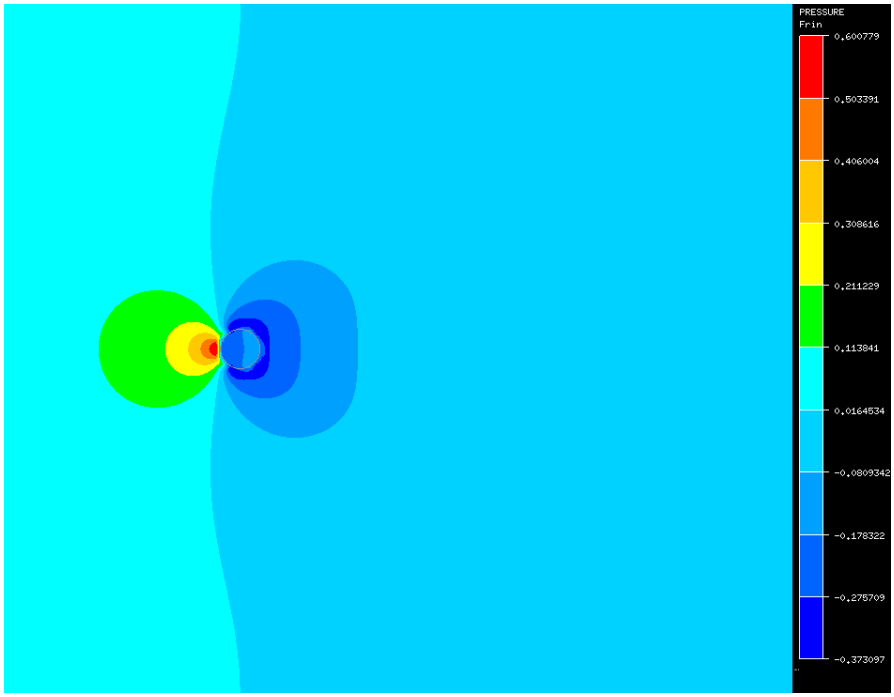

Figure 6.23: Pressure contours at Re = 20.

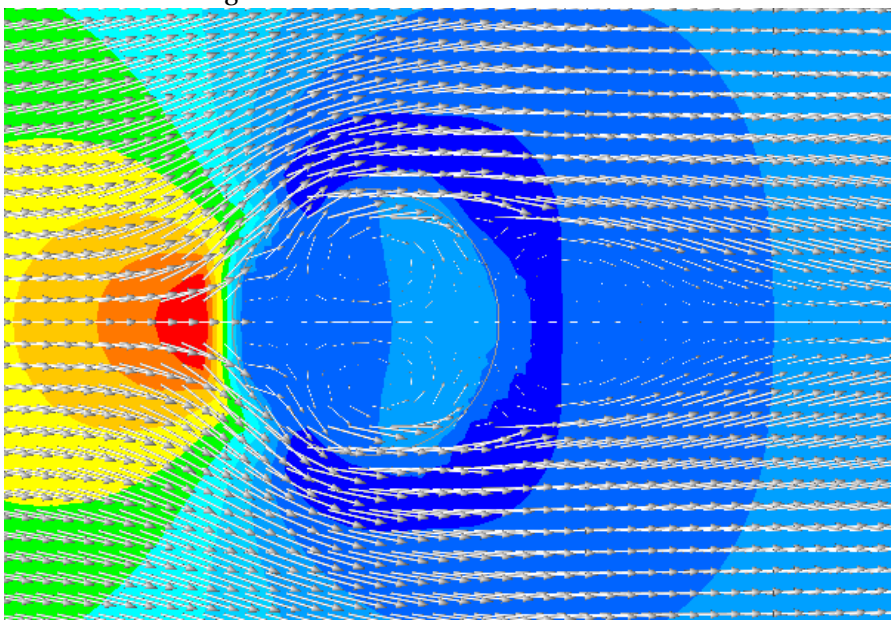

Figure 6.24: Pressure contours and velocity arrows at Re = 20.

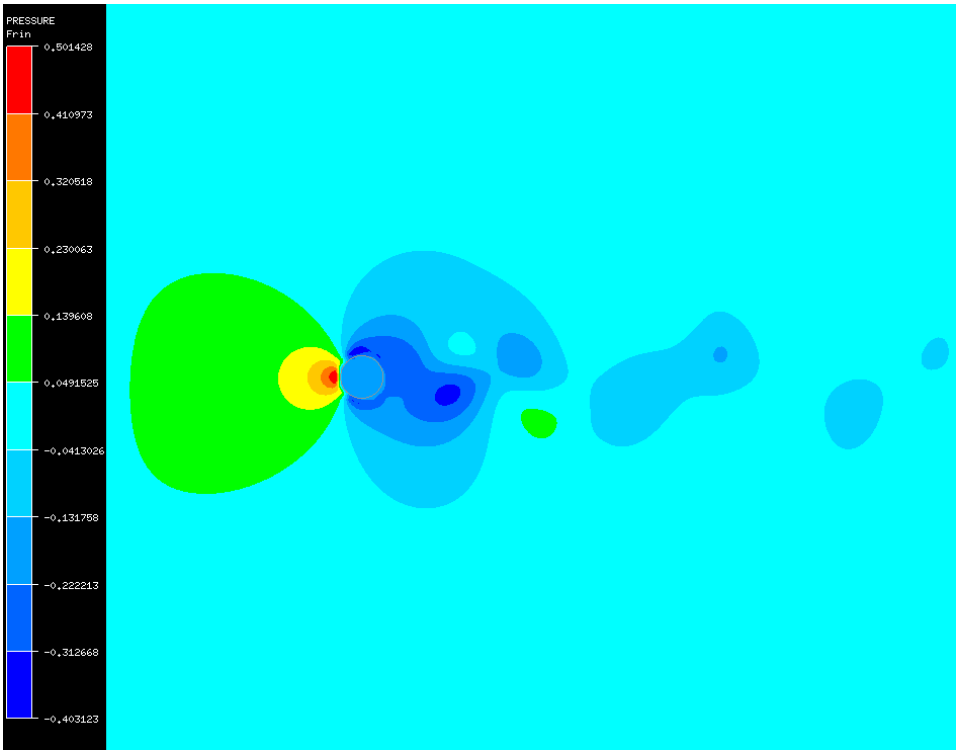

Figure 6.25: Pressure contours at Re = 100.

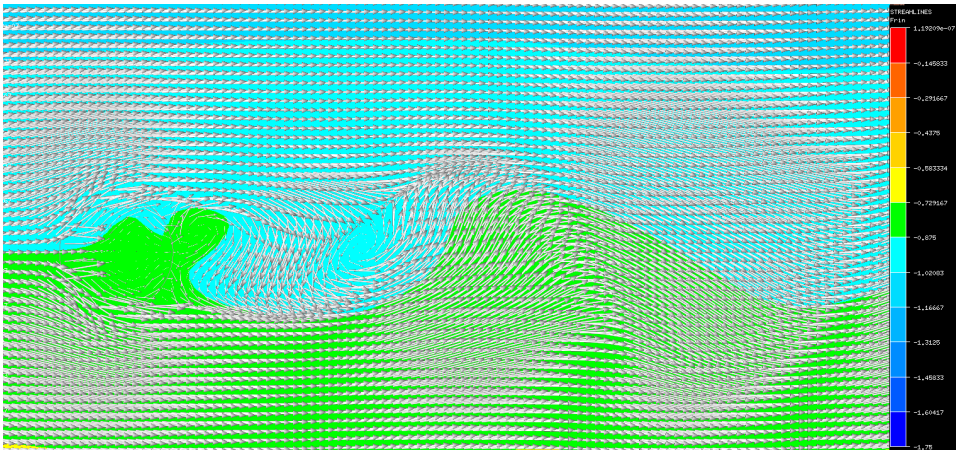

Figure 6.26: Streamlines and velocity arrows at Re = 100.

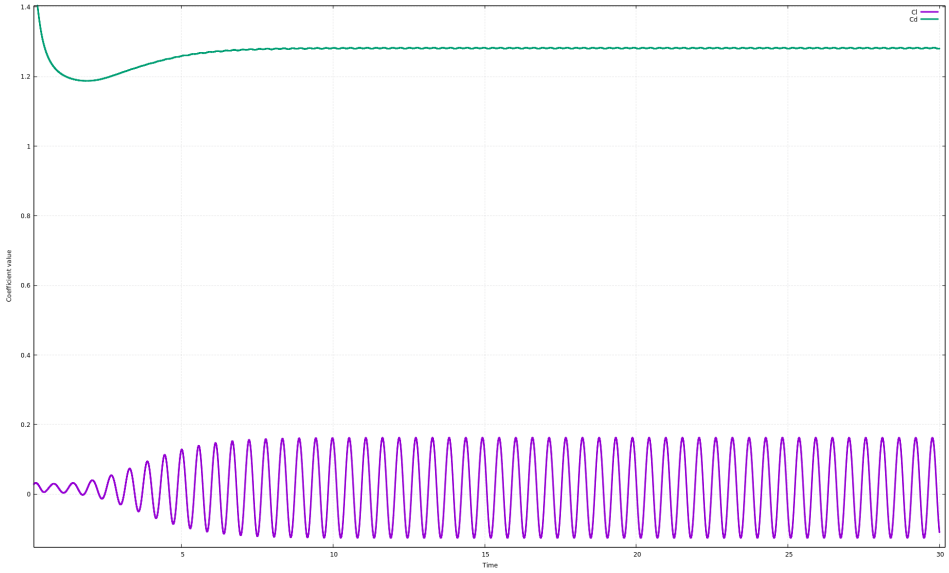

Figure 6.27: Drag (green) and lift (purple) coefficients at Re = 100.

The displacement found in the Poisson solver for the Gauss Seidel algorithm generate a small contribution to the lift coefficient at Reynolds number 20. It may provide some contribution to the other coefficients too, but it is easier to notice in the lift coefficient calculations since it should have been zero. The produced error only differ 0.57% and will be neglected in all parts of the problem analysis.

6.2.5 Efficiency analysis

To analyses which part of the Navier Stokes solver who vast most of the running time, a time break-down of one time-step is presented for the final test case. Table 6.9 present the different parts in a Navier Stokes solver where the Gauss Seidel method is used in the Poisson solver and table 6.10 present the different parts in a Navier Stokes solver where a full multigrid algorithm is used in the Poisson solver. For both cases, it can be seen that the Poisson solver is the most decisive part, but there is an extreme difference by changing to full multigrid in the deflection on the print-out procedure. To illuminate, the print-out procedure will not run every time-step, only for visualization of the numerical result.

Table 6.9: Time break-down when using Gauss Seidel.

Solution process	% of the computational time
Poisson solver	99.9%
IBM solver	0.0%
Print-out procedure	0.08%
Other	0.02%

Table 6.10: Time break-down when using full multigrid.

Solution process	% of the computational time
Poisson solver	75.6%
IBM solver	0.3%
Print-out procedure	21.8%
Other	2.3%

Chapter 7

Conclusion and Recommendations for Further Work

7.1 Conclusions

To increase the calculation efficiency in a CFD-solver, investigation of the Poisson solver was mentioned. The vast majority of the run-time is spent at this process, and by solving it on a final test case with the iterative scheme Gauss Seidel minimum 99.9% of the computational time was spent here. A full multigrid method was developed and decreased the run-time of the Poisson solver with a factor of 110 (on a domain with 65536 grid-cells) compared to Gauss Seidel on a simple test case. The speed-up factor are related to the mesh resolution, but increases when the number of grid-cells increases. Consequently, the run-time spent on solving the Poisson equation in the CFD-solver on a final test case was reduced to minimum 75.6% with the full multigrid algorithm.

The CFD-solver with a full multigrid method was validated with an FSI problem solved by IBM. Force coefficients acting on a cylinder in a flow with Reynolds number 20 and 100 where used as reference values in several refinement tests. The accuracy of the resolution in the domain is significantly dependent on the size of Δt in the numerical simulation. Explicit Euler has a small stability area which force Δt to be quite small (depending on the problem) to provide reliable numerical results.

Areas of large velocity or pressure gradients need to have fine enough mesh resolution, e.g. areas close to the buff cylinder and in the von Karmen vortex street. Together with a large domain compared to the cylinder diameter, width should be at least 17.5 times the diameter and the length should be at least 20 times the diameter, the resolution is kept for Re = 20. The domain have a huge impact on the calculated force coefficients, so more thorough analysis should therefore be done on the domain to validate the numerical solution for Re = 100. In the next section some recommendations for further work and changes are provided to optimize the solver.

7.2 Recommendations for Further Work

In this section some recommendations for future work and changes are provided to optimize the solver.

7.2.1 Short-Term

One of the most important criteria for a CFD-solver is to provide accurate results in a certain time limit. To achieve that criteria for the present CFD-solver, the development of the domain and how to place the geometry in the flow should be reversed. A *larger domain* in both (x- and y-) directions are preferred. To get a more versatile code, the IBM module should be validated for *moving geometries* and expand into a *full fluid structure interaction* solver.

Different experimentations consisting of changing the transformation schemes in multigrid to increase the efficiency could be done. The restriction procedure could for example be changed to *full-weighted* restriction, explained under chapter 3.

A part of the code which have potential for efficiency and accuracy increase of the numerical simulations without to much effort, is the temporal-discretization scheme. Explicit Euler is restricted to so small Δt for this problem, hence a *higher order Runge-Kutta scheme*, see subsection 2.3.2, would use much less computational time.

7.2.2 Long-Term

If the CFD-solver shall work for turbulent or transitional flow, the resolution requires *3D simulations*. To capture all scales of motions Δx_i and Δt needs to be so small, hence a *super computer* and *parallelization* of the code is required. A CFD-solver with multigrid methods are highly parallelizable because each pro-

cesses can simultaneously be performed at all grid points. For more details of parallel multigrid processing see Brandt (1981).

To reduce the unnecessary computational effort preformed in the numerical simulation, *adaptive grid refinement* (subsection 2.2.2) should be included in the CFD-solver.

Appendix A

Acronyms and Symbols

Acronyms

CFD Computational Fluid Dynamics

IBM Immersed Boundary Method

SOR Successive Over Relaxation

GS Gauss Seidel

FSI Fluid Structure Interaction

MG Multigrid

GMG Geometric Multigrid

AMG Algebraic Multigrid

FMG Full Multigrid

FSI Fluid Structure Interactions

BC Boundary Condition

1D One-Dimension

2D Two-Dimensions

PM Projection Method

ODE Ordinary Differential Equation

PDE Partial Differential Equation

CFL Courant-Friedrichs-Lewy

FDM Finite Difference Method

FVM Finite Volume Method

FEM Finite Element Method

FTCS Forward-Time Central-Space

MAC Marker and cell

flops Floating Point Operations

RMS Root-mean square

CPU Central Processing Unit

Symbols

d Depth of an multigrid

n Number of grid points

λ Eigenvalue

Ω Computational domain

$\partial\Omega$ Boundary of domain

Ω_b Domain occupied by geometry

$\partial\Gamma_b$ Boundary of geometry

$\mathbf{f_b}$ Forcing function

h Grid cell-size, equidistant

Δx_i Grid cell-size in x-direction (i=1) and y-direction (i=2)

t Time

Δt Time-step

x_i Cartesian coordinates, (x, y)

u Velocity in x-direction

v Velocity in y-direction

p Pressure

Δp Change in pressure

u^* Predicted velocity in x-direction

v^* Predicted velocity in y-direction

i Position in x-direction in a Cartesian grid

j Position in y-direction in a Cartesian grid

ρ Density

ν Kinematic viscosity

Re Reynolds number

St Strouhals number

$\mathscr{O}$

$\mathscr{R}$

n Normal vector

ω Damping coefficient

ω_{opt} Optimal damping coefficient

C Courant number

C_{max} Criteria for CFL-condition

I_h^{2h} Restriction operator/matrix

I_{2h}^h Prolongation operator/matrix

k Iteration counter

$\boldsymbol{e}$ Error

r Residual

f Right-hand side of a linear system

A Coefficient matrix in a linear system

s Solve with iteration

e Solve exact or with iteration

r Restrict the system

p Prolongate the system

γ Number of cycles in multigrid

υ_1 Number of pre-iterations

υ_2 Number of coarse-iterations

υ_3 Number of post-iterations

β convergence factor

$U_{i,j}$ Fluid velocity in a cell next to an immersed boundary

C_L Lift coefficient

C_D Drag coefficient

F_D Drag force

F_L Lift force

U_∞ Inflow velocity

f_v vortex shedding frequency

T_v vortex shedding period

I Inflow length in domain

L Domain length

W Domain width

Appendix B

Source code

Listing B.1: head.h, all functions.

```
#ifndef Header_Header
#define Header_Header
int main();
void FDMsolver(double **, double **, double **);
void linearSolver(double t, double **, double **, double **, double **deltaP);
void multigrid(double t, double **, double **, double **, double **deltaP);
void GaussSeidel(int imax, int nn, double **f, double **phi, int *iter, double *res,
     double *error);
void SOR(int nn, double **f, double **phi, int *iter, double *res, double *error);
void correctVelo(int nn, double **, double **, double **deltaP, double **uFlag,
    double **vFlag);
void presCorr(int nn, double **p, double **deltaP);
void velBCfield(int nn, double **, double **);
void pBCfield(int nn, double **);
double residual(int nn, double **f, double **phi, int i, int j);
void weightedResidual(int nn, double **RHS, double **phi, double **R);
void injectedResidual(int nn, double **RHS, double **phi, double **R);
void injected(int nC, double **R);
void weighted(int nC, double **R);
void interpolation(int nC, double **E);
void addValue(int nn, double **E, double **phi);
void presCorr(int nn, double **p, double **deltaP);
```

```
21 void solveCoarse(double **phi, double **f);
22 void startResidual(int nF, double **RHS, double **phi, double **R);
23 void stream(double **, double **, double **);
24 void results(double **, double **, double **, double **, double **, double **,
       double **, double **, double **);
25 void IC(int, double **, double **, double **);
26 void validatePoisson();
27 void stability(void);
28 double avg(int nn, double **matrix);
29 void CFL(double, double);
30 double maxNorm(int nn, double **matrix);
31 double RMS(int nn, double **x);
32 double sum(int nn, double **matrix);
33 void toZero(int dim, double **matrix);
34 void toZeroM(int dd, double ***matrix);
35 void arrayMatrix(int nn, double ***array, double **matrix, int dd);
36 void matrixArray(int nn, double **matrix, double ***array, int dd);
37 void copy(double ***from, double ***to, int dd);
38 void ccopy(double **from, double **to);
39 double interp(void);
40 double linearInterp(double nn, double x0, double x1, double y0, double y1);
41 int powerOf(int k);
42 int log_2(int n);
43 //---------- From the included file nrutil.c----------------------------//
44 void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch);
45 double **dmatrix(long nrl, long nrh, long ncl, long nch);
46 double ***f3tensorD(long nrl, long nrh, long ncl, long nch, long ndl, long ndh);
47 void free_f3tensorD(double ***t, long nrl, long nrh, long ncl, long nch,
48  long ndl, long ndh);
49 void nrutil();
50 #endif
```

Listing B.2: input.h, file to change input variables.

```
#ifndef SHAREFILE_INCLUDED
#define SHAREFILE_INCLUDED
#ifdef  MAIN_FILE
/*-------- Input variables ----------------*/
  const int n = 128;  //Numer of internal cells
  const double epsi = 0.000001; //Error
  const int Re = 100;   //Reynholds number --> (RealRe/2*radius)
  const double rho = 1.0; //Density
  const double nu = 0.05; //Kinematic viscosity
  const double tmax = 30.0; //End time
  const double dt = 0.00001; //Time stepsize
  const int itmax = 500; //Max inerations
  const int itPre = 2;  //Max inerations - PRE-smoothing MG
  const int itCoarse = 35; //Max iterations - COARSE-smoothing MG
  const int itPost = 8;  //Max inerations - POST-smoothing MG

/*------ Set wall type and velocity --------*/
  const int wall = 3;  //1:Lid driven top,2:Couette flow,3:Channel-velocity,
    //4:No-slip channel flow,5:Preiodic-free-slip,6:Poiseuille flow
  const double velX = 1.0; //Stream velocity - x direction[m/s]
  const double velY = 0.0; //Stream velocity - y direction[m/s]

  const double dimX = 1.0; //Scaling factor in x-dim
  const double dimY = 1.0; //Scaling factor in y-dim

  const int method = 2;    //1: Gauss Seidel, 2:SOR, 3:two-grid iteration 4:V-cycle
     multigrid, 5:Full-multigrid
  const int d = 0;  //Depth of multigird --> n = 512, 256, 128, 64, 32, 16, 8, 4, 2.
      d have to be at least 2.
  const int ncycle = 0;  //Number of repeated V-cycles in multigrid

/*------ IBM ------------------------------*/
  const int runIBM = 1;  //0:not run, 1:run
  const int calcForce = 1; //0:not calculate forses, 1: calculate forces
  const int bType = 3;   //1:random points as a simple polygon,
    //2:two lines-equal number of points (nPoint/2), 3:circle
  const int nPoint = 50; //Number of points to form the geometry
```

```
36
37    const int bMotion = 0;   //0: No body motion, 1: Body motion on
38    const int mDir = 1;   //0 = x-dir, 1 = y-dir of motion
39    const double bFreq = 6.28; //Motion frequency of the body, approx 2*pi
40    const double bAmp = 0.001; //Motion amplitude of the body
41    const double nMotion = 10; //Number of motion cycles.
42
43    //Max and min positions of the geometry in domain
44    const double xmin = 0.45;
45    const double xmax = 0.55;
46    const double ymin = 0.45;
47    const double ymax = 0.55;
48
49    const int maxIntersec = 500;
50    //Simple polygon or two lines, {x-coord},{y-coord}
51    const double pointPos[2][4] = {{0.0, 1.0, 1.0, 0.0},{0.90625, 0.90625, 0.09375,
        0.09375}};
52    //Cylinder
53    const double CG[2] = {0.5,0.5};//Mass center of cylinder
54    const double radius = 0.05;   //Radius of cylinder
55
56 /*------ Intervall for writing out to screen and vtf-file ----------*/
57    const int writeOut = 500; //Write out intervall for vtf-file
58    const int printOut = 1;   //Print out to terminal
59
60 //-------- Validation ---------------------------------------------//
61    const int validate = 0; //0:no validation 1:Poisson validation
62 #else
63    extern int n, Re, itmax, wall, nr, bType, nPoint, maxIntersec, method, runIBM, d,
        ncycle, itPre, itPost, itCoarse;
64    extern int bMotion, mDir, calcForce, fromYin, toYin, fromYout, toYout, writeOut,
        printOut, validate;
65    extern double epsi, tmax, dt, velX, velY, xmin, xmax, ymin, ymax, radius, rho,
        bFreq, bAmp, nMotion, nu, dimX, dimY;
66    extern double element1[5], element2[2], pointPos[2][50], CG[2];
67 #endif
68 #endif
```

Listing B.3: main.c, main file of the solver.

```
#include "head.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "nrutil.h"
#define MAIN_FILE
#include "input.h"

void printFile(double, double **, double **, double **, double [4][nPoint]);
void IBM(double t, double **, double **, double body[4][nPoint], double **uFlag,
     double **vFlag);
void setBody(double t, double body[4][nPoint]);
/*------------------ Main program ----------------------------*/
int main()
{
  /*------------------ Variable definition -------------------*/
  int saveOut, newIC, step, time;
  int  i, j, iter = 0, countW = 1, countP = 1, id;
  double omega, beta, div = 0.0, integer, divMax, diff;
  double **u, **v, **p, body[4][nPoint], fraction;
  double **uFlag, **vFlag, **deltaP;

  p = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  u = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  v = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  deltaP = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  uFlag = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  vFlag = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  /*---------- Check stability requirements ---------------*/
  stability();
  if (validate == 1){printf("VALIDATION \n");validatePoisson();}
  else
  {
  /*-- Set initial conditions for primitive-variables --*/
  IC(newIC, u, v, p);
  toZero(n, deltaP);
  toZero(n, uFlag);
```

```
37  toZero(n, vFlag);
38  velBCfield(n, u, v);
39  /*------------------ Time iteration, main loop ------------------------*/
40  for (double t = 0; t <= tmax; t = t+dt)
41    {
42      step = (int)(t/dt) + 1;
43      fraction = modf(step/printOut, &integer);
44      /*Set body and calculate body motions if start motion is given*/
45      setBody(t, body);
46      /*------FSM predictor (Use FTCS-scheme with staggered grid) ---*/
47      FDMsolver(u, v, p);
48      velBCfield(n, u, v);
49      /*---------------------- Run IBM-------------------------------------*/
50      if (runIBM == 1)
51      {
52        if (fraction == 0.0 && step == (printOut*integer)){printf("RUN IBM\n");}
53        IBM(t, u, v, body, uFlag, vFlag);
54      }
55      /*---------------- Choosen method to solve Poisson ----------------*/
56      iter = 0;
57      if (method == 1 || method == 2)
58      {
59        if (fraction == 0.0 && step == (printOut*integer)){printf("LINEAR SOLVER \n"
    );}
60        linearSolver(t, u, v, p, deltaP);
61      }else if (method == 3 || method == 4 || method == 5)
62      {
63        if (fraction == 0.0 && step == (printOut*integer)){printf("MULTIGRID \n");}
64        multigrid(t, u, v, p, deltaP);
65      }else
66        {
67          printf("Error in choice of method,\n");
68          printf("method = 1, method = 2 OR method = 3.\n");
69          exit (1);
70        }
71      if (fraction == 0.0 && step == (printOut*integer)){printf("t = %f \n\n\n", t)
    ;}
72
```

```
73          /*---------------- PM corrector ------------------------------*/
74          correctVelo(n, u, v, deltaP, uFlag, vFlag);
75          velBCfield(n, u, v);
76          /*---------------- Write to screen/file ----------------------*/
77          printFile(t, u, v, p, body);
78          }
79        }
80    free_dmatrix(p, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
81    free_dmatrix(u, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
82    free_dmatrix(v, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
83    free_dmatrix(deltaP, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
84    free_dmatrix(uFlag, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
85    free_dmatrix(vFlag, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
86    return 0;
87  }
```

Listing B.4: solver.c, FDM-scheme.

```
#include "head.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "input.h"
void FDMsolver(double **u, double **v, double **p)
{
  int i, j;
  double h = 1./n;
  double fux, fuy, fvx, fvy, visu, visv, pdu, pdv;

  for (i = 1; i <= (int)(dimX*n); i++)
    {
      for (j = 1; j <= (int)(dimY*n); j++)
 {
          //Advective terms
    fux = (powf(u[i][j] + u[i+1][j],2) - powf(u[i-1][j] + u[i][j],2)) * (0.25/h);
    fuy = (((v[i][j] + v[i+1][j]) * (u[i][j] + u[i][j+1])) - ((v[i][j-1] + v[i+1][j
      -1]) * (u[i][j-1] + u[i][j]))) * (0.25/h);
    fvx = (((u[i][j] + u[i][j+1]) * (v[i][j] + v[i+1][j])) - ((u[i-1][j] + u[i-1][j
      +1]) * (v[i-1][j] + v[i][j]))) * (0.25/h);
    fvy = (powf(v[i][j] + v[i][j+1],2) - powf(v[i][j-1] + v[i][j],2)) * (0.25/h);

          //viscous terms = nu*laplacian/
     visu = (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4.0*u[i][j]) / (Re*powf(
      h,2));
    visv = (v[i+1][j] + v[i-1][j] + v[i][j+1] + v[i][j-1] - 4.0*v[i][j]) / (Re*powf(h
      ,2));

          //Pressure gradient/
          pdu = (p[i][j] - p[i+1][j])/h;
          pdv = (p[i][j] - p[i][j+1])/h;

          //PM-predrictor (u and v tilde), pdu - fux - fuy + visu = RHS /
    u[i][j] = u[i][j] + dt*(pdu - fux - fuy + visu);
    v[i][j] = v[i][j] + dt*(pdv - fvx - fvy + visv);

```

```
34              if (u[i][j] != u[i][j])
35          {
36            printf("NaN or inf achieved, program aborted... \n");
37            exit(EXIT_FAILURE); //Exit program
38          }
39      }
40        }
41  }
```

Listing B.5: IBM.c, all functions.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "head.h"
#include "input.h"

/*Functions used in this file*/
void setBodyIBM(double xa, double xb, double ya, double yb, double body[4][nPoint],
     double bodyPos[2][nPoint], double bodyVel[2][nPoint]);
void gridShifting(double gridShift[2][n], int xDim, int yDim, int);
void intersectionP(int *nIntersec, int, int xDim, int yDim, int intersecInfo[3][
     maxIntersec], double intersecValue[3][maxIntersec], double bpdyPos[2][nPoint]);
void velocity(int nIntersec, double xa, double ya, double bodyPos[2][nPoint], double
      bodyVel[2][nPoint], int intersecInfo[3][maxIntersec], double intersecValue[3][
     maxIntersec], double **u, double **v,
double **uFlag, double **vFlag, double *Cl, double *Cd);

void IBM(double t, double **u, double **v, double body[4][nPoint], double **uFlag,
     double **vFlag)
{
  int nIntersec = 0;
  double h = 1./n, Cl = 0, Cd = 0;
  /*Domain around body, starting from zero, see report*/
  double xa = (bType == 2) ? 0.0:floor(xmin/h)*h, xb = (bType == 2) ? n*h:ceil(xmax/
     h)*h, ya = (bType == 2) ? 0.0:floor(ymin/h)*h, yb = (bType == 2) ? n*h:ceil(ymax
     /h)*h;
  int xDim, yDim; //Dimensions of the sorunding domain
  /*Define martix*/
  double bodyPos[2][nPoint], bodyVel[2][nPoint];
  double intersecValue[3][maxIntersec];
  int intersecInfo[3][maxIntersec];
  FILE *fileForce;

  xDim = (int)((xb - xa)/h); yDim = (int)((yb - ya)/h);
  setBodyIBM(xa, xb, ya, yb, body, bodyPos, bodyVel);
  //Finds intersectionInfo and Value for each intersection
  intersectionP(&nIntersec, 1, xDim, yDim, intersecInfo, intersecValue, bodyPos);
```

```
31    //Calculation the interpolated velocities
32    velocity(nIntersec, xa, ya, bodyPos, bodyVel, intersecInfo, intersecValue, u , v,
          uFlag, vFlag, &Cl, &Cd);
33    //Print Forces to file
34    if (calcForce == 1)
35    {
36      fileForce = fopen("Force_calculations.txt", "a");
37      fprintf(fileForce, "%f %f %f \n", t, Cl, Cd);
38      fclose(fileForce);
39    }
40  }
41  /*====================================================================*/
42  /*---------------------- Set body positions ---------------------------*/
43  /*====================================================================*/
44  void setBody(double t, double body[4][nPoint])
45  {
46    /*Internal variables*/
47    double h = 1./n, dTheta, theta, integer;
48    int step = (int)(t/dt) + 1;
49    double fraction = modf(step/printOut, &integer);
50
51    /*Set body values*/
52    if (bType == 1 || bType == 2)
53    {
54      for (int point = 0; point < nPoint; point ++)
55      {
56        body[0][point] = pointPos[0][point];  //x-pos
57        body[1][point] = pointPos[1][point];  //y-pos
58        body[2][point] = 0.0;     //u-vel
59        body[3][point] = 0.0;         //v-vel
60      }
61    }else if (bType == 3) //Cylinder
62    {
63      dTheta = 2*M_PI/nPoint;
64      for (int point = 0; point < nPoint; point ++)
65      {
66        theta = point*dTheta;
67        body[0][point] = CG[0] + radius*cos(theta);
```

```
68        body[1][point] = CG[1] + radius*sin(theta);
69        body[2][point] = 0.0;
70        body[3][point] = 0.0;
71      }
72    }else
73    {
74      printf("\n");
75      printf("Fatal error when finding body values\n");
76      printf("Not a valid body type.\n");
77      exit (1);
78    }
79    //Apply body motions
80    if (bMotion == 1 && bType == 3 && bFreq*t <= nMotion*2*M_PI)
81    {
82      if (fraction == 0.0 && step == (printOut*integer)){printf("IN MOTION\n");}
83      for (int point = 0; point < nPoint; point ++)
84      {
85        body[mDir][point] += bAmp*(sin(bFreq*t));
86        body[mDir+2][point] = bAmp*bFreq*cos(bFreq*t);
87      }
88    }
89    return;
90  }
91  /*=====================================================================*/
92  /*---------- Set body positions in surrounding domain ----------------*/
93  /*=====================================================================*/
94  void setBodyIBM(double xa, double xb, double ya, double yb, double body[4][nPoint],
        double bodyPos[2][nPoint], double bodyVel[2][nPoint])
95  {
96    /*Internal variables*/
97    double h = 1./n;
98
99    /*Error if the sorunding domain is less than two cells from the boundary of the
        computional domain*/
100   if ((xa < 2*h || xb > (n*h - 2*h) || ya < 2*h || yb > (n*h - 2*h)) && bType != 2)
101   {
102     printf("\n");
103     printf("Fatal error when using IBM\n");
```

```
104       printf("The body is to close the boundary for the computional domain.\n");
105       printf("Need at least two cells = %f distance\n", 2*h);
106       exit (1);
107     }
108
109     for (int point = 0; point < nPoint; point ++)
110       {
111         bodyPos[0][point] = body[0][point] - xa;
112         bodyPos[1][point] = body[1][point] - ya;
113         bodyVel[0][point] = body[2][point]; //0.0;
114         bodyVel[1][point] = body[3][point]; //0.0;
115       }
116     return;
117 }
118 /*=====================================================================*/
119 /*---------------------- Grid shifting ------------------------------*/
120 /*=====================================================================*/
121 void gridShifting(double gridShift[2][n], int xDim, int yDim, int state)
122 {
123   double h = 1./n;
124   /*set values*/
125   for (int vLine = 0; vLine < xDim; vLine++)
126   {
127     gridShift[0][vLine] = (vLine + 0.5*state)*h;
128   }
129   for (int uLine = 0; uLine < yDim; uLine++)
130   {
131     gridShift[1][uLine] = (uLine + 0.5*state)*h;
132   }
133   return;
134 }
135
136 /*=========================================================================*/
137 /*Find intersections positions (x,y) between body boundary and computional grid */
138 /*=========================================================================*/
139 void intersectionP(int *nIntersec, int state, int xDim, int yDim, int intersecInfo
      [3][maxIntersec], double intersecValue[3][maxIntersec], double bodyPos[2][nPoint
      ])
```

```
140 {
141   /*Goal: find intersection points in x and y position,
142     find the distance from the cloasest outer velocity in intersection direction
143   */
144   int count = 0, pointPlusEn, dummy1 = 0, dummy2 = 0, dummy3 = 0;
145   double h = 1./n, diff = h/10.0, pdy, pdx, ppdx, ppdy;
146   double gridShift[2][n];
147
148   //Set values for the soroundin grid at u- and v- lines
149   gridShifting(gridShift, xDim, yDim, state);
150
151   //Iterating trough points to find intersection points
152   for (int point = 0; point < nPoint; point ++)
153   {
154     pointPlusEn = (point == nPoint-1) ? 0:point+1;
155     pdx = bodyPos[0][pointPlusEn] - bodyPos[0][point];
156     pdy = bodyPos[1][pointPlusEn] - bodyPos[1][point];
157     //Finding intersection points close to velocity-lines in v-direction.
158     for (int vLine = 0; vLine < xDim; vLine ++)
159     {
160       // If pointPlusEn has larger x-position compared to point
161       if(bodyPos[0][pointPlusEn] > bodyPos[0][point]){int dd = point; dummy3 = 1;
      point = pointPlusEn; pointPlusEn = dd;}
162       if ( (gridShift[0][vLine] >= bodyPos[0][pointPlusEn]) && (bodyPos[0][point] >=
      gridShift[0][vLine]))
163       {
164         intersecInfo[0][count] = 2; //intersection wall
165         if ( dummy1 == 1 ){ dummy1 = 0; pointPlusEn = 0; point = nPoint-1;}
166         else if ( dummy3 == 1 ){int dd = point; dummy3 = 0; point = pointPlusEn;
      pointPlusEn = dd;}
167         intersecInfo[1][count] = point;
168         intersecInfo[2][count] = pointPlusEn;
169         intersecValue[0][count] = (-pdx > 0) ? 1:-1;
170         intersecValue[1][count] = gridShift[0][vLine]; //x-pos
171         intersecValue[2][count] = bodyPos[1][point] - ((bodyPos[0][point]-gridShift
      [0][vLine])/pdx)*pdy; //y-pos
172         count += 1;
173       }
```

```
174     //Return dummy 3
175     if( dummy3 == 1 ){int dd = point; dummy3 = 0; point = pointPlusEn; pointPlusEn =
          dd;}
176     }
177     for (int uLine = 0; uLine < yDim; uLine ++)
178     {
179       // If pointPlusEn has larger y-position compared to point
180       if(bodyPos[1][pointPlusEn] > bodyPos[1][point]){int dd = point; dummy2 = 1;
        point = pointPlusEn; pointPlusEn = dd;}
181       if ( (gridShift[1][uLine] >= bodyPos[1][pointPlusEn]) && (bodyPos[1][point] >=
          gridShift[1][uLine]))
182       {
183         intersecInfo[0][count] = 1; //intersection wall
184         if ( dummy1 == 1 ){dummy1 = 0; pointPlusEn = 0; point = nPoint-1;}
185         else if ( dummy2 == 1 ){int dd = point; dummy2 = 0; point = pointPlusEn;
        pointPlusEn = dd;}
186         intersecInfo[1][count] = point;
187         intersecInfo[2][count] = pointPlusEn;
188         intersecValue[0][count] = (pdy > 0) ? 1:-1;
189         intersecValue[1][count] = bodyPos[0][pointPlusEn] - ((bodyPos[1][pointPlusEn
        ]-gridShift[1][uLine])/pdy)*pdx; //x-pos
190         intersecValue[2][count] = gridShift[1][uLine]; //y-pos
191         count += 1;
192       }
193     //Return dummy 2
194     if ( dummy2 == 1 ){int dd = point; dummy2 = 0; point = pointPlusEn; pointPlusEn
        = dd;}
195     }
196   //Return dummy 1
197   if ( dummy1 == 1 ){point = nPoint - 1; dummy1 = 0;}
198
199   /*Error if there is to many intersection points compared to the allocated matrix*/
200   if (count >= maxIntersec)
201   {
202     printf("\n");
203     printf("Fatal error when using IBM\n");
204     printf("There is to many intersection points compared to the allocated matrix.\n
        ");
```

```
205       printf("maxIntersec needs to be larger than %i\n", maxIntersec);
206       exit (1);
207     }
208   }
209    *nIntersec = count;     //Total number of intersections
210    return;
211 }
212 /*================================================================*/
213 /*------------ IBM velocities at each intersection point ------------*/
214 /*================================================================*/
215 //Finding IBM velocities with linear interpolation
216 void velocity(int nIntersec, double xa, double ya, double bodyPos[2][nPoint], double
        bodyVel[2][nPoint], int intersecInfo[3][maxIntersec], double intersecValue[3][
       maxIntersec], double **u, double **v, double **uFlag, double **vFlag, double *Cl
       , double *Cd)
217 {
218    int line, node;
219    double intersecVel, nodeVel, h = 1./n, Fl = 0.0, Fd = 0.0;
220    double **uInterP, **vInterP; //Interpolated velocities
221    double **fx, **fy; //Forcing terms. Force from body on fluid cells due to
       acceleration
222
223    uInterP = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
224    vInterP = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
225    fx= dmatrix(0, (int)(dimX*n)-2, 0, (int)(dimY*n)-2);
226    fy= dmatrix(0, (int)(dimX*n)-2, 0, (int)(dimY*n)-2);
227
228    toZero(n, uInterP);
229    toZero(n, vInterP);
230    toZero(n, uFlag);
231    toZero(n, vFlag);
232    toZero(n-4, fx);
233    toZero(n-4, fy);
234
235    for (int i = 0; i < nIntersec ; i++)
236    {
237      if (intersecInfo[0][i] == 1)
238      {
```

```
239         //Find current velocity at intersection points from the cegment between the
            nodes/
240         intersecVel = linearInterp(intersecValue[2][i], bodyPos[1][intersecInfo[1][i
            ]], bodyPos[1][intersecInfo[2][i]], bodyVel[0][intersecInfo[1][i]], bodyVel[0][
            intersecInfo[2][i]]);
241         //The number on u-velocity line in y-dir (where 1 is the u-velo corresponding
            to node 1)
242         line = (int)((intersecValue[2][i]/h)+0.5 + floor(ymin/h));
243         //The nearest fluid node in x-dir. Starting from 1, node 0 is the node outside
             the surounding domain.
244         node = (int)(rint((intersecValue[1][i]/h) + intersecValue[0][i]*0.51) + floor(
            xmin/h));
245         //Impose the IBM conditions on the fluid/computional grid/
246         nodeVel = linearInterp(node*h , intersecValue[1][i] + xa, (node+intersecValue
            [0][i])*h, intersecVel, u[(int)(node+intersecValue[0][i])][line]);
247         //Forcing terms/
248         fx[node][line] = fx[node][line] - ((nodeVel-u[node][line])/dt)*rho*h*h;
249         //Store interpolated velocity/
250         u[node][line] = nodeVel;
251         //Flag the corrected velocity, PM-solver should not calculate it one more time
252         uFlag[node][line] = 1.0;
253       }else if (intersecInfo[0][i] == 2)
254       {
255         //Find current velocity at intersection points from the cegment between the
            nodes/
256         intersecVel = linearInterp(intersecValue[1][i], bodyPos[0][intersecInfo[1][i
            ]], bodyPos[0][intersecInfo[2][i]], bodyVel[1][intersecInfo[1][i]], bodyVel[1][
            intersecInfo[2][i]]);
257         //The number on v-velocity line in x-dir (where 1 is the v-velo corresponding
            to node 1)
258         line = (int)((intersecValue[1][i]/h) + 0.5 + floor(xmin/h));
259         //The nearest fluid node in y-dir
260         node = (int)(rint((intersecValue[2][i]/h) + intersecValue[0][i]*0.51) + floor(
            ymin/h));
261         //Impose the IBM conditions on the fluid/computional grid/
262         nodeVel = linearInterp(node*h , intersecValue[2][i] + ya, (node+intersecValue
            [0][i])*h,intersecVel, v[line][(int)(node+intersecValue[0][i])]);
263         //Forcing terms/
```

```
264         fy[line][node] = fy[line][node] - ((nodeVel-v[line][node])/dt)*rho*h*h;
265         //Store interpolated velocity/
266         v[line][node] = nodeVel;
267         //Flag the corrected velocity, PM-solver should not calculate it one more time
268         vFlag[line][node] = 1.0;
269       }
270     }
271     //Calculate forces
272     if (calcForce == 1)
273       {
274         for (int i = 0; i <= (int)(dimX*n)-2; i++)
275           {
276             for (int j = 0; j <= (int)(dimY*n)-2; j++)
277               {
278                 Fl += fy[i][j];
279                 Fd += fx[i][j];
280               }
281           }
282         *Cl = 2.0*Fl/(2*radius); //Cl = 2*Fl/(rho*v^2*A)
283         *Cd = 2.0*Fd/(2*radius); //Cd = 2*Fd/(rhod*v^2*A)
284     }
285     free_dmatrix(uInterP, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
286     free_dmatrix(vInterP, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
287     free_dmatrix(fx, 0, (int)(dimX*n)-2, 0, (int)(dimY*n)-2);
288     free_dmatrix(fy, 0, (int)(dimX*n)-2, 0, (int)(dimY*n)-2);
289     return;
290 }
291 /*================================================================*/
292 /*-------------- Linear interpolation 1D ---------------------------*/
293 /*================================================================*/
294 double linearInterp(double nn, double x0, double x1, double y0, double y1)
295 {
296     double linearInterp;
297     /* nn = interp point
298        x0 = boundary point, known function valuelvalue required as left operand of
           assignment
299        x1 = boundary point, known function value
300        y0 = function value at x0
```

```
301         y1 = function value at x1
302     */
303     return linearInterp = y0 +((y1-y0)*fabs(nn-x0))/fabs(x1-x0);
304 }
```

Listing B.6: linearSolver.c, Gauss Seidel method and SOR method is found here.

```
#include <math.h>
#include <stdio.h>
#include "head.h"
#include "input.h"
void linearSolver(double t, double **u, double **v, double **p, double **deltaP)
{
  int print, iter = 0;
  double **RHS, div,  h = 1./n, res = 0.0, error = 0.0, integer;
  int step = (int)(t/dt) + 1;
  double fraction = modf(step/printOut, &integer);
  RHS = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  toZero(n, RHS);

  //Forming the right hand side of the poisson equation rho/dt*div(u_tilde):
  for (int j = 1; j <= (int)(dimY*n); j++)
    {
      for (int i = 1; i <= (int)(dimX*n); i++)
 {
   div = (u[i][j] - u[i-1][j] + v[i][j] - v[i][j-1])/h; //Here, h = dx = dy
         RHS[i][j] = div/dt;
        }
    }
  /*-------- Pressure-correction, Poisson eq. ---------------*/
  if (method == 1)
  {
    GaussSeidel(itmax, n, RHS, deltaP, &iter, &res, &error);
    if (fraction == 0.0 && step == (printOut*integer))
    {
      printf("GAUSS SEIDEL \n");
      printf("Iterations: %i \n", iter);
      printf("Residual: %f. \n\n", res);
    }
  }else if (method == 2)
  {
    SOR(n, RHS, deltaP, &iter, &res, &error);
    if (fraction == 0.0 && step == (printOut*integer))
    {
```

```
38        printf("SOR \n");
39        printf("Iterations: %i \n", iter);
40        printf("Residual: %f. \n\n", res);
41      }
42    }
43    if (fraction == 0.0 && step == (printOut*integer))
44      {
45        printf("\nTotal nr of iterations: %i \n", iter);
46        printf("End residual: %g \n", res);
47      }
48    /*---------------------- Correct pressure ----------------------*/
49    presCorr(n, p, deltaP);
50    free_dmatrix(RHS, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
51    return;
52  }
53  /*--------------------------------------------------------------*/
54  /*---------------------- PM corrector ----------------------*/
55  /*--------------------------------------------------------------*/
56  // Correct velocities without overwriting interpolated velocities from IBM
57  void correctVelo(int nn, double **u, double **v, double **deltaP, double **uFlag,
        double **vFlag)
58  {
59    double h = 1./nn;
60
61    if (runIBM == 1) //IBM
62      {
63        for (int j = 1; j <= (int)(dimY*nn); j++)
64   {
65     for (int i = 1; i <= (int)(dimX*nn); i++)
66       {
67                 if (uFlag[i][j] == 1 && vFlag[i][j] == 1)
68                 {
69                   continue;  //Continue the for-loop
70                 }else if (uFlag[i][j] == 1)
71                 {
72           v[i][j] = v[i][j] - (1/rho)*(dt/h)*(deltaP[i][j+1] - deltaP[i][j]);
73                 }else if (vFlag[i][j] == 1)
74                 {
```

```
75          u[i][j] = u[i][j] - (1/rho)*(dt/h)*(deltaP[i+1][j] - deltaP[i][j]);
76               }else
77               {
78          u[i][j] = u[i][j] - (1/rho)*(dt/h)*(deltaP[i+1][j] - deltaP[i][j]);
79          v[i][j] = v[i][j] - (1/rho)*(dt/h)*(deltaP[i][j+1] - deltaP[i][j]);
80               }
81        CFL(u[i][j], v[i][j]); //CFL-condition
82       }
83  }
84     }else
85     {
86       for (int j = 1; j <= (int)(dimY*nn); j++)
87  {
88    for (int i = 1; i <= (int)(dimX*nn); i++)
89       {
90         u[i][j] = u[i][j] - (1/rho)*(dt/h)*(deltaP[i+1][j] - deltaP[i][j]);
91         v[i][j] = v[i][j] - (1/rho)*(dt/h)*(deltaP[i][j+1] - deltaP[i][j]);
92         CFL(u[i][j], v[i][j]); //CFL-condition
93       }
94  }
95     }
96   velBCfield(nn, u, v);
97   return;
98 }
99 /*--------------------------------------------------------------------*/
100 /*------------------ Pressure-correctin ------------------------------*/
101 /*--------------------------------------------------------------------*/
102 void presCorr(int nn, double **p, double **deltaP)
103 {
104   double avgPress, h = 1./nn;
105
106   for (int j = 1; j <= (int)(dimY*nn); j++)
107     {
108       for (int i = 1; i <= (int)(dimX*nn); i++)
109  {
110    p[i][j] = p[i][j] + deltaP[i][j];
111  }
112     }
```

```
113   if (wall == 5 || wall == 6)
114     {
115       avgPress = avg(nn, p);
116       for (int j = 1; j <= (int)(dimY*nn); j++)
117         {
118           for (int i = 1; i <= (int)(dimX*nn); i++)
119      {
120               p[i][j] = p[i][j] - avgPress;
121             }
122         }
123     }
124   pBCfield(nn, p);
125   return;
126 }
127 /*---------------------------------------------------------------*/
128 /*-------------- Gauss Seidel is used to smooth -----------------*/
129 /*---------------------------------------------------------------*/
130 void GaussSeidel(int imax, int nn, double **f, double **phi, int *iter, double *res,
      double *error)
131 {
132   double h = 1./nn, **phiDiff, **phiPrev, **phiErr;
133   double residual = 0.0, test1, integer;
134   int it;
135   phiDiff = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
136   phiErr = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
137   phiPrev = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
138
139   *iter = 0;
140   for (it = 1; it <= imax; it++)
141     {
142       toZero(nn, phiDiff); toZero(nn, phiPrev); toZero(nn, phiErr);
143       for (int j = 1; j <= (int)(dimY*nn); j++)
144         {
145           for (int i = 1; i <= (int)(dimX*nn); i++)
146             {
147               phiPrev[i][j] = phi[i][j];
148               phi[i][j] = 0.25*(phi[i+1][j] + phi[i-1][j] + phi[i][j+1] + phi[i][j
     -1] - h*h*f[i][j]);
```

```
149                 phiErr[i][j] = f[i][j] - phi[i][j];
150                 phiDiff[i][j] = phi[i][j] - phiPrev[i][j];
151         }
152  }
153          pBCfield(nn, phi);
154         //Find residual between RHS and LHS
155         test1 = maxNorm(nn, phiDiff);
156         if (test1 <= 0.0){residual = 0.0;}
157         else {residual = maxNorm(nn, phiDiff);}
158
159         if (residual <= epsi){break;}
160         if (it >= itmax){break;}
161       }
162    *error = maxNorm(nn, phiErr);
163    *res = residual;
164    *iter = it-1;
165    free_dmatrix(phiErr, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
166    free_dmatrix(phiDiff, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
167    free_dmatrix(phiPrev, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
168    return;
169 }
170 /*------------------------------------------------------------------*/
171 /*---------------------------- SOR ---------------------------------*/
172 /*------------------------------------------------------------------*/
173 void SOR(int nn, double **f, double **phi, int *iter, double *res, double *error)
174 {
175    double h = 1./nn, **phiErr, **phiPrev;
176    double residual = 0.0, omega, test1;
177    int it;
178    phiErr = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
179    phiPrev = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
180    omega = interp(); //Using linear interpolation, omega_opt
181
182
183    *iter = 0;
184    for (it = 1; it <= itmax; it++)
185      {
186        toZero(nn, phiErr); toZero(nn, phiPrev);
```

```
187       for (int j = 1; j <= (int)(dimY*nn); j++)
188         {
189           for (int i = 1; i <= (int)(dimX*nn); i++)
190             {
191               phiPrev[i][j] = phi[i][j];
192               phi[i][j] = (1-omega)*phiPrev[i][j] + omega*0.25*(phi[i+1][j] + phi[i
      -1][j] + phi[i][j+1] + phi[i][j-1] - h*h*f[i][j]);
193               phiErr[i][j] = phi[i][j] - phiPrev[i][j];
194       }
195  }
196       pBCfield(nn, phi);
197       //Find residual between RHS and LHS
198       test1 = maxNorm(nn, phiErr);
199       if (test1 <= 0.0){residual = 0.0;}
200       else {residual = maxNorm(nn, phiErr);}
201       if (residual <= epsi){break;}
202       if (it >= itmax){break;}
203     }
204   *res = residual;
205   *iter = it;
206   free_dmatrix(phiErr, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
207   free_dmatrix(phiPrev, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
208   return;
209 }
210 /*--------------------------------------------------------------*/
211 /*---------------- linear interpolation for omega_opt ----------*/
212 /*--------------------------------------------------------------*/
213 double interp(void)
214 {
215   double nn[9] = {0, 5, 10, 20, 30, 40, 60, 100, 500};
216   double oo[9] = {1.7, 1.78, 1.86, 1.92, 1.95, 1.96, 1.97, 1.98, 1.99};
217   int i, size;
218   double interp, x0, x1, y0, y1;
219   size = 9;
220   for (i = 1; i < size; i++)
221     {
222       if (nn[i] > n)
223  {
```

```
224     x0 = nn[i-1];
225     x1 = nn[i];
226     y0 = oo[i-1];
227     y1 = oo[i];
228     break;
229   }
230       }
231    return interp = y0 +((y1-y0)*(n-x0))/(x1-x0);
232 }
```

Listing B.7: multigrid.c, all multigird functions are found here.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "head.h"
#include "input.h"
void twoStepM(int nn, double t, double **RHS, double **deltaP, int *iter, double
     error[ncycle], double endRes[ncycle]);
void Vcycle(int nn, double t, double **f, double **MVAL, int *iter, double error[
     ncycle], double endRes[ncycle]);
void fullMultigrid(int nn, double t, double **f, double **value, int *it, double
     error[ncycle], double endRes[ncycle]);

void multigrid(double t, double **u, double **v, double **p, double **deltaP)
{
  int nF = n, k, iter = 0, step = (int)(t/dt) + 1;
  double h = 1./n, integer, div, residual[ncycle], error[ncycle];
  double fraction = modf(step/printOut, &integer), **RHS;

  RHS = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  toZero(n, RHS);
  //--------------- Check if n is a power of 2 -----------------/
  k = log_2(n);
  if (n != powerOf(k))
  {
    printf("\n");
    printf("Fatal error when using MULTIGRID\n");
    printf("N is not a power of 2.\n");
    exit (1);
  }
  //---------------- Set the right hand side ---------------------/
  for (int j = 1; j <= (int)(dimY*nF); j++)
    {
      for (int i = 1; i <= (int)(dimX*nF); i++)
 {
   div = (u[i][j] - u[i-1][j] + v[i][j] - v[i][j-1])/h; //Here, h = dx = dy
         RHS[i][j] = (rho/dt)*div;
       }
```

```
35      }
36    //------------------ Run the choosen method ------------------------/
37    if (method == 3)  //Two-step multigrid
38      {
39        if (fraction == 0.0 && step == (printOut*integer)){printf("TWO-STEP MULTIGRID\
        n");}
40        twoStepM(n, t, RHS, deltaP, &iter, residual, error);
41    }else if (method == 4)  //V-cycle multigrid
42      {
43        if (fraction == 0.0 && step == (printOut*integer)){printf("V-CYCLE MULTIGRID \
        n\n");}
44        Vcycle(n, t, RHS, deltaP, &iter, residual, error);
45    }else if (method == 5)  //Full multigrid
46      {
47        if (fraction == 0.0 && step == (printOut*integer)){printf("FULL MULTIGRID \n")
        ;}
48        fullMultigrid(n, t, RHS, deltaP, &iter, residual, error);
49    }
50    if (fraction == 0.0 && step == (printOut*integer))
51      {
52        printf("\nTotal nr of iterations: %i \n", iter);
53      }
54    //------------------- Correct pressure ---------------------------/
55    presCorr(nF, p, deltaP);
56    pBCfield(n, p);
57
58    free_dmatrix(RHS, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
59  }
60  /*============================================================*/
61  /* two-step multigrid(fine -> coarce -> fine)      */
62  /*============================================================*/
63  void twoStepM(int nn, double t, double **RHS, double **value, int *it, double error[
        ncycle], double endRes[ncycle])
64  {
65    int nC = nn/2, nF = nn, k, nIter = 0, iter = 0, numb = (int)(t/dt), print = 0;
66    double h = 1./nn, res = 0.0, err = 0.0, integer;
67    int step = (int)(t/dt) + 1;
68    double fraction = modf(step/printOut, &integer);
```

```
69    double **R, **E, div;
70
71    R = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
72    E = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
73
74    //-------Relax v times on the finest grid-----/
75    GaussSeidel(itPre, nF, RHS, value, &iter, &res, &err);
76    if (fraction == 0.0 && step == (printOut*integer))
77    {
78      printf("Pre-smoothing - iterations: %i \n", iter);
79      printf("Residual: %g. \n\n", res);
80    }
81    //Loop trough a fixed number of V-cycles.
82    for (int c = 0; c < ncycle; c++)
83      {
84        if (fraction == 0.0 && step == (printOut*integer)){printf("\nCYCLE NUMBER %i \
      n", c+1);}
85        //-- Restrict the residual from fine grid to coarse grid --/
86        toZero(n, R);
87  //      weightedResidual(nC, RHS, value, R);
88        injectedResidual(nC, RHS, value, R);
89        //Use zero as initial guess for E on the coarse grid approximation of the
      error E.
90        toZero(nF, E);
91        GaussSeidel(itCoarse, nC, R, E, &iter, &res, &err);
92        if (fraction == 0.0 && step == (printOut*integer))
93        {
94          printf("Smooth coarse grid - iterations: %i \n", iter);
95          printf("Residual: %g. \n\n", res);
96        }
97        //Interpolate from coarse grid to fine grid
98        interpolation(nC, E);
99        pBCfield(nF, value);
100       addValue(nF, E, value);
101       //Post-smoothing, removing error
102       GaussSeidel(itPost, nF, RHS, value, &iter, &res, &err);
103       if (fraction == 0.0 && step == (printOut*integer))
104       {
```

```
105         printf("Post-smoothing - iterations: %i \n", iter);
106         printf("Residual: %g. \n\n", res);
107       }
108       nIter +=iter;
109       endRes[c] = res;
110       error[c] = err;
111       if (res <= epsi){break;}
112     }
113   *it = nIter;
114   free_dmatrix(R, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
115   free_dmatrix(E, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
116   return;
117 }
118 /*============================================================*/
119 /*    V-cycle multigrid       */
120 /*============================================================*/
121 void Vcycle(int nn, double t, double **f, double **MVAL, int *it, double error[
      ncycle], double endRes[ncycle])
122 {//MVAL = deltaP
123   int nC = nn/2, nF = nn, k, nIter = 0, iter = 0, dCurr, count = 0, numb, print = 0;
124   double h = 1./nn, integer, res = 0.0, err = 0.0;
125   int step = (int)(t/dt) + 1, limit;
126   double fraction = modf(step/printOut, &integer);
127   //-----------------Allocate matrix-----------------------/
128   double **value, **newValue, **R, **E, div;
129   double ***RHS, ***VAL, **MRHS;
130
131   MRHS = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
132   value = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
133   newValue = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
134   R = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
135   E = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
136   RHS = f3tensorD(0, d, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
137   VAL = f3tensorD(0, d, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
138
139   toZero(n, value);
140   toZero(n, R);
141   toZero(n, MRHS);
```

```
142   for (int dd = 0; dd < d; dd++)
143     {
144       toZeroM(dd, RHS);
145       toZeroM(dd, VAL);
146     }
147
148   matrixArray(nF, f, RHS, 0);
149   matrixArray(nF, MVAL, VAL, 0);
150
151   //-------Relax v times on the finest grid-----/
152   GaussSeidel(itPre, nF, f, MVAL, &iter, &res, &err);
153   if (fraction == 0.0 && step == (printOut*integer))
154   {
155     printf("Pre-smoothing 1 - iterations: %i \n", iter);
156     printf("Residual: %g. \n\n", res);
157   }
158   //Loop trough a fixed number of V-cycles.
159   for (int c = 0; c < ncycle; c++)
160     {
161       if (fraction == 0.0 && step == (printOut*integer)){printf("\nCYCLE NUMBER %i \
      n", c+1);}
162       nC = nn/2, nF = nn;
163       //-- Restrict the residual from fine grid to coarse grid --/
164       arrayMatrix(nF, RHS, MRHS, 0);
165       //  weightedResidual(nC, MRHS, MVAL, R);
166       injectedResidual(nC, MRHS, MVAL, R);
167       //Set R equals to RHS for the current depth and store it in RHS
168       matrixArray(nC, R, RHS, 1);
169
170       k = log_2(nn);
171       limit = (k > d) ? d:k;
172       dCurr = 2;
173       //---------------- Go coarser ----------------------------/
174       while (dCurr < limit)
175 {
176   // Set number of grid points in coarse and fine grid
177   nF = nC; nC = nF/2;
178   //Use zero as initial guess for E on the coarse grid approximation of the error E
```

```
    .
179   toZero(n, E);
180   toZero(n, MRHS);
181   arrayMatrix(nF, RHS, MRHS, dCurr-1);
182
183   GaussSeidel(itPre, nF, MRHS, E, &iter, &res, &err);
184   if (fraction == 0.0 && step == (printOut*integer))
185         {
186           printf("Pre-smoothing %i - iterations: %i \n", dCurr, iter);
187           printf("Residual: %g. \n\n", res);
188         }
189
190   //Store correrction
191   matrixArray(nF, E, VAL, dCurr-1);
192   //Restrict the residual
193   //weightedResidual(nC, MRHS, E, R);
194   injectedResidual(nC, MRHS, E, R);
195   //Set R equals to RHS for the current depth and store it in RHS
196   matrixArray(nC, R, RHS, dCurr);
197   dCurr += 1;
198 }
199       //---------------- Solve coarsest grid ---------------------//
200       dCurr -= 1;
201       if (nC < 2){printf("\n\nFatal error in MULTIGRID. \nCoarsest grid has n less
      than 2!\n"); exit(1);}
202
203       //OR solve directly with A^Lh
204       toZero(n, E);
205       toZero(n, MRHS);
206       arrayMatrix(nC, RHS, MRHS, dCurr);
207
208       if (nC == 2 )
209       {
210         solveCoarse(E, MRHS);
211         if (fraction == 0.0 && step == (printOut*integer)){printf("SOLVED EXACTLY
      FOR n = 2. \n");}
212       }else
213 {
```

```
214     GaussSeidel(itCoarse, nC, MRHS, E, &iter, &res, &err);
215     if (fraction == 0.0 && step == (printOut*integer))
216       {
217         printf("Coarsest grid - iterations: %i \n", iter);
218         printf("Residual: %g. \n\n", res);
219       }
220   }
221        dCurr -= 1;
222        //---------------- Go finer -------------------------------//
223        while (0 <= dCurr)
224   {
225     //Interpolate from coarse grid to fine grid
226     interpolation(nC, E); //Get interpolated E
227
228     toZero(n, MVAL);
229     arrayMatrix(nF, VAL, MVAL, dCurr);
230
231     // Set pressure BC on MVAL
232     addValue(nF, E, MVAL); //Get value = value + E
233     pBCfield(nF, MVAL);
234
235     toZero(n, MRHS);
236     arrayMatrix(nF, RHS, MRHS, dCurr);
237
238     //Post-smoothing, removing error
239     GaussSeidel(itPost, nF, MRHS, MVAL, &iter, &res, &err);
240     if (fraction == 0.0 && step == (printOut*integer))
241           {
242             printf("Post-smoothing %i - iterations: %i \n", dCurr+1, iter);
243             printf("Residual: %g. \n\n", res);
244           }
245     nIter +=iter;
246
247           if (dCurr == 0){matrixArray(nF, MVAL, VAL, 0);}
248     // Set number of grid points in coarse and fine grid
249     nC = nF; nF = nC*2;
250     dCurr -= 1;
251   }
```

```
252         endRes[c] = res;
253         error[c] = err;
254         if (res <= epsi){break;}
255       }
256     *it = nIter;
257
258     free_dmatrix(MRHS, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
259     free_dmatrix(value, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
260     free_dmatrix(newValue, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
261     free_dmatrix(R, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
262     free_dmatrix(E, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
263     free_f3tensorD(RHS, 0, d, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
264     free_f3tensorD(VAL, 0, d, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
265     return;
266 }
267 /*============================================================*/
268 /*    Full multigrid        */
269 /*============================================================*/
270 void fullMultigrid(int nn, double t, double **f, double **value, int *it, double
        error[ncycle], double endRes[ncycle])
271 { //value = deltaP
272     int nC = nn/2, nF = nn, nnC, nnF, k, nIter = 0, iter = 0, dCurr, count = 0, numb,
        print = 0;
273     double h = 1./nn, res = 0.0, err = 0.0, integer;
274     int step = (int)(t/dt) + 1, limit;
275     double fraction = modf(step/printOut, &integer);
276
277     k = log_2(nn);
278     limit = (k > d) ? d:k;
279     dCurr = limit-1;
280     //-----------------Allocate matrix-----------------------/
281     double **R, **MRHS, **MVAL, div;
282     double ***reRHS, ***RHS, ***VAL;
283
284     MRHS = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
285     MVAL = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
286     R = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
287     reRHS = f3tensorD(0, d, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
```

```
288   RHS = f3tensorD(0, d, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
289   VAL = f3tensorD(0, d, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
290
291   toZero(n, R);
292   toZero(n, MRHS);
293   toZero(n, MVAL);
294   for (int dd = 0; dd < d; dd++)
295     {
296       toZeroM(dd, reRHS);
297       toZeroM(dd, RHS);
298       toZeroM(dd, VAL);
299
300     }
301   //Forming the right hand side of the poisson equation
302   matrixArray(nF, f, reRHS, dCurr);
303
304   //-- Fill temporary coarser RHS with restrictions from the calculated RHS --//
305   //------------------ Go coarser until next coarsest grid ---------------------//
306   while (dCurr > 0)
307   {
308     injected(nC, f);
309     //weighted(nC, MRHS, value);
310     matrixArray(nC, f, reRHS, dCurr-1);
311     dCurr -= 1;
312     nF = nC; nC = nC/2;
313   }
314   //------------------ Set initial solution on coarsest grid -------------------//
315   if (nF < 2){printf("\n\n Fatal error in MULTIGRID. \n Coarsest grid has n less
        than 2! \n"); exit(1);}
316
317   injected(nC, f);
318   toZero(n, MVAL);
319   //Solve on coarsest grid
320   if (nF == 2){printf("SOLVED EXACTLY FOR n = 2. \n"); solveCoarse(MVAL, f);}
321   else
322     {
323       GaussSeidel(itCoarse, nC, f, MVAL, &iter, &res, &err);
324       if (fraction == 0.0 && step == (printOut*integer))
```

```
325         {
326           printf("Coarsest grid (START) - iterations: %i \n", iter);
327           printf("Residual: %g. \n\n", res);
328         }
329     }
330
331   matrixArray(nF, MVAL, VAL, dCurr);
332   dCurr += 1;
333   nC = nF; nF = nF*2;
334   //---------------------- Nested iteration loop --------------------------//
335   while (dCurr < limit)
336   {
337     toZero(n, MVAL);
338     arrayMatrix(nC, VAL, MVAL, dCurr-1);
339
340     interpolation(nC, MVAL); //Get interpolated MVAL back
341
342     matrixArray(nF, MVAL, VAL, dCurr);
343     copy(reRHS, RHS, dCurr);
344
345     for (int c = 1; c <= ncycle; c++) //Loop trough a fixed number of V-cycles.
346       {
347         if (fraction == 0.0 && step == (printOut*integer)){printf("CYCLE NUMBER %i \
      n", c);}
348         nnF = nF; nnC = nC;
349         for (int dCalc = dCurr; dCalc >= 1; dCalc--) //Loop downward in V.
350           {
351             toZero(n, MVAL);
352             arrayMatrix(nnF, VAL, MVAL, dCalc);
353             toZero(n, MRHS);
354             arrayMatrix(nnF, RHS, MRHS, dCalc);
355
356             GaussSeidel(itPre, nnF, MRHS, MVAL, &iter, &res, &err);
357             if (fraction == 0.0 && step == (printOut*integer))
358             {
359               printf("Pre-smoothing %i - iterations: %i \n", dCalc, iter);
360               printf("Residual: %g. \n\n", res);
361             }
```

```
362
363             toZero(n, R);
364             //weightedResidual(nnC, MRHS, MVAL, R);
365             injectedResidual(nnC, MRHS, MVAL, R); //Restricted residual, use as the
    next RHS
366
367             matrixArray(nnC, R, RHS, dCalc-1);
368             toZero(n, MVAL);
369             matrixArray(nnC, MVAL, VAL, dCalc-1); //Zero as initial guess for next
    smoothning
370             nnF = nnC; nnC = nnC/2;
371           }
372         //Solve on coarsest grid
373         if (nnF == 2){printf("SOLVED EXACTLY FOR n = 2. \n"); solveCoarse(MRHS, MVAL
    );}
374         else
375           {
376             GaussSeidel(itCoarse, nnF, R, MVAL, &iter, &res, &err);
377             if (fraction == 0.0 && step == (printOut*integer))
378               {
379                 printf("Coarsest grid - iterations: %i \n", iter);
380                 printf("Residual: %g. \n\n", res);
381               }
382           }
383         matrixArray(nnF, MVAL, VAL, 0);
384         nnC = nnF; nnF = nnF*2;
385
386         for (int dCalc = 1; dCalc <= dCurr; dCalc++) //Loop upward in V.
387           {
388             toZero(n, MVAL);
389             toZero(n, value);
390             arrayMatrix(nnF, VAL, MVAL, dCalc-1);
391             arrayMatrix(nnF, VAL, value, dCalc);
392
393             //Interpolate from coarse grid to fine grid
394             interpolation(nnC, MVAL);
395             addValue(nnF, MVAL, value); //Get value = value + E
396
```

```
397             // Set pressure BC on value
398             pBCfield(nnF, value);
399
400             toZero(n, MRHS);
401             arrayMatrix(nnF, RHS, MRHS, dCalc);
402
403             //Post-smoothing, removing error
404             GaussSeidel(itPost, nnF, MRHS, value, &iter, &res, &err);
405             if (fraction == 0.0 && step == (printOut*integer))
406             {
407               printf("Post-smoothing %i - iterations: %i \n", dCalc, iter);
408               printf("Residual: %g. \n\n", res);
409             }
410             nIter +=iter;
411             matrixArray(nnF, value, VAL, dCalc);
412             nnC = nnF; nnF = nnF*2;
413           }
414         endRes[c-1] = res;
415       }
416     dCurr += 1;
417     nC = nF; nF = nF*2;
418   }
419   *it = nIter;
420   free_dmatrix(MRHS, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
421   free_dmatrix(MVAL, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
422   free_dmatrix(R, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
423   free_f3tensorD(reRHS, 0, d, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
424   free_f3tensorD(RHS, 0, d, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
425   free_f3tensorD(VAL, 0, d, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
426   return;
427 }
428 /*-------------------------------------------------------------*/
429 /*---------------- Residual -----------------------------------*/
430 /*-------------------------------------------------------------*/
431 // r = f - A*phi, phi:approximation of value.
432 double residual(int nn, double **f, double **phi, int i, int j)
433 {
434   double R, h = 1./nn;
```

```
435   R = f[i][j] - (1/pow(h,2))*(phi[i+1][j] + phi[i-1][j] + phi[i][j+1] + phi[i][j-1]
          - 4*phi[i][j]);
436   return R;
437 }
438 /*--------------------------------------------------------------*/
439 /*----------------- Restriction of residual  -------------------*/
440 /*--------------------------------------------------------------*/
441 void weightedResidual(int nC, double **RHS, double **phi, double **R)
442 {
443   int ii, jj, nn = nC*2;
444   double Rcurr, R1, R2, R3, R4, R5, R6, R7, R8;
445   toZero(n, R);
446
447   for (int j = 2; j < (int)(dimY*nC); j++)
448     {
449       for (int i = 2; i < (int)(dimX*nC); i++)
450  {
451           jj = j*2+1; ii = i*2+1;
452           Rcurr = residual(nn, RHS, phi, ii, jj);
453           R1 = residual(nn, RHS, phi, ii-1, jj-1);
454           R2 = residual(nn, RHS, phi, ii, jj-1);
455           R3 = residual(nn, RHS, phi, ii+1, jj-1);
456           R4 = residual(nn, RHS, phi, ii-1, jj);
457           R5 = residual(nn, RHS, phi, ii+1, jj);
458           R6 = residual(nn, RHS, phi, ii-1, jj+1);
459           R7 = residual(nn, RHS, phi, ii, jj+1);
460           R8 = residual(nn, RHS, phi, ii+1, jj+1);
461
462           R[i][j] = 0.25*(Rcurr + 0.5*(R2 + R4 + R5 + R7) + 0.25*(R1 + R3 + R6 + R8)
          );
463         }
464     }
465   //Set ghost cells
466   for (int j = 1; j <= (int)(dimY*nC); j++)
467     {
468       jj = j*2-1;
469       Rcurr = residual(nn, RHS, phi, 1, jj);
470       R[1][j] = Rcurr;
```

```
471       Rcurr = residual(nn, RHS, phi, (int)(dimX*nn), jj);
472       R[(int)(dimX*nC)][j] = Rcurr;
473     }
474   for (int i = 1; i <= (int)(dimX*nC); i++)
475     {
476       ii = i*2-1;
477       Rcurr = residual(nn, RHS, phi, ii, 1);
478       R[i][1] = Rcurr;
479       Rcurr = residual(nn, RHS, phi, ii, (int)(dimY*nn));
480       R[i][(int)(dimY*nC)] = Rcurr;
481     }
482 }
483 /*--------------------------------------------------------------*/
484 void injectedResidual(int nC, double **RHS, double **phi, double **R)
485 {
486   int ii, jj, nF = nC*2;
487   double h = 1./nF;
488   toZero(n, R);
489
490   //Loop over cells and find restricted residual for the coarse grid-cells
491   for (int j = 1; j <= (int)(dimY*nC); j++)
492     {
493       for (int i = 1; i <= (int)(dimX*nC); i++)
494  {
495           jj = j*2; ii = i*2;
496           R[i][j] = residual(nF, RHS, phi, ii, jj);
497         }
498     }
499 }
500 /*--------------------------------------------------------------*/
501 /*---------------------- Restriction ---------------------------*/
502 /*--------------------------------------------------------------*/
503 void weighted(int nC, double **R)
504 {
505   int iC, iF, jC, jF, nF = nC*2;
506   double **Rout;
507   Rout = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
508   toZero(n, Rout);
```

```
509   //Loop over cells and find restricted values for the coarse grid-cells
510   for (jC = 1, jF = 2 ; jC <= (int)(dimY*nC); jC++, jF+=2)
511     {
512       for (iC = 1, iF = 2 ; iC <= (int)(dimX*nC); iC++, iF+=2)
513  {
514           Rout[iC][jC] = 0.25*(R[iF][jF] + 0.5*(R[iF][jF-1] + R[iF-1][jF] + R[iF+1][
     jF]
515    + R[iF][jF+1]) + 0.25*(R[iF-1][jF-1] + R[iF+1][jF-1] + R[iF-1][jF+1] + R[iF+1][jF
      +1]));
516         }
517     }
518   ccopy(Rout, R);
519   free_dmatrix(Rout, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
520 }
521 /*--------------------------------------------------------------*/
522 void injected(int nC, double **R)
523 {
524   int iC,iF, jC, jF, nF = nC*2;
525   double **Rout;
526   Rout = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
527   toZero(n, Rout);
528   //Loop over cells and find restricted values for the coarse grid-cells
529   for (jC = 1, jF = 2 ; jC <= (int)(dimY*nC); jC++, jF+=2)
530     {
531       for (iC = 1, iF = 2 ; iC <= (int)(dimX*nC); iC++, iF+=2)
532  {
533           Rout[iC][jC] = R[iF][jF];
534         }
535     }
536   ccopy(Rout, R);
537   free_dmatrix(Rout, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
538 }
539 /*--------------------------------------------------------------*/
540 /*------------------- Add Value --------------------------------*/
541 /*--------------------------------------------------------------*/
542 void addValue(int nn, double **E, double **phi)
543 {
544   //Add interpolated value to fine grid cell value
```

```
545   for (int j = 1; j <= (int)(dimY*nn); j++)
546     {
547       for (int i = 1; i <= (int)(dimX*nn); i++)
548  {
549    phi[i][j] += E[i][j];
550
551         }
552     }
553   return;
554 }
555 /*--------------------------------------------------------------*/
556 /*---------------------- Interpolasjon -------------------------*/
557 /*--------------------------------------------------------------*/
558 void interpolation(int nC, double **E)
559 {
560   double **returnE;
561   int iC, jC, iF, jF, nF = nC*2;
562
563   returnE = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
564   toZero(n, returnE);
565
566  //Bilinear interpolation for internal nodes
567   for (jC = 0, jF = 0; jC <= (int)(dimY*nC); jC++, jF+=2)
568     {
569       for (iC = 0, iF = 0; iC <= (int)(dimX*nC); iC++, iF+=2)
570  {
571           returnE[iF][jF] = 0.5625*E[iC][jC] + 0.1875*(E[iC+1][jC] + E[iC][jC+1]) +
     0.0625*E[iC+1][jC+1];
572           returnE[iF+1][jF] = 0.5625*E[iC+1][jC] + 0.1875*(E[iC+1][jC+1] + E[iC][jC
     ]) + 0.0625*E[iC][jC+1];
573           returnE[iF][jF+1] = 0.5625*E[iC][jC+1] + 0.1875*(E[iC+1][jC+1] + E[iC][jC
     ]) + 0.0625*E[iC+1][jC];
574           returnE[iF+1][jF+1] = 0.5625*E[iC+1][jC+1] + 0.1875*(E[iC+1][jC] + E[iC][
     jC+1]) + 0.0625*E[iC][jC];
575         }
576     }
577   // Set mixed prolongation at boundary nodes
578   for (jC = 0, jF = 0; jC <= (int)(dimY*nC); jF+=2, jC++)
```

```
579     {
580       returnE[0][jF] = 0.75*E[1][jC] + 0.25*E[1][jC+1];
581       returnE[0][jF+1] = 0.75*E[1][jC+1] + 0.25*E[1][jC];
582       returnE[1][jF] = 0.75*E[1][jC] + 0.25*E[1][jC+1];
583       returnE[1][jF+1] = 0.75*E[1][jC+1] + 0.25*E[1][jC];
584
585       returnE[(int)(dimX*nF)][jF+1] = 0.75*E[(int)(dimX*nC)][jC+1] + 0.25*E[(int)(
      dimX*nC)][jC];
586       returnE[(int)(dimX*nF)][jF] =  0.75*E[(int)(dimX*nC)][jC] + 0.25*E[(int)(dimX*
      nC)][jC+1];
587       returnE[(int)(dimX*nF)+1][jF+1] = 0.75*E[(int)(dimX*nC)][jC+1] + 0.25*E[(int)(
      dimX*nC)][jC];
588       returnE[(int)(dimX*nF)+1][jF] =  0.75*E[(int)(dimX*nC)][jC] + 0.25*E[(int)(
      dimX*nC)][jC+1];
589     }
590   for (jC = 0, jF = 0; jC <= (int)(dimX*nC); jF+=2, jC++)
591     {
592       returnE[jF][1] = 0.75*E[jC][1] + 0.25*E[jC+1][1];
593       returnE[jF+1][1] = 0.75*E[jC+1][1] + 0.25*E[jC][1];
594       returnE[jF][0] = 0.75*E[jC][1] + 0.25*E[jC+1][1];
595       returnE[jF+1][0] = 0.75*E[jC+1][1] + 0.25*E[jC][1];
596
597       returnE[jF+1][(int)(dimY*nF)] = 0.75*E[jC+1][(int)(dimY*nC)] + 0.25*E[jC][(int
      )(dimY*nC)];
598       returnE[jF][(int)(dimY*nF)] =  0.75*E[jC][(int)(dimY*nC)] + 0.25*E[jC+1][(int)
      (dimY*nC)];
599       returnE[jF+1][(int)(dimY*nF)+1] = 0.75*E[jC+1][(int)(dimY*nC)] + 0.25*E[jC][(
      int)(dimY*nC)];
600       returnE[jF][(int)(dimY*nF)+1] =  0.75*E[jC][(int)(dimY*nC)] + 0.25*E[jC+1][(
      int)(dimY*nC)];
601     }
602   //Set return matrix
603   ccopy(returnE, E);
604   free_dmatrix(returnE, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
605   return;
606 }
607 /*-------------------------------------------------------------*/
608 /*--------------- Solve coarsest grid----------------------*/
```

```
609 /*-------------------------------------------------------------*/
610 void solveCoarse(double **phi, double **f)
611 {
612   double h = 0.5, **returnPhi;
613   returnPhi = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
614   toZero(n, returnPhi);
615 
616   returnPhi[1][1] = (phi[0][1] + phi[2][1] + phi[1][0] + phi[1][2] - h*h*f[1][1])
        /4.0;
617 
618   ccopy(returnPhi, phi);
619   free_dmatrix(returnPhi, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
620 }
621 /*-------------------------------------------------------------*/
622 /*---------------- Transform array to matrix ------------------*/
623 /*-------------------------------------------------------------*/
624 void arrayMatrix(int nn, double ***array, double **matrix, int dd)
625 {
626   for (int j = 0; j <= (int)(dimY*nn)+1; j++)
627     {
628       for (int i = 0; i <= (int)(dimX*nn)+1; i++)
629  {
630           matrix[i][j] = array[dd][i][j];
631         }
632     }
633 }
634 /*-------------------------------------------------------------*/
635 /*---------------- Transform matrix to array ------------------*/
636 /*-------------------------------------------------------------*/
637 void matrixArray(int nn, double **matrix, double ***array, int dd)
638 {
639   for (int j = 0; j <= (int)(dimY*nn)+1; j++)
640     {
641       for (int i = 0; i <= (int)(dimX*nn)+1; i++)
642  {
643           array[dd][i][j] = matrix[i][j];
644         }
645     }
```

```
646 }
647 /*--------------------------------------------------------------*/
648 /*---------------------- Copy values ---------------------------*/
649 /*--------------------------------------------------------------*/
650 void copy(double ***from, double ***to, int dd)
651 {
652   for (int j = 0; j < (int)(dimY*n)+2; j++)
653     {
654       for (int i = 0; i < (int)(dimX*n)+2; i++)
655         {
656           to[dd][i][j] = from[dd][i][j];
657         }
658     }
659 }
660 /*--------------------------------------------------------------*/
661 void ccopy(double **from, double **to)
662 {
663   for (int j = 0; j <= (int)(dimY*n)+1; j++)
664     {
665       for (int i = 0; i <= (int)(dimX*n)+1; i++)
666  {
667           to[i][j] = from[i][j];
668         }
669     }
670 }
671 /*--------------------------------------------------------------*/
672 /*----------------- set matrix to zero -----------------------*/
673 /*--------------------------------------------------------------*/
674 void toZero(int dim, double **matrix)
675 {
676   for (int j = 0; j <= (int)(dimY*dim)+1; j++)
677     {
678       for (int i = 0; i <= (int)(dimX*dim)+1; i++)
679  {
680           matrix[i][j] = 0.0;
681         }
682     }
683 }
```

```
684 /*-------------------------------------------------------------*/
685 /*---------- set matrix with diff dim to zero ----------*/
686 /*-------------------------------------------------------------*/
687 void toZeroM(int dd, double ***matrix)
688 {
689   for (int i = 0; i < (int)(dimX*n)+2; i++)
690     {
691       for (int j = 0; j < (int)(dimY*n)+2; j++)
692         {
693           matrix[dd][i][j] = 0.0;
694         }
695      }
696 }
697 /*-------------------------------------------------------------*/
698 /*---------- calculating 2 logarithm of n ----------*/
699 /*-------------------------------------------------------------*/
700 int log_2(int nn)
701 {
702   int absN = abs(nn), baseNr = 2, realNr;
703
704   if ( nn == 0 )
705   {
706     realNr = 0;
707   }
708   else
709   {
710     realNr = 0;
711     while ( baseNr <= absN )
712     {
713       realNr = realNr + 1;
714       baseNr = baseNr * 2;
715     }
716   }
717   return realNr;
718 }
719 /*-------------------------------------------------------------*/
720 /*---------- calculating 2 power of value k ----------*/
721 /*-------------------------------------------------------------*/
```

```
722 int powerOf(int k)
723 {
724   int value;
725
726   if ( k < 0 )
727   {
728     printf("Error: b^c = a, c i negative. n needs to be positive.");
729     exit(1);
730   }
731   else if ( k == 0 ){value = 1;}
732   else if ( k == 1 ){value = 2;}
733   else
734   {
735     value = 1;
736     for (int i = 1; i <= k; i++ )
737     {
738       value = value * 2;
739     }
740   }
741   return value;
742 }
```

Listing B.8: BC.c,velocity and pressure BC's.

```
#include "head.h"
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "input.h"
/*--------------------------------------------------------------------------*/
/*                    BC FOR VELOCITY                                       */
/*--------------------------------------------------------------------------*/
void velBCfield(int nn, double **u, double **v)
{
  double h = 1./nn;

  if (wall == 1)      //BC's for lid driven cavety flow
    {
      for (int j = 0; j <= (int)(dimY*nn)+1; j++)
 {
   v[0][j] = -v[1][j];
   v[(int)(dimX*nn)+1][j] = -v[(int)(dimX*nn)][j];
   u[0][j] = 0.0;
   u[(int)(dimX*nn)][j] = 0.0;
 }
      for (int i = 0; i <= (int)(dimX*nn)+1; i++)
 {
   u[i][(int)(dimY*nn)+1] = -u[i][(int)(dimY*nn)] + 2.0*velX;
   u[i][0] = -u[i][1];
   v[i][(int)(dimY*nn)] = 0.0;
   v[i][0] = 0.0;
 }
    }else if (wall == 2)     //Couette flow
    {
      for (int j = 0; j <= (int)(dimY*nn)+1; j++)
 {
   u[0][j] = u[1][j];
   v[0][j] = v[1][j];
   u[(int)(dimX*nn)+1][j] = u[(int)(dimX*nn)][j];
   v[(int)(dimX*nn)+1][j] = v[(int)(dimX*nn)][j];
 }
```

```
38        for (int i = 0; i <= (int)(dimX*nn)+1; i++)
39  {
40    v[i][(int)(dimY*nn)] = 0.0;
41    v[i][0] = 0.0;
42    u[i][(int)(dimY*nn)+1] = -u[i][(int)(dimY*nn)] + 2.0*velX;
43    u[i][0] = -u[i][1];
44  }
45     }else if (wall == 3) //Channel flow with inflow velocity
46     {
47          for (int j = 0; j <= (int)(dimY*nn)+1; j++)
48    {
49      u[0][j] = velX;
50      v[0][j] = v[1][j];
51      u[(int)(dimX*nn)+1][j] = u[(int)(dimX*nn)][j];
52      v[(int)(dimX*nn)+1][j] = v[(int)(dimX*nn)][j];
53    }
54          for (int i = 0; i <= (int)(dimX*nn)+1; i++)
55    {
56      v[i][0] = 0.0;
57      v[i][(int)(dimY*nn)] = 0.0;
58      u[i][(int)(dimY*nn)+1] = u[i][(int)(dimY*nn)];
59      u[i][0] = u[i][1];
60    }
61     }else if (wall == 4)  //Channel flow with inflow velocity
62     {
63       for (int j = 0; j <= (int)(dimY*nn)+1; j++)
64  {
65    u[0][j] = velX;
66    v[0][j] = v[1][j];
67    u[(int)(dimX*nn)+1][j] = u[(int)(dimX*nn)][j];
68    v[(int)(dimX*nn)+1][j] = v[(int)(dimX*nn)][j];
69  }
70        for (int i = 0; i <= (int)(dimX*nn)+1; i++)
71  {
72    v[i][0] = 0.0;
73    v[i][(int)(dimY*nn)] = 0.0;
74    u[i][(int)(dimY*nn)+1] = -u[i][(int)(dimY*nn)];
75    u[i][0] = -u[i][1];
```

```
76  }
77     }else if (wall == 5)  //channel flow, peroidic bc and free-slip
78     {
79       for (int j = 0; j <= (int)(dimY*nn)+1; j++)
80  {
81    u[0][j] = u[(int)(dimX*nn)][j];
82    v[0][j] = v[(int)(dimX*nn)][j];
83    u[(int)(dimX*nn)+1][j] = u[1][j];
84    v[(int)(dimX*nn)+1][j] = v[1][j];
85  }
86       for (int i = 0; i <= (int)(dimX*nn)+1; i++)
87  {
88    v[i][(int)(dimY*nn)] = 0.0;
89    v[i][0] = 0.0;
90    u[i][(int)(dimY*nn)+1] = u[i][(int)(dimY*nn)];
91    u[i][0] = u[i][1];
92  }
93     }else if (wall == 6)  //Poiseuille flow (peroidic bc and no-slip)
94     {
95       for (int j = 0; j <= (int)(dimY*nn)+1; j++)
96  {
97    u[0][j] = u[(int)(dimX*nn)-1][j];
98    v[0][j] = v[(int)(dimX*nn)][j];
99           u[(int)(dimX*nn)][j] = u[1][j];
100   v[(int)(dimX*nn)+1][j] = v[1][j];
101 }
102      for (int i = 0; i <= (int)(dimX*nn)+1; i++)
103 {
104   v[i][(int)(dimY*nn)] = 0.0;
105   v[i][0] = 0.0;
106   u[i][(int)(dimY*nn)+1] = -u[i][(int)(dimY*nn)];
107   u[i][0] = -u[i][1];
108 }
109    }
110 }
111 /*-------------------------------------------------------------------------*/
112 /*                        BC FOR PRESSURE                                  */
113 /*-------------------------------------------------------------------------*/
```

```
114 void pBCfield(int nn, double **p)
115 {
116   int i, j;
117   double h = 1./nn;
118 
119   if (wall == 2 || wall == 3 || wall == 4 || wall == 8) //channel flow
120     {
121       for (j = 0; j <= (int)(dimY*nn)+1; j++)
122  {
123    p[0][j] = p[1][j];
124    p[(int)(dimX*nn)+1][j] = 0.0;
125  }
126       for (i = 0; i <= (int)(dimX*nn)+1; i++)
127  {
128    p[i][(int)(dimY*nn)+1] = p[i][(int)(dimY*nn)];
129    p[i][0] = p[i][1];
130  }
131     }
132   else if (wall == 5 || wall == 6 || wall == 7) //periodic BC at outflow
133     {
134       for (j = 0; j <= (int)(dimY*nn)+1; j++)
135  {
136    p[0][j] = p[(int)(dimX*nn)][j];
137    p[(int)(dimX*nn)+1][j] = p[1][j];
138  }
139       for (i = 0; i <= (int)(dimX*nn)+1; i++)
140  {
141    p[i][(int)(dimY*nn)+1] = p[i][(int)(dimY*nn)];
142    p[i][0] = p[i][1];
143  }
144     }
145 }
```

Listing B.9: results.c, functions for post-processing.

```
#include <stdio.h>
#include "head.h"
#include "input.h"
#include <stdlib.h>
#include <math.h>

void vtfFile(double **, double **, double **, double **, double **, int, double, int
    , int, double [4][nPoint]);
void setBodyIBM(double, double, double, double, double [4][nPoint], double [2][
    nPoint], double [2][nPoint]);
void gridShifting(double [2][n], int, int, int);
void intersectionP(int *, int, int, int, int [3][maxIntersec], double [3][
    maxIntersec], double [2][nPoint]);
void IBMvelocity(double **, double **, double [4][nPoint]);
/*--------------------------------------------------------------*/
/*---------------- Print to file -------------------------------*/
/*--------------------------------------------------------------*/
//Results and write out at writeOut intervall
void printFile(double t, double **u, double **v, double **p, double body[4][nPoint])
{
  double **u_new, **v_new, **p_new, **psi_new, **div_new, **psi, **u_test, **v_test;
  int step = (int)(t/dt) + 1, id = (int)(tmax/(dt*writeOut)) + 1, limit;
  double integer, fraction = modf(step/writeOut, &integer);

  u_new = dmatrix(0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
  v_new = dmatrix(0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
  p_new = dmatrix(0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
  psi_new = dmatrix(0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
  div_new = dmatrix(0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
  psi = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);

  u_test = dmatrix(0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
  v_test = dmatrix(0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
  ccopy(u_new, u_test);  ccopy(v_new, v_test);

  toZero(n-1, u_new);
  toZero(n-1, v_new);
```

```
35    toZero(n-1, p_new);
36    toZero(n-1, psi_new);
37    toZero(n-1, div_new);
38
39    if (fraction == 0.0 && step == (writeOut*integer))
40      {
41          stream(psi, u, v);
42          results(u, v, p, psi, u_new, v_new, p_new, psi_new, div_new);
43   vtfFile(u_new, v_new, p_new, div_new, psi_new, integer, t, id, writeOut, body);
44      }
45    free_dmatrix(v_test, 0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
46    free_dmatrix(u_test, 0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
47    free_dmatrix(p_new, 0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
48    free_dmatrix(u_new, 0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
49    free_dmatrix(v_new, 0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
50    free_dmatrix(psi_new, 0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
51    free_dmatrix(div_new, 0, (int)(dimX*n)+1, 0, (int)(dimY*n)+1);
52    free_dmatrix(psi, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
53  }
54  /*-------------------------------------------------------------*/
55  /*----- calculating results for u, v, p, psi and div -------*/
56  /*-------------------------------------------------------------*/
57  //Regenerating velocity vectors and pressure,
58  //removing ghost cells and averaging
59  void results(double **u, double **v, double **p, double **psi, double **u_new,
        double **v_new, double **p_new, double **psi_new, double **div_new)
60  {
61    int i, j;
62    double h = 1./(n+1.);
63    for (i = 0; i <= (int)(dimX*n); i++)//Averaging for GLview
64      {
65        for (j = 0; j <= (int)(dimY*n); j++)
66   {
67      u_new[i][j] = (u[i][j]+u[i][j+1])/2;
68      v_new[i][j] = (v[i][j]+v[i+1][j])/2;
69      p_new[i][j] = (p[i][j]+p[i][j+1]+p[i+1][j+1]+p[i+1][j])/4;
70      psi_new[i][j] = -(psi[i][j] + psi[i+1][j])/2;
71
```

```
72  }
73      }
74    for (i = 0; i <= (int)(dimX*n); i++)//Dicergence, needs u_new and v_new
75      {
76        for (j = 0; j <= (int)(dimY*n); j++)
77  {
78            div_new[i][j] = (u_new[i+1][j]-u_new[i][j])/h + (v_new[i][j+1]-v_new[i][j])
       /h;
79  }
80      }
81  }
82  /*-------------------------------------------------------------*/
83  /*----------calculating streamfunction ----------------------*/
84  /*-------------------------------------------------------------*/
85  void stream(double **psi, double **u, double **v)
86  {
87    int i, j;
88    double h = 1./n;
89    for (i = 1; i <= (int)(dimX*n); i++)
90      {
91        psi[i][0] = psi[i-1][0] - v[i][0]*h;
92      }
93
94    for (i = 0; i <= (int)(dimX*n)+1; i++)
95      {
96        for (j = 1; j <= (int)(dimY*n); j++)
97  {
98     psi[i][j] = psi[i][j-1] + u[i][j]*h;
99  }
100     }
101 }
```

Listing B.10: vtfFile.c, a function to generate the vtf-file for GLview.

```
#include <stdio.h>
#include "input.h"

void vtfFile(double **u_new, double **v_new, double **p_new, double **div_new,
     double **psi_new, int count, double t, int id, int writeOut, double body[4][
     nPoint])
{
  int i, j, time;
  double h = 1./n;
  FILE *file;

  if (count == 1)
    {
      file = fopen("glview.vtf", "w");
      fprintf(file, "*VTF-1.00 \n \n");
      fprintf(file, "*NODES 1 \n");
      for (i = 0; i <= (int)(dimX*n); i++)
 {
          for (j = 0; j <= (int)(dimY*n); j++)
     {
       fprintf(file, "%f %f %i \n", i*h, j*h, 0);
     }
 }
      if (runIBM == 1)
        {
          fprintf(file, "\n\n*NODES 2 \n");
          for (int point = 0; point < nPoint; point++)
     {
              fprintf(file, "%f %f %i \n", body[0][point], body[1][point], 0);
     }
         }
      fprintf(file, "\n*ELEMENTS 1  \n");
      fprintf(file, "%%NODES #1  \n");
      fprintf(file, "%%QUADS  \n");
      for (i = 1; i <= (int)(dimX*n); i++)
 {
   for(j = 1; j <= (int)(dimY*n) ; j++)
```

```
36      {
37        fprintf(file, "%i %i %i %i  \n", j+(i-1)*((int)(dimY*n)+1),
38   j+1+(i-1)*((int)(dimY*n)+1), j+1+i*((int)(dimY*n)+1), j+i*((int)(dimY*n)+1));
39      }
40  }
41          if (runIBM == 1)
42          {
43            if (bType == 1 || bType == 3)
44              {
45          fprintf(file, "\n*ELEMENTS 2 \n");
46          fprintf(file, "%%NODES #2 \n");
47          fprintf(file, "%%BEAMS \n");
48          for (int point = 1; point <= nPoint; point++)
49    {
50                    if (point == nPoint){fprintf(file, "%i %i \n", point, 1);}
51                    else{fprintf(file, "%i %i \n", point, point + 1);}
52                  }
53              }else if (bType == 2)
54              {
55          fprintf(file, "\n*ELEMENTS 2 \n");
56          fprintf(file, "%%NODES #2 \n");
57          fprintf(file, "%%BEAMS \n");
58          for (int point = 1; point <= nPoint/2-1; point++)
59    {
60                    fprintf(file, "%i %i\n", point, point + 1);
61                  }
62          fprintf(file, "\n*ELEMENTS 3 \n");
63          fprintf(file, "%%NODES #2 \n");
64          fprintf(file, "%%BEAMS \n");
65          for (int point = nPoint/2 + 1; point <= nPoint-1; point++)
66    {
67                    fprintf(file, "%i %i \n", point, point + 1);
68                  }
69              }
70          }
71        fprintf(file, "\n*GLVIEWGEOMETRY 1 \n");
72        fprintf(file, "%%ELEMENTS \n");
73          if (runIBM == 1)
```

```
74          {
75            if (bType == 1 || bType == 3){fprintf(file, "%i %i \n",2,1);}
76            else if (bType == 2){fprintf(file, "%i %i %i \n",3,2,1);}
77          }else
78  {
79    fprintf(file, "%i \n", 1);
80  }
81      }else
82      {
83        file = fopen("glview.vtf", "a");
84      }
85    //Pressure result for each timestep
86    fprintf(file,"\n\n*RESULTS %i \n", count);
87    fprintf(file,"%%DIMENSION 1 \n");
88    fprintf(file,"%%PER_NODE #1 \n");
89    for (i = 0; i <= (int)(dimX*n); i++)
90      {
91        for (j = 0; j <= (int)(dimY*n); j++)
92  {
93    fprintf(file,"%f \n", p_new[i][j]);
94  }
95      }
96    //Velocity result for each timestep
97    fprintf(file,"\n*RESULTS %i \n", id + count);
98    fprintf(file,"%%DIMENSION 3 \n");
99    fprintf(file,"%%PER_NODE #1 \n");
100   for (i = 0; i <= (int)(dimX*n); i++)
101     {
102       for (j = 0; j <= (int)(dimY*n); j++)
103 {
104   fprintf(file,"%f %f %f \n", u_new[i][j], v_new[i][j], 0.0);
105 }
106     }
107   //Streamfunction for each time-step
108   fprintf(file,"\n*RESULTS %i \n", (2*id) + count);
109   fprintf(file,"%%DIMENSION 1 \n");
110   fprintf(file,"%%PER_NODE #1 \n");
111   for (i = 0; i <= (int)(dimX*n); i++)
```

```
112     {
113       for (j = 0; j <= (int)(dimY*n); j++)
114  {
115    fprintf(file,"%f \n", psi_new[i][j]);
116  }
117     }
118   //Divergence for each time-step
119   fprintf(file,"\n*RESULTS %i \n", (3*id) + count);
120   fprintf(file,"%%DIMENSION 1 \n");
121   fprintf(file,"%%PER_NODE #1 \n");
122   for (i = 0; i <= (int)(dimX*n); i++)
123     {
124       for (j = 0; j <= (int)(dimY*n); j++)
125  {
126    fprintf(file,"%f \n", div_new[i][j]);
127  }
128     }
129   //If last time-step, link results to it
130   if (t > ((double)tmax-(dt*writeOut)))
131     {
132       //Pressure
133       fprintf(file,"\n*GLVIEWSCALAR 1 \n");
134       fprintf(file,"%%NAME \"PRESSURE\" \n");
135       for (time =  1; time <= count; time++)
136  {
137    fprintf(file,"%%STEP %i \n", time);
138           fprintf(file,"%%STEPNAME \"Time: %f \" \n", dt*time*writeOut);
139    fprintf(file,"%i \n", time);
140  }
141       //Velocity
142       fprintf(file,"\n*GLVIEWVECTOR 1 \n");
143       fprintf(file,"%%NAME \"VELOCITY\" \n");
144       for (time =  1; time <= count; time++)
145  {
146    fprintf(file,"%%STEP %i \n", time);
147           fprintf(file,"%%STEPNAME \"Time: %f \" \n", dt*time*writeOut);
148    fprintf(file,"%i \n", id + time);
149  }
```

```
150         //Streamfunction
151         fprintf(file,"\n*GLVIEWSCALAR 2 \n");
152         fprintf(file,"%%%NAME \"STREAMLINES\" \n");
153         for (time =  1; time <= count; time++)
154   {
155     fprintf(file,"%%%STEP %i \n", time);
156           fprintf(file,"%%%STEPNAME \"Time: %f \" \n", dt*time*writeOut);
157     fprintf(file,"%i \n", (2*id) + time);
158   }
159         //Dirvergence
160         fprintf(file,"\n*GLVIEWSCALAR 3 \n");
161         fprintf(file,"%%%NAME \"DIVERGENCE\"  \n");
162         for (time =  1; time <= count; time++)
163   {
164     fprintf(file,"%%%STEP %i \n", time);
165           fprintf(file,"%%%STEPNAME \"Time: %f \" \n", dt*time*writeOut);
166     fprintf(file,"%i \n", (3*id) + time);
167   }
168         fprintf(file," \n");
169      }
170    //Close file
171    fclose(file);
172 }
```

Listing B.11: validation.c, a function to validate the Poisson solver.

```
#include <time.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "input.h"
#include "head.h"

void twoStepM(int nn, double t, double **RHS, double **deltaP, int *iter, double
     error[ncycle], double endRes[ncycle]);
void Vcycle(int nn, double t, double **f, double **MVAL, int *iter, double error[
     ncycle], double endRes[ncycle]);
void fullMultigrid(int nn, double t, double **f, double **value, int *it, double
     error[ncycle], double endRes[ncycle]);

void validatePoisson()
{
  double h = 1./n, residual, err, endRes[ncycle], error[ncycle], x;
  double **RHS, **exact, **deltaP, **diff;
  int iter = 0, k = 2, nn, mh = method; //k = wave number
  clock_t start, end;
  double cpuTimeUsed;
  FILE *fileError, *fileIter, *fileDiff, *fileCPU, *fileSolutions, *fileRes, *
     fileGSerror;

  RHS = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  exact = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  deltaP = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
  diff = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);

  for (nn = n; nn <= n; nn = nn*2)
    {
      toZero(n, deltaP);
      toZero(n, RHS);
      toZero(n, exact);
      printf("\n nn = %i \n", nn);
      h = 1./nn, residual = 0.0, iter = 0;

```

```
34        for (int j = 1; j <= (int)(dimY*n); j++)
35  {
36    for (int i = 1; i <= (int)(dimX*n); i++)
37      {
38        //Test problem, set right hand side
39        RHS[i][j] = cos(k*M_PI*i*h)*cos(k*M_PI*j*h);
40        // The exact solution/analytical solution
41        exact[i][j] = (-1.0/(2.0*powf(k*M_PI,2)))*RHS[i][j];
42      }
43  }
44        //Set boundary conditions
45        pBCfield(nn, deltaP);
46        pBCfield(nn, exact);
47
48        if (mh == 1)    //Gauss Seidel
49  {
50    //Set clock, start
51    start = clock();
52    //Run equation slover
53    GaussSeidel(itmax, nn, RHS, deltaP, &iter, &residual, &err);
54    //Stop clock
55    end = clock();
56            printf("Tot iterations: %i \n", iter);
57            printf("Residual: %g \n", residual);
58            printf("Error: %g \n\n\n", err);
59  }else if (mh == 2) //SOR
60  {
61    //Set clock, start
62    start = clock();
63    //Run equation slover
64    SOR(nn, RHS, deltaP, &iter, &residual, &err);
65    //Stop clock
66    end = clock();
67            printf("Tot iterations: %i \n", iter);
68            printf("Residual: %g \n", residual);
69            printf("Error: %g \n\n\n", err);
70  }else if (mh == 3) //Two-step multigrid
71  {
```

```
72    //Set clock, start
73    start = clock();
74    //Run equation slover
75    twoStepM(nn, 0, RHS, deltaP, &iter, error, endRes);
76    //Stop clock
77    end = clock();
78            printf("Tot iterations: %i \n\n\n", iter);
79  }else if (mh == 4) //V-cycle multigrid
80  {
81    //Set clock, start
82    start = clock();
83    //Run equation slover
84    Vcycle(nn, 0, RHS, deltaP, &iter, error, endRes);
85    //Stop clock
86    end = clock();
87            printf("Tot iterations: %i \n\n\n", iter);
88  }else if (mh == 5) //Full multigrid
89  {
90    //Set clock, start
91    start = clock();
92    //Run equation slover
93    fullMultigrid(nn, 0, RHS, deltaP, &iter, error, endRes);
94    //Stop clock
95    end = clock();
96            printf("Tot iterations: %i \n\n\n", iter);
97  }
98        //CLOCKS_PER_SEC = the number of clock ticks per second.
99        //Calculate processor time
100       cpuTimeUsed = ((double) (end - start)) / CLOCKS_PER_SEC;
101
102       //Print to file number of iterations
103       fileIter = fopen("Validation/validation_iteration.txt", "a");
104       fprintf(fileIter, "%i %i \n", nn , iter);
105       fclose(fileIter);
106       //Print to file the processor time
107       fileCPU = fopen("Validation/validation_CPU.txt", "a");
108       fprintf(fileCPU, "%i %f \n", nn , cpuTimeUsed);
109       fclose(fileCPU);
```

```
110       //Print to file the difference between the exact solution and the numerical
      solution
111       fileDiff = fopen("Validation/validation_diff.txt", "w");
112       fileSolutions = fopen("Validation/validation_solutions.txt", "w");
113       for (int i = 1; i <= (int)(dimX*n); i++)
114  {
115           for (int j = 1; j <= (int)(dimY*n); j++)
116             {
117        diff[i][j] = exact[i][j] - deltaP[i][j];
118             }
119    x = i*h;
120 //   fprintf(fileDiff, "%f %g \n", x , diff[i][(int)(nn/2)]);
121           fprintf(fileSolutions, "%f %g %g \n", x , deltaP[i][(int)(nn/2)], exact[i
      ][(int)(nn/2)]);
122  }
123       fprintf(fileDiff, "%i %g \n", nn , RMS(nn,diff));
124       fclose(fileDiff);
125       fclose(fileSolutions);
126       //Print to file numerical error
127       fileError = fopen("Validation/validation_SOLUerror.txt", "a");
128       fprintf(fileError, "%i %g \n", nn , avg(nn, diff));
129 //      fprintf(fileError, "%i %g \n", nn , RMS(nn,diff));
130 //      fprintf(fileError, "%i %g \n", nn , maxNorm(nn,diff));
131       fclose(fileError);
132       if (mh == 3 || mh == 4)
133       {
134         //Print residual and error of each V-cycle
135         fileRes = fopen("Validation/validation_residual.txt", "w");
136         fileGSerror = fopen("Validation/validation_GSerror.txt", "w");
137         for (int c = 0; c < ncycle; c++)
138    {
139             fprintf(fileRes, "%i %g \n", c+1 , endRes[c]);
140             fprintf(fileGSerror, "%i %g \n", c+1 , error[c]);
141           }
142         fclose(fileRes);
143         fclose(fileGSerror);
144       }else{
145         fileRes = fopen("Validation/validation_residual.txt", "a");
```

```
146         fileGSerror = fopen("Validation/validation_GSerror.txt", "a");
147         fprintf(fileRes, "%i %f \n", nn, residual);
148         fprintf(fileGSerror, "%i %g \n", nn, err);
149         fclose(fileRes);
150         fclose(fileGSerror);
151       }
152     }
153   free_dmatrix(RHS, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
154   free_dmatrix(deltaP, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
155   free_dmatrix(exact, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
156   free_dmatrix(diff, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
157   return;
158 }
```

Listing B.12: criteria.c, functions for CFL-condition, RMS, max-norm etc.

```
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include "head.h"
#include "input.h"
/*--------------------------------------------------------------------*/
/*----------------- Set initial conditions ---------------------------*/
/*--------------------------------------------------------------------*/
void IC(int newIC, double **u, double **v, double **p)
{
  int i, j;
  FILE *file;
  printf("n = %i, dt = %f, Re = %i \n", n, dt, (int)(Re*2*radius));

  if (newIC == 0)
    {
      for (i = 0; i <= (int)(dimX*n)+1; i++)
        {
          for (j = 0; j <= (int)(dimY*n)+1; j++)
        {
       v[i][j] = 0.0;
       u[i][j] = 0.0;
       p[i][j] = 0.0;
     }
 }
    }
    else
    {
      file = fopen("Timestep.txt", "r");
      for (i = 0; i <= (int)(dimX*n)+1; i++)
 {
   for (j = 0; j <= (int)(dimY*n)+1; j++)
     {
       fscanf(file,"%lf %lf %lf", &u[i][j], &v[i][j], &p[i][j]);
     }
 }
      fclose(file);
```

```
38      }
39  }
40  /*-------------------------------------------------------------------*/
41  /*---------- Stability requirements ---------------------------------*/
42  /*-------------------------------------------------------------------*/
43  void stability(void)
44  {
45    double h = 1./n;
46    //Stability requirements, CFL if velocity is equals to 1.
47    if (dt > h || dt > (((double)Re*powf(h,2))/4) || dt > (2/(double)Re))
48      {
49        printf("Warning! dt should be less then %f, %f or %f, but are %f \n", h, (((
       double)Re*pow(h,2))/4) , (2/(double)Re), dt);
50        exit(EXIT_FAILURE); //Exit program
51      }
52  }
53  /*-------------------------------------------------------------------*/
54  /*---------- Maximum Norm/uMax --------------------------------------*/
55  /*-------------------------------------------------------------------*/
56  double maxNorm(int nn, double **matrix)
57  {
58    double max = 0.0;
59    for (int i = 1; i <= (int)(dimX*n); i++)
60      {
61        for (int j = 1; j <= (int)(dimY*nn); j++)
62   {
63            if (fabs(matrix[i][j]) > max){max = fabs(matrix[i][j]);}
64          }
65      }
66    return max;
67  }
68  /*-------------------------------------------------------------------*/
69  /*---------- Root Mean Square (RMS) ---------------------------------*/
70  /*-------------------------------------------------------------------*/
71  double RMS(int nn, double **x)
72  {
73    double RMS, **pow2;
74    pow2 = dmatrix(0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
```

```
75    for (int i = 1; i <= (int)(dimX*nn); i++)
76      {
77        for (int j = 1; j <= (int)(dimY*nn); j++)
78   {
79     pow2[i][j] = x[i][j]*x[i][j];
80   }
81      }
82    RMS = sqrt(sum(nn, pow2))/((int)(dimX*nn)*(int)(dimY*nn));
83    free_dmatrix(pow2, 0, (int)(dimX*n)+2, 0, (int)(dimY*n)+2);
84    return RMS;
85  }
86  /*--------------------------------------------------------------------*/
87  /*---------------------- Average -------------------------------------*/
88  /*--------------------------------------------------------------------*/
89  double avg(int nn, double **matrix)
90  {
91    double avg = 0.0;
92    for (int i = 1; i <= (int)(dimX*nn); i++)
93      {
94        for (int j = 1; j <= (int)(dimY*nn); j++)
95   {
96     avg = avg + matrix[i][j];
97   }
98      }
99    avg = avg/((int)(dimX*nn)*(int)(dimY*nn));
100   return avg;
101 }
102 /*--------------------------------------------------------------------*/
103 /*---------------------- Sum -----------------------------------------*/
104 /*--------------------------------------------------------------------*/
105 double sum(int nn, double **matrix)
106 {
107   double sum = 0.0;
108   for (int i = 1; i <= (int)(dimX*nn); i++)
109     {
110       for (int j = 1; j <= (int)(dimY*nn); j++)
111  {
112    sum = sum + matrix[i][j];
```

```
113  }
114      }
115    return sum;
116 }
117 /*-------------------------------------------------------------------*/
118 /*--------------------- CFL-condition -------------------------------*/
119 /*-------------------------------------------------------------------*/
120 void CFL(double u, double v)
121 {
122   double h = 1./n;
123   if (u*dt/h > 1 || v*dt/h > 1)
124     {
125       printf("Warning!! CFL > 1 when u*(dt/h) = %f and v*(dt/h) = %f \n", u*dt/h, v*
      dt/h);
126       exit(EXIT_FAILURE); //Exit program
127     }
128 }
```

Bibliography

Aarsnes, M. F. Navier stokes solver. My project thesis.

Balaras, E. (2004). Modeling complex boundaries using an external force field on fixed cartesian grids in large-eddy simulations. *Computers & Fluids*, 33(3):375–404.

Brandt, A. (1977). Multi-level adaptive solutions to boundary-value problems. *Mathematics of computation*, 31(138):333–390.

Brandt, A. (1981). Multigrid solvers on parallel computers.

Chorin, A. J. (1968). Numerical solution of the navier-stokes equations. *Mathematics of computation*, 22(104):745–762.

Djeddi, S. R., Masoudi, A., and Ghadimi, P. (2013). Numerical simulation of flow around diamond-shaped obstacles at low to moderate reynolds numbers. *American Journal of Applied Mathematics and Statistics*, 1(1):11–20.

Drikakis, D., Iliev, O., and Vassileva, D. (1998). A nonlinear multigrid method for the three-dimensional incompressible navier–stokes equations. *Journal of Computational Physics*, 146(1):301–321.

Euler, L. (1768). *Institutionum calculi integralis*, volume 1. imp. Acad. imp. Saènt.

Fadlun, E., Verzicco, R., Orlandi, P., and Mohd-Yusof, J. (2000). Combined immersed-boundary finite-difference methods for three-dimensional complex flow simulations. *Journal of computational physics*, 161(1):35–60.

Goldstein, D., Handler, R., and Sirovich, L. (1993). Modeling a no-slip flow boundary with an external force field. *Journal of Computational Physics*, 105(2):354–366.

Guy, R. D., Philip, B., and Griffith, B. E. (2015). Geometric multigrid for an implicit-time immersed boundary method. *Advances in Computational Mathematics*, 41(3):635–662.

Hackbusch, W. (1985). Multi-grid methods and applications, vol. 4 of springer series in computational mathematics.

Harlow, F. H., Welch, J. E., et al. (1965). Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of fluids*, 8(12):2182.

Hirt, C., Nichols, B., and Romero, N. (1975). Sola: A numerical solution algorithm for transient fluid flows. *NASA STI/Recon Technical Report N*, 75:32418.

Kahan, W. M. (1958). *Gauss-Seidel methods of solving large systems of linear equations*. PhD thesis, Thesis–University of Toronto.

Kutta, M. W. (1901). Beitrag zur näherungsweisen integration totaler differentialgleichungen. *Zeitschrift für Mathematik und Physik*.

Lai, M.-C. and Peskin, C. S. (2000). An immersed boundary method with formal second-order accuracy and reduced numerical viscosity. *Journal of Computational Physics*, 160(2):705–719.

LeVeque, R. J. (1992). *Numerical methods for conservation laws*. Springer Science & Business Media.

Mccormick, S. F. (1988). *Multigrid methods: Theory, applications, and supercomputing*. CRC Press.

Mohd-Yusof, J. (1997). For simulations of flow in complex geometries. *Annual Research Briefs*, 317.

Mossige, J. C. (2017). Numerical simulations of swimming fish. A project thesis.

Park, J., Kwon, K., and Choi, H. (1998). Numerical solutions of flow past a circular cylinder at reynolds numbers up to 160. *Journal of Mechanical Science and Technology*, 12(6):1200–1205.

Patankar, S., Liu, C., and Sparrow, E. (1977). Fully developed flow and heat transfer in ducts having streamwise-periodic variations of cross-sectional area. *Journal of Heat Transfer*, 99(2):180–186.

Peskin, C. S. (1972). Flow patterns around heart valves: a numerical method. *Journal of computational physics*, 10(2):252–271.

Peyret, R. and Taylor, T. D. Computational methods for fluid flow.

Richard H. Pletcher, John C. Tannehill, D. A. A. (2013). *Computional Fluid Mechanics and Heat Transfer*. CRC Press.

Richtmyer, R.-D. and Morton, K.-W. (1967). Difference methods for initial-value problems.

Strang, G. (2006). Linear algebra and its applications, thomson, brooks/cole, belmont, ca. Technical report, ISBN 0-030-10567-6.

Stüben, K. (1983). Algebraic multigrid (amg): experiences and comparisons. *Applied mathematics and computation*, 13(3):419–451.

Stüben, K. and Trottenberg, U. (1982). Multigrid methods: Fundamental algorithms, model problem analysis and applications. In *Multigrid methods*, pages 1–176. Springer.

Taira, K. and Colonius, T. (2007). The immersed boundary method: a projection approach. *Journal of Computational Physics*, 225(2):2118–2137.

Thompson, J. F., Soni, B. K., and Weatherill, N. P. (1998). *Handbook of grid generation*. CRC press.

Vanella, M., Posa, A., and Balaras, E. (2014). Adaptive mesh refinement for immersed boundary methods. *Journal of Fluids Engineering*, 136(4):040909.

Wesseling, P. (1995). Introduction to multigrid methods. Technical report, DTIC Document.

Wienands, R. and Oosterlee, C. W. (2001). On three-grid fourier analysis for multigrid. *SIAM Journal on Scientific Computing*, 23(2):651–671.

Xu, J. and Zikatanov, L. (2017). Algebraic multigrid methods. *Acta Numerica*, 26:591–721.

Xu, S. and Wang, Z. J. (2006). An immersed interface method for simulating the interaction of a fluid with moving boundaries. *Journal of Computational Physics*, 216(2):454–493.