Comparison of Finite Element Methods for the Navier-Stokes Equations

by

Arne Jørgen Arnesen

MASTER THESIS

for the degree of

Master in Computational Science and Engineering

Faculty of Mathematics and Natural Sciences
UNIVERSITY OF OSLO

June 25, 2010

Acknowledgment

This thesis was written in the periode August 2009 to June 2010. As a part of the requirement for the degree of Master in Computational Science and Engineering at the Faculty of Mathematics and Natural Sciences,University of Oslo.

First of all, I wish to thank my supervisors Kent-André Mardal and Kristian Valen-Sendstad for their advice and support. They have always been available for questions and discussions. Thanks to people at Simula Research Laboratory. Finally, I wish to thank all my family and friends, especially my parents Gina and Willy for constant support through the years.

Arne Jørgen Arnesen
Oslo, Juni 2010

Contents

Chapter 1

Introduction

The development of numerical methods to simulate fluid flows with applications, has been a research area of great progress over the past half-century. In Scientific Computing, numerical approaches for solving differential equations is one of the cornerstones. The goal is to find a numerical solution that approximates the solution of the differential equation in a best possible manner. A combination of accuracy and efficiency is key components in any well-developed algorithm for solving numerical problems.

We shall in this thesis study approximations of Navier-Stokes equations, which describe incompressible Newtonian viscous fluid flow. These equations are based on general conservation laws for a continuum, described in detail in Chapter 2. The main objective of this thesis is to apply numerical solution strategies and perform simulations of the Navier-Stokes equations. Today there are many open questions concerning this topic, and there exist many research groups designing new and improved numerical approaches for these equations. Due to the limited time frame, we have made a choice to concentrate on an operator-splitting approach, and a mixed finite element discretizing of the Navier-Stokes equations (Chapter 4). There are major differences in the setup of these two methods, and we want to evaluate the properties of each of them. The focus will be to measure the processor time (CPU time) and determine the accuracy using error estimates, and then compare the results.

The Finite Element Method (FEM) will be employed for solving our Computational Fluid Dynamic (CFD) problems. An important strength of the Finite Element Method is its flexibility to handle geometrically complicated domains. We will take advantage of this property when we look closer into the main application in this thesis. This is to perform aneurysm simulations of flow in blood vessels (Chapter 6.3). However, before we go into this very interesting topic, it is necessary to verify the implementations. In Chapter 3 we test our solution algorithms up against well-known theory in fluid mechanics. The simulations of the classical computational fluid dynamic problems are given in Chapter 3.1. Another important verification is the convergence of the solution when we refine

the domain. The method of manufactured solutions is applied for this purpose, and is explained further in Chapter 3.2.

It requires sophisticated software applications to deal with such a complex domain as a blood vessel. We need programs that reconstructs image-based medical data, and also generate a surface mesh that can be computed. A brief description of the software tools used in the thesis is given in Chapter 5. To implement the Navier-Stokes equations with a Finite Element Method approach, we have taken advantage of an automated solution software. The programming language applied is Python, and the Finite Element simulations are done with the FEniCS Project and its interface Dolfin.

Chapter 2

The Mathematical Model

The aim of this chapter is to formally derive equations governing the motion of an incompressible Newtonian fluid. In the literature, the resulting equations are often referred to as the incompressible Navier-Stokes equations. The equations are derived from the principles of conservation of mass and momentum, and we will apply the Reynolds transport theorem in the formulation of these conservation principles.

The general case of the Reynolds transport theorem from [11] see e.g. states that,

$$\frac{d}{dt}\int_{V(t)} \mathbf{f}\, dV = \int_{V(t)} \frac{\partial \mathbf{f}}{\partial t}\, dV + \int_{\partial V(t)} (\,\mathbf{v}\cdot\mathbf{n}\,)\mathbf{f}\, ds. \tag{2.1}$$

Where $V(t)$ is neither a fixed volume or a material volume, with boundary surface $\partial V(t)$ moving. Here $\mathbf{n}$ is the outward unit normal, $\mathbf{v}$ represent the velocity field at the moving boundary, and $\mathbf{f}(\mathbf{x}, t)$ is a vector field.

For a *fixed volume* (2.1) becomes,

$$\frac{d}{dt}\int_{V} \mathbf{f}\, dV = \int_{V} \frac{\partial \mathbf{f}}{\partial t}\, dV, \tag{2.2}$$

since $\mathbf{v} = 0$ at the fixed boundary, and V is not a function of time in this case.

2.1 Conservation of mass

Consider a fixed domain occupied by a fluid inside a closed surface $\partial\Omega$ enclosing Ω. The velocity of the fluid can be described by a vector field $\mathbf{u}(\mathbf{x}, t)$ and the density $\rho(\mathbf{x}, t)$. The total mass of the fluid inside Ω is $\int_\Omega \rho\, d\Omega$. The rate of change of mass inside Ω is given by,

$$\frac{d}{dt}\int_{\Omega} \rho\, d\Omega = \int_{\Omega} \frac{\partial \rho}{\partial t}\, d\Omega, \tag{2.3}$$

since the volume is fixed (2.2) is valid.
The amount of fluid flowing out of Ω through $\partial\Omega$ is given by,

$$\int_{\partial\Omega} \rho\mathbf{u} \cdot \mathbf{n}\, ds. \tag{2.4}$$

Here $\mathbf{n}$ is the outward unit normal vector of the surface.

The principle of conservation of mass states that:

The time rate of change of mass in a fixed Ω equals the amount of fluid flowing through $\partial\Omega$.

Mathematically expressed in a integral form for a fixed domain;

$$\int_{\Omega} \frac{\partial \rho}{\partial t}\, d\Omega = -\int_{\partial\Omega} \rho\mathbf{u} \cdot \mathbf{n}\, ds. \tag{2.5}$$

The divergence theorem may be applied to the surface integral in (2.5), changing it into a volume integral,

$$\int_{\Omega} \frac{\partial \rho}{\partial t} + \nabla \cdot (\, \rho\mathbf{u}\,)\, d\Omega = 0. \tag{2.6}$$

Since this is valid for an arbitrary domain Ω it follows that,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\, \rho\mathbf{u}\,) = 0 \tag{2.7}$$

which is called the *continuity equation* and expresses the differential form of the principle of conservation of mass.

For the special case of an incompressible fluid flow (density constant), the *continuity equation* reduces to:

$$\mathrm{div}(\mathbf{u}) = \nabla \cdot \mathbf{u} = 0 \tag{2.8}$$

2.2 Conservation of momentum

Conservation of momentum also commonly known as Newton′s second law $\mathbf{F} = m\mathbf{a}$. According to this law, the time rate of change of momentum is equal to the sum of forces $\mathbf{F}$ acting on the fluid. Writing momentum as $\int_\Omega \rho\mathbf{u}\, d\Omega$ we get,

$$\frac{D}{Dt} \int_{\Omega} \rho\mathbf{u}\, d\Omega = F. \tag{2.9}$$

The particle derivative operator is defined as,

$$\frac{D(\cdot)}{Dt} := \frac{\partial(\cdot)}{\partial t} + (\,\mathbf{u}\cdot\nabla\,)(\cdot) \tag{2.10}$$

where $\mathbf{u}$ is the velocity of the fluid.

We divide the sum of forces F, into two new terms:

- *volume forces* (ex: gravity)
- *surface forces*(ex: stress)

Then (2.9) becomes;

$$\frac{D}{Dt}\int_\Omega \rho\mathbf{u}\,d\Omega = \int_{\partial\Omega}\sigma\cdot\mathbf{n}\,ds + \int_\Omega \rho f\,d\Omega \tag{2.11}$$

where σ is the stress tensor, $\mathbf{f}$ present gravity and $\mathbf{n}$ is the outward unit normal vector.

The divergence theorem may be applied to the surface integral (2.11), changing it into a volume integral, and in addition apply the particle derivative (2.10) for a fixed volume,

$$\int_\Omega \frac{\partial}{\partial t}(\,\rho\mathbf{u}\,) + (\,\mathbf{u}\cdot\nabla\,)(\,\rho\mathbf{u}\,)\,d\Omega = \int_\Omega \big(\nabla\cdot\sigma + \rho f\big)\,d\Omega \tag{2.12}$$

where (2.12) represents a system of 3 equations with 12 unknown. For further derivation we need some information about the stress tensor σ.

For a Newtonian viscous fluid we split the stress tensor σ into normal stresses and shear stresses τ, and we have a linear coupling between stress and strain called *the constitutive law for a Newtonian fluid*. The stress tensor can then be seen as;

$$\sigma = -pI + \tau = (-p + \lambda\nabla\cdot\mathbf{u})I + 2\mu\epsilon,$$

where

$$\epsilon = \frac{1}{2}[\,\nabla\mathbf{u} + (\,\nabla\mathbf{u}\,)^T]. \tag{2.13}$$

Here ϵ is the strain tensor, p is the pressure and μ, λ are parameters describing viscosity.

For the special case of an incompressible fluid, the strain tensor (2.13) is simplified, since the viscosity parameters μ, λ will be constant. Also applying the continuity property (2.8), we get;

$$\nabla\cdot\sigma = \nabla\cdot(-p + \lambda\nabla\cdot\mathbf{u})I + \nabla\cdot 2\mu\epsilon = -\nabla p + \mu\Delta\mathbf{u}. \tag{2.14}$$

Now, setting the simplified expression for $\nabla \cdot \sigma$ into (2.12),

$$\int_\Omega \rho(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u}) \, d\Omega = \int_\Omega -\nabla p + \mu \nabla^2 \mathbf{u} + \rho f \, d\Omega \tag{2.15}$$

where *density* ρ constant. Since this is valid for an arbitrary domain Ω it follows that,

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} &= -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + f & (2.16) \\ \nabla \cdot \mathbf{u} &= 0 & (2.17) \end{aligned}$$

here $\nu = \frac{\mu}{\rho}$ is the kinematic viscosity.

We have derived the Navier-Stokes equations above for an incompressible Newtonian viscous fluid, from the conservation laws of mass and momentum.

Chapter 3

Test problems and Exact solutions

The Navier-Stokes equations can be solved exactly for very simple cases. Under certain assumptions, existence and uniqueness of weak solutions exists. The problem is that there is no general mathematical theory for these equations. This is called the Navier- Stokes existence and smoothness problem, and are one of the *Millennium Prize Problems* [7]. This means that to find general solutions and existence of these, is far beyond the scope of the present thesis :)

The purpose of this chapter is to perform simulations of some selected test problems, and verify the numerical methods that are implemented. Our first experiments concerns verification of correct flow structure of some classical computational fluid dynamic problems, known from the theory of fluid mechanics.

Next we will apply the method of manufactured solutions. This method is based on constructing an artificial known solution. Then inserted into the equation of interest to reproduce the known solution. Declaration of the manufactured solutions are given in Chapter 3.2, while the simulation results are moved to Chapter 6.2. The simulations below are based on the numerical methods described in Chapter 4.

3.1 Classical CFD Problems

3.1.1 Plane Poiseuille Flow

Plane Poiseuille flow describes the laminar flow of a viscous fluid between parallel plates, with a pressure drop along the length of the plates. The fluid flows from high to low pressure with no-slip condition (fluid have zero velocity relative to the boundary) on the plates, exerting a shear stress on the plates in the direction of the flow.

Observe from the simulation in Figure 3.1 that the velocity profile is parabolic, and the fluid achieves the highest speed in the center between the two plates. This

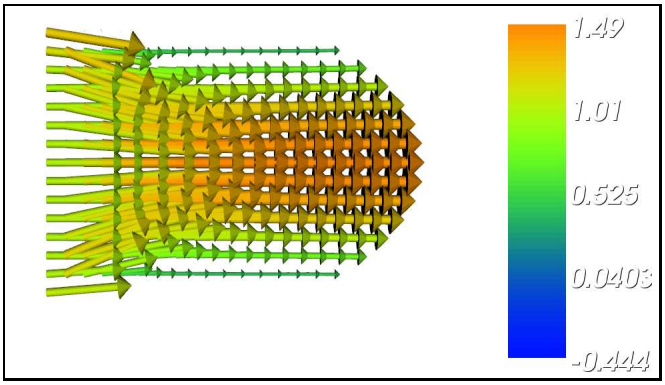


Figure 3.1: Plane Poiseuille flow driven by an externally imposed pressure gradient ($\frac{dp}{dx} < 0$) on a 2D domain $\Omega = [0, 1] \times [0, 1]$, between two stationary plats.

confirms the theory of Chapter 9.4 in [11].

3.1.2 Driven Cavity

Imagine a square infinitely long container filled with a viscous fluid, and the fluid is initially at rest. We have simulated the flow of an incompressible viscous fluid in a 2D square domain for the time $t \in [0, T]$. Here we have $T = 10.0$ in this case. The lid (top plate) of the container is instantaneously set from zero to a certain velocity, and this will launch the fluid movement. Imposes no-slip conditions on all boundary walls with the exception of the top boundary that moves in x-direction at speed 1.0.

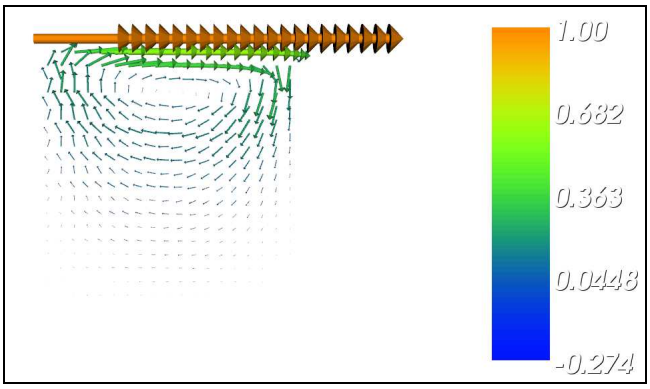


Figure 3.2: Lid driven cavity flow by a top plate moving in x- direction, at final steady state and $\nu = 1.0$ on a 2D domain $\Omega = [0, 1] \times [0, 1]$.

Figure 3.2 and Figure 3.3 show the velocity field for two different viscosities($\nu = 1.0, 0.1$) at a given time. Observe the difference in the flow structure of the two figures. From literature in Chapter 5.1 in [6], the simulations are verified.

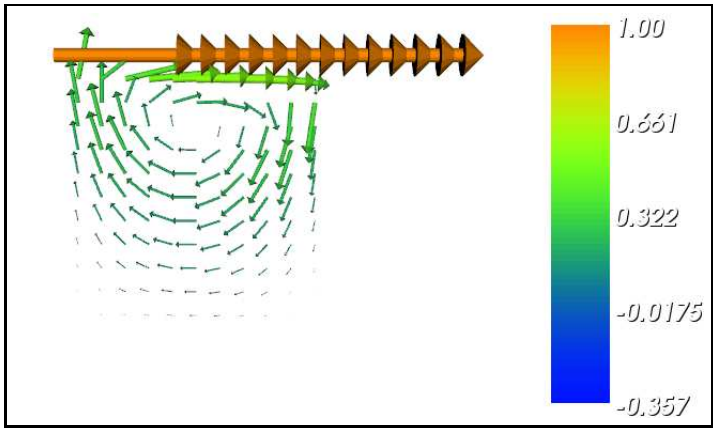


Figure 3.3: Lid driven cavity flow by a top plate moving in x- direction, at final steady state and $\nu = 0.1$ on a 2D domain $\Omega = [0,1] \times [0,1]$.

3.1.3 Plane Couette Flow

Plane Couette flow is a laminar flow of a viscous fluid. It is a steady flow between two parallel plates where the bottom plate is stationary while the top plate moves with a constant velocity in x- direction. In the simulated case below the pressure gradient is zero and the only force on the fluid is due to the moving top plate i.e the flow driven by the motion of the upper plate. This can of course be generalized by applying a pressure gradient in the direction parallel to the plates.

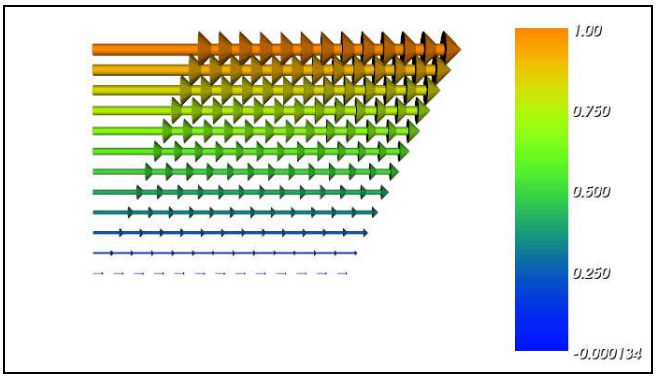


Figure 3.4: Plane Couette flow driven by an externally imposed moving top plate, without externally pressure gradient ($\frac{dp}{dx} = 0$), between two stationary plats on a 2D domain $\Omega = [0,1] \times [0,1]$.

Observe from the simulation Figure 3.4 that the viscosity at the top and bottom plates makes the fluid stick to the boundary which is why a shear develops within the interior of the fluid. The top plate moves and the bottom plate have

no-slip condition. From theory (Chapter 9.4 in [11]) the exact solution of Couette problems reduce to a linear velocity profile $u(y) = y\,U$, where U represent velocity in x- direction. In our case $U = 1.0$.

Results:

The simulations of the three test problems confirm the correct flow structure compared with the theory in fluid mechanics [6], [11] and [17]. We are pleased with this result, and continues with verification of convergence in the next section.

3.2 Manufactured solutions

The method of manufactured solution is a technique by which numerical methods can be verified to ensure that the implementation have been coded correctly. The method is based on selecting an artificial solution which is adapted such that it satisfies the governing equation. In our case it is required that the solution is chosen divergence free. Then we insert the artificial solution into the equation, and from this we can determine a source term $\mathbf{f}$ (to calculate $\mathbf{f}$, see Chapter 3.2.1). By inserting the source term into the equation, we can solve it.The purpose is to recreate the artificial solution, and achieve convergence as we refines the grid resolution.

3.2.1 2D Exampel

The purpose is to solve the two following problems with the method of manufactured solutions, and hopefully achieve convergence.

The time dependent Stokes Problem,

$$\begin{aligned} \mathbf{u}_t - \nu\Delta\mathbf{u} - \nabla p &= \mathbf{f} \\ \nabla\cdot\mathbf{u} &= 0 \end{aligned} \tag{3.1}$$

and the Navier- Stokes Problem,

$$\begin{aligned} \mathbf{u}_t + (\mathbf{u}\cdot\nabla\mathbf{u}) - \nu\Delta\mathbf{u} - \nabla p &= \mathbf{f} \\ \nabla\cdot\mathbf{u} &= 0. \end{aligned} \tag{3.2}$$

The artificial solution is defined as follows,

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} sin(y) \\ sin(x) \end{bmatrix} \tag{3.3}$$

and

$$\nabla p = -p_{grad}$$

Observe from (3.3) that the velocity solution is divergence free. The pressure gradient $\nabla p = 1.0$.

We can now calculate the source terms;

$$\mathbf{f} = \mathbf{u}_t - \nu\Delta\mathbf{u} - \nabla p = \begin{bmatrix} \nu\, sin(y) + p_{grad} \\ \nu\, sin(x) \end{bmatrix} \tag{3.4}$$

$$\mathbf{f} = \mathbf{u}_t + (\mathbf{u}\cdot\nabla\mathbf{u}) - \nu\Delta\mathbf{u} - \nabla p = \begin{bmatrix} sin(x)\, cos(y) + \nu\, sin(y) + p_{grad} \\ sin(y)\, cos(x) + \nu\, sin(x) \end{bmatrix} \tag{3.5}$$

Results of the simulations are printed in Chapter 6.2.

3.2.2 3D Exampel

The artificial solution in the 3D case is defined as follows,

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} sin(y) \\ sin(x) \\ 0 \end{bmatrix} \tag{3.6}$$

and

$$\nabla p = -p_{grad}$$

Observe from (3.6) that the velocity solution is divergence free. The pressure gradient $\nabla p = 1.0$.

We can then calculate the source terms;

$$\mathbf{f} = \mathbf{u}_t - \nu\Delta\mathbf{u} - \nabla p = \begin{bmatrix} \nu\, sin(y) + p_{grad} \\ \nu\, sin(x) \\ 0 \end{bmatrix} \tag{3.7}$$

$$\mathbf{f} = \mathbf{u}_t + (\mathbf{u}\cdot\nabla\mathbf{u}) - \nu\Delta\mathbf{u} - \nabla p = \begin{bmatrix} sin(x)\, cos(y) + \nu\, sin(y) + p_{grad} \\ sin(y)\, cos(x) + \nu\, sin(x) \\ 0 \end{bmatrix} \tag{3.8}$$

Results of the simulations are printed in Chapter 6.2.

Chapter 4

Numerical methods

The established model for viscous Newtonian incompressible fluid flow is given by the Navier-Stokes equations derived earlier. The aim of this chapter is to apply numerical solution strategies for solving this set of equations. Performing computational modeling on a physical problem where high precision is essential, requires large computer resources. This is due to large linear systems that may contain millions of unknown.

Development of numerical methods for incompressible viscous fluid flow is a field of great progress these days. This can be seen as an underlying topic of computational fluid dynamics (CFD), which is very important in modern industry and science. The search for a numerical method that is efficient, stable and at the same time can handle complex geometries are important properties in this context. The Finite Element Method (FEM) will be discussed in the next section, and this method is very flexible and can be adopted to complicated domains. It also has a very smooth setup.

Further, two common ways of discretizing the Navier-Stokes equations will be introduced and derived. To speed up the solution algorithms we will discuss the incorporation of iterative solution methods and preconditioning, and finally present an order-optimal method for large systems of linear equations.

4.1 Finite Element Method

In this section we will apply the finite element method on the Navier-Stokes equations,

$$\frac{\partial \mathbf{u}}{\partial t} + (\,\mathbf{u}\cdot\nabla\,)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} + \mathbf{f} \quad \text{in } \Omega \tag{4.1}$$

$$\nabla\cdot\mathbf{u} = 0 \quad \text{in } \Omega, \tag{4.2}$$

combined with proper boundary conditions.

4.1.1 Weak Formulation

The weak formulation is based on transforming (4.1) and (4.2) into a variational problem, also called the weak formulation.This transformation is mainly done because in many cases we can not find a sufficiently smooth enough *classical solution*, satisfying the differential equation we want to solve in the proper space. A clever way to find solutions for such problems that arise frequently in applied mathematics, is to seek the weak solution of the problem. We seek an integral solution with lower regularity in a less restricted space. With this formulation we require the use of function spaces, specifically Sobolev function spaces. For a more detailed description of Sobolev spaces and their properties see [5].

Definitions and Notation needed further in the document

.

- Let $\mathbf{X}$ be a real linear space. A mapping $||\ ||: \mathbf{X} \to \big[0, \infty\big)$ is called a *norm* if,
  i) $||u+v|| \leq ||u|| + ||v||\ \ \forall\ u, v \in \mathbf{X}$.
  ii) $||\alpha u|| = |\alpha|\, ||u||\ \ \forall\ u \in \mathbf{X},\ \alpha \in \mathbb{R}$.
  iii) $||u|| = 0$ if and only if $u = 0$.

- Let $\mathbf{V}$ be a real linear space. A mapping $(\ ,\ ) : \mathbf{V} \times \mathbf{V} \to \mathbb{R}$ is called an *inner product* if,
  i) *symmetric* i.e $(u, v) = (v, u)\ \forall\ u, v \in \mathbf{V}$.
  ii) *bilinear*
  iii) $(u, u) \geq 0\ \forall\ u \in \mathbf{V}$.
  iv) $(u, u) = 0$ if and only if $u = 0$.

- The associated *norm* and *inner product* is $||u|| = (u, u)^{\frac{1}{2}}$, and the Cauchy- Schwarz inequality states $|(u, v)| \leq ||u||\, ||v||$.

- A *Hilbert space* H is a Banach space (complete, normed linear space) provided with an inner product which generates the norm.

- H^m is a Sobolev space of functions on Ω with m derivatives in L^2 and we use the simpler notation $||\cdot||_m$ instead of $||\cdot||_{H^m}$.

- The *dual space* (see [5]) of H_0^m with respect to the L^2 inner product will be denoted by H^{-m}.

- The spaces involving time are defined by,

$$||u||_{L^2(0,T;X)} = \Big(\int_0^T ||u(t)||_X^2\, dt \Big)^{\frac{1}{2}},$$

where $||\cdot||_X$ is the spatial norm in the space X.

Inner products

$$(u, v) = \int_\Omega u \cdot v \, dx \tag{4.3}$$

$$(\mathbf{u}, \mathbf{v}) = \int_\Omega \mathbf{u} \cdot \mathbf{v} \, dx \tag{4.4}$$

Lebesque and Sobolev function spaces

$$L^2(\Omega) := \left\{ u : \Omega \to \mathbb{R} \,\middle|\, \int_\Omega u^2 < \infty \right\} \tag{4.5}$$

$$\mathbf{L^2(\Omega)} := \left\{ \mathbf{u} : \Omega \to \mathbb{R}^d \,\middle|\, u_i \in L^2 \;\; \forall \; i \right\} \tag{4.6}$$

$$H^1(\Omega) := \left\{ u : \Omega \to \mathbb{R} \,\middle|\, \int_\Omega (u^2 + \nabla u^2) < \infty \right\} \tag{4.7}$$

$$\mathbf{H^1(\Omega)} := \left\{ \mathbf{u} : \Omega \to \mathbb{R}^d \,\middle|\, u_i \in H^1 \;\; \forall \; i \right\} \tag{4.8}$$

$$L_0^2 := \left\{ u \in L^2 \,\middle|\, \int_\Omega u = 0 \right\} \tag{4.9}$$

$$H_0^1 := \left\{ u \in H^1 \,\middle|\, u = 0 \text{ on } \partial\Omega \right\} \tag{4.10}$$

Function Spaces

In general when we look at a continuous problem, we devote the domain (values for which the function is defined) by Ω. In the weak formulation we now assume V to be a function space where we seek our solution of the variational problem. Later as we will see in the FEM formulation, V is replaced by the finite dimensional subspace V_h. Where the parameter h represents the scale of the discretization. This discrete function space V_h, is usually in FEM formulation constructed by low degree piecewise continuous polynomials.

To derive a weak formulation of the Navier-Stokes equations we require a set of test functions $\mathbf{v}$ and a set of test functions q, chosen respectively in proper function spaces. By multiplying (4.1) by a test function $\mathbf{v}$ and (4.2) by a test

function q, and integrate over the domain Ω we get,

$$\begin{aligned}\int_\Omega \Big(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\Big)\cdot\mathbf{v}\,d\Omega &= \int_\Omega \Big(-\frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u}\Big)\cdot\mathbf{v}\,d\Omega + \int_\Omega \mathbf{f}\cdot\mathbf{v}\,d\Omega \\ \int_\Omega q\Big(\nabla\cdot\mathbf{u}\Big)\,d\Omega &= 0.\end{aligned}$$

Performing integration by part of the first term on the right hand side of the momentum equation,

$$\begin{aligned}-\int_\Omega \frac{1}{\rho}\nabla p\cdot\mathbf{v}\,d\Omega &= \frac{1}{\rho}\Big(\int_\Omega p\,\nabla\cdot\mathbf{v}\,d\Omega - \int_{\partial\Omega} p\mathbf{v}\cdot\mathbf{n}\,ds\Big) \\ \int_\Omega \nu\nabla^2\mathbf{u}\cdot\mathbf{v}\,d\Omega &= \nu\Big(-\int_\Omega \nabla\mathbf{u} : \nabla\mathbf{v}\,d\Omega + \int_{\partial\Omega}\frac{\partial\mathbf{u}}{\partial\mathbf{n}}\mathbf{v}\,ds\Big)\end{aligned}$$

Where $\mathbf{n}$ is the outward unit normal to $\partial\Omega$. We end up with the following,

$$\begin{aligned}(\frac{\partial\mathbf{u}}{\partial t},\mathbf{v}) + (\mathbf{u}\cdot\nabla\mathbf{u},\mathbf{v}) + \nu(\nabla\mathbf{u},\nabla\mathbf{v}) - \frac{1}{\rho}(p,\nabla\cdot\mathbf{v}) &= (\mathbf{f},\mathbf{v}) + \nu(\frac{\partial\mathbf{u}}{\partial\mathbf{n}},\mathbf{v})_{\partial\Omega} - \frac{1}{\rho}(p\mathbf{n},\mathbf{v})_{\partial\Omega} \\ (q,\nabla\cdot\mathbf{u}) &= 0. \qquad (4.11)\end{aligned}$$

Observe now, the integration by part has lowered the degree on the differential operators for both unknown $\mathbf{u}$ and p. Thus we can seek a solution in a larger space, with less regularity than in the original problem (4.1) and (4.2).

With a proper choice of test functions with appropriate boundary conditions defined on the boundary, we can simplify the expression (4.11) even more.By introducing an abstract formulation, we can rewrite the problem as follows with homogeneous boundary conditions.

Find $\mathbf{u}\in L^2(0,T;[H_0^1(\Omega)]^d)$, *with* $\mathbf{u}'\in L^2(0,T;[H^{-1}(\Omega)]^d)$ *and* $p\in L^2(0,T;L_0^2(\Omega))$ *such that,*

$$\begin{aligned}a(\mathbf{u},\mathbf{v}) + b(p,\mathbf{v}) &= f(\mathbf{v}), \quad \forall\ \mathbf{v}\in[H_0^1(\Omega)]^d, \quad \text{a.e.}\quad t\in[0,T] \qquad (4.12)\\ b(q,\mathbf{u}) &= 0, \quad \forall\ q\in L_0^2(\Omega), \quad \text{a.e.}\quad t\in[0,T] \qquad (4.13)\end{aligned}$$

where

$$\begin{aligned}a(\mathbf{u},\mathbf{v}) &= (\frac{\partial\mathbf{u}}{\partial t},\mathbf{v}) + (\mathbf{u}\cdot\nabla\mathbf{u},\mathbf{v}) + \nu(\nabla\mathbf{u},\nabla\mathbf{v}) \\ b(p,\mathbf{v}) &= (p,\nabla\cdot\mathbf{v}) \\ f(\mathbf{v}) &= (f,\mathbf{v}).\end{aligned}$$

Observe that the boundary-value problem (4.12), (4.13) is defined such that the boundary terms in (4.11) vanish on $\partial\Omega$. This is because we have chosen the appropriate function spaces H_0^1 and L_0^2.

To summarize, (4.12) and (4.13) has a possible solution called the weak solution that does not need to be a classical solution. However, if there exist a classical solution of the problem above, then (4.1) and (4.2) is equivalent to (4.12) and (4.13) when we require homogeneous boundary conditions. The procedure of deriving the weak formulation in this section, is intuitive and easy to extend to the finite element formulation in the next section.

4.1.2 FEM Formulation

The purpose in this section is to find an approximated solution in finite dimensional subspaces of the weak formulation from the previous section. The discretized version will be an intuitive transition from the established weak formulation.

In finite element literature the unknown function to be approximated is referred to as a trial function, and the function we multiply the equation with is called a test function. In general, the continuous domain Ω is partitioned into a number of cells such that $\Omega_h = \bigcup_{e=1}^{N_e} \Omega_e$, where Ω_h is an approximation of Ω. This is also called a grid. The parameter h represents the scale of the discretization, and determines the grid refinement. Grid vertices are denoted x_i. The cells are typically shaped as simple polygons such as intervals(1D), triangles(2D) or tetrahedra(3D). However, other shapes are also possible.

A key point in the finite element method is to specify suitable function spaces for the test and trial functions. The function spaces are constructed of lower order polynomials. More precisely, they are based on piecewise continuous low order polynomials over a single cell Ω_e. The composition of a single cell with a suitable chosen function space with some specific properties is called a *finite element* [2]. The most widely used finite elements are the Lagrange elements [9], and Figure 4.1 and Figure 4.2 show respectively first order and second order Lagrange finite elements.

Suppose that we have $V_h \subset H_0^1$ and $Q_h \subset L_0^2$ with respective finite dimensions N_V and N_Q, and assume further that these subspaces consists of linearly independent functions which forms a basis. We can then seek approximations of $\mathbf{u}$ and p respectively in the subspaces V_h and Q_h. Expressed as a linear combination of basis functions $\mathbf{v}_i$ spanning the space $[V_h]^d$ and q_i spanning the space Q_h such

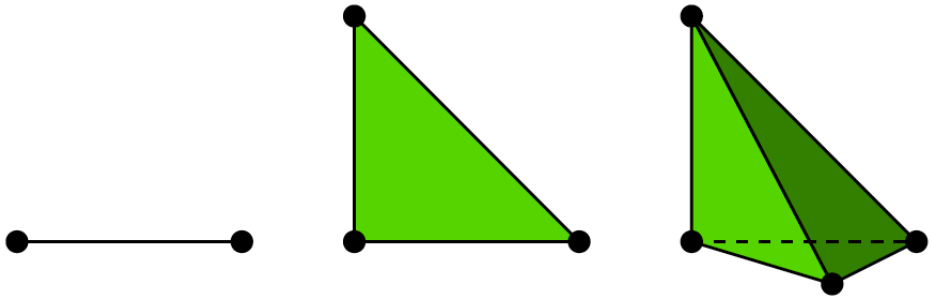


Figure 4.1: First order Lagrange finite elements in 1D,2D and 3D.

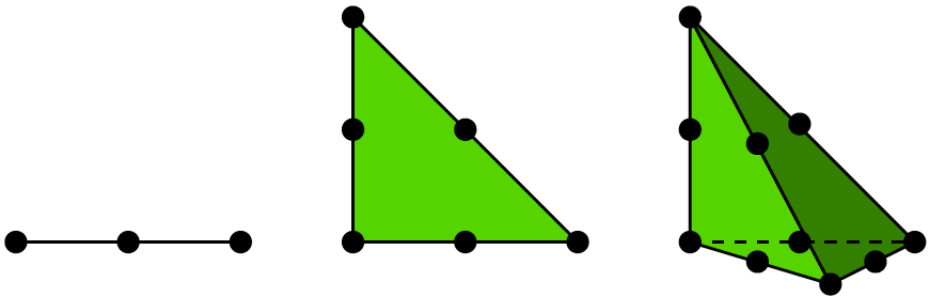


Figure 4.2: Second order Lagrange finite elements in 1D,2D and 3D.

that,

$$\mathbf{u} \approx \mathbf{u}_h(t) = \sum_{i=1}^{N_V} \mathbf{u}_i(t)\mathbf{v}_i(x) \tag{4.14}$$

$$p \approx p_h(t) = \sum_{i=1}^{N_Q} p_i(t)q_i(x). \tag{4.15}$$

The finite element formulation is defined as follows:

Find $\mathbf{u}_h \in L^2(0,T;[V_h(\Omega)]^d)$, *with* $\mathbf{u}'_h \in L^2(0,T;[V_h(\Omega)]^d)$ *and* $p_h \in L^2(0,T;Q_h(\Omega))$ *such that*

$$a(\mathbf{u}_h,\mathbf{v}_h) + b(p_h,\mathbf{v}_h) = f(\mathbf{v}_h), \quad \forall\ \mathbf{v}_h \in [V_h(\Omega)]^d, \quad \text{a.e.} \quad t \in [0,T] \tag{4.16}$$

$$b(q_h,\mathbf{u}_h) = 0, \quad \forall\ q_h \in Q_h, \quad \text{a.e.} \quad t \in [0,T] \tag{4.17}$$

By performing a spatial discretization we can write the element formulation as a block system,

$$\begin{bmatrix} \mathbf{u}_t \\ 0 \end{bmatrix} - \begin{bmatrix} N(\mathbf{u}) & Q \\ Q^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ 0 \end{bmatrix} \tag{4.18}$$

where

$$\begin{aligned} N(\mathbf{u}) &= \nu\nabla^2 - \mathbf{u}\cdot\nabla \\ Q &= \nabla \\ n(\mathbf{u},\mathbf{v}) &= \nu(\nabla\mathbf{u},\nabla\mathbf{v}) + (\mathbf{u}\cdot\nabla\mathbf{u},\mathbf{v}) \\ q(p,\mathbf{v}) &= (p,\nabla\cdot\mathbf{v}). \end{aligned}$$

The Differential Algebraic Equations (DAEs) (4.18), can be complicated to solve since the system may be singular! There are several ways to cure this problem e.g. selection of specific spatial discretization or stabilization techniques to ensure that the system is solvable. Topics as mixed finite elements and the Brezzi conditions will be discussed later in the chapter. However, first we want to motivate a family of methods that avoid this singular problem.

4.2 Projection Algorithms

The projection algorithms is a type of operator splitting algorithms. It is a family of numerical methods for solving the Navier-Stokes equations numerically. One main difficulty with solving the incompressible Navier-Stokes equations is the handling of the pressure term and the incompressibility constraint. The projection approaches deals with these problems by splitting the full complicated problem of Navier-Stokes equations (4.1) and (4.2) into smaller simpler equations, and solve these in an efficient manner. This technique splits the differential operators such that we ends up with solving a vector convection-diffusion equation and Poisson like equations, instead of the Navier-Stokes equations. The methods makes a discretizing in time before space, and then discretizing the time- discrete equations in space.

Typically, the algorithms consists of two steps. In the first step we neglect the incompressibility restriction and computes a tentative velocity. Second step is to make a projection out of a corrected velocity built on the pressure onto a divergence free vector velocity field, to get the updated velocity and pressure.

4.2.1 Semi Implicit Projection Method

We will now go through a widely used technique for simulating incompressible viscous fluid flow. This is an operator-splitting approach in combination with finite element method. Observe that we discretize in time prior space.

By choosing an backward semi implicit convection Euler scheme in time we get a fairly robust and stable method compared to an explicit version, where there is a restriction on the time step length,

$$v^* = -\Delta t\, v^l \cdot \nabla v^* - \frac{\Delta t}{\rho}\beta \nabla p^l + \Delta t \nu \Delta v^* + v^l + \Delta t g^{l+1}. \tag{4.19}$$

Here we are solving for a tentative velocity v^*. This v^* generally does not obey the continuity part of the Navier-Stokes equations, so we must apply a correction velocity later such as our final velocity satisfies the continuity equation. Also note the linearization in the convection term, and observe from the linearization that (4.19) turns out to be a *convection-diffusion equation*. The parameter $\beta \in [0, 1]$ is used to adjust the amount of the "*old*" pressure information see [13].

Next, setup the equation for v^{l+1} that we demand to fulfill the continuity equation,

$$v^{l+1} = -\Delta t\, v^l \cdot \nabla v^{l+1} - \frac{\Delta t}{\rho}\nabla p^{l+1} + \Delta t \nu \Delta v^{l+1} + v^l + \Delta t g^{l+1}. \tag{4.20}$$

Now, we look more closely at two features that are significant to this type of methods.

1. Define a correction velocity such that,
   $v^c = v^{l+1} - v^*$.

2. The condition on the final velocity v^{l+1} such that,
   $\nabla \cdot v^{l+1} = 0$.

Subtracting (4.19) from (4.20) yields an expression for v^c,

$$\begin{aligned} v^c &= \Delta t(\, -v^l \cdot \nabla v^c + \nu \Delta v^c\,) - \frac{\Delta t}{\rho}\nabla(p^{l+1} - \beta p^l) \\ &= S(v^c) - \frac{\Delta t}{\rho}\nabla(p^{l+1} - \beta p^l) \\ \nabla \cdot v^c &= -\nabla \cdot v^*. \end{aligned} \tag{4.21}$$

Note that in the last equation, we have used the properties of the two features above. It is a common simplification to neglect the $S(v^c)$ term from (4.21). By eliminating v^c from the first equality in (4.21), we turn it into a *Poisson equation*,

$$\Delta \phi = \Delta(p^{l+1} - \beta p^l) = \frac{\rho}{\Delta t}\nabla \cdot v^*. \tag{4.22}$$

After computed the Poisson problem (4.22), we perform an update on the pressure and the velocity,

$$p^{l+1} = \beta p^l + \phi \tag{4.23}$$

$$v^{l+1} = v^* - \frac{\Delta t}{\rho}\nabla\phi. \tag{4.24}$$

4.2.2 Results and Error Estimate Projection Method

We have reduced the incompressible Navier-Stokes equations to simpler standard PDE problems. These problems can be solved efficiently numerically, and is easy to implement from a programmers point of view. Another advantage of this class of methods is that we can apply standard finite elements. The velocity and pressure can be represented using identical basis functions. This feature simplifies the program code dramatically compared to implementation of the finite element method with mixed finite elements, see Chapter 4.3. For mixed finite elements discretizing it is favorable to apply some sophisticated numerical software tool like Diffpack or FEniCS.

The drawback for the algorithm derived above, is that we demand more boundary conditions on the pressure than what is required in the original Navier-Stokes equations. This can create some problems. Observe that the reason for the unnatural boundary conditions is a cause of dropping the $S(v^c)$ term in (4.21). From this omission, we derived a Poisson equation (4.22) for the pressure difference ϕ. A suitable boundary conditions to ϕ must be known at the whole boundary to solve a Poisson problem. On the other hand, the pressure only needs to be specified as a function of time when solving the Navier-Stokes equations. In other words, these kinds of simplifications introduce non compatible boundary conditions with the original system (4.1) and (4.2), and can produce large errors in the pressure near the boundaries.

The problem with unnatural boundary conditions that arise when dropping the $S(v^c)$ term can be justified. If we keep the $S(v^c)$ term in (4.21), we end up with solving a much more complicated problem. This will be a stationary Stokes problem also called a saddle point problem.

For the implementation of the algorithm outlined above, see Appendix A.2. The Finite Element simulations are done with The FEniCS Project and its interface Dolfin. Observe from the program code that we have applied standard finite elements (identical order for elements) for discretizing velocity and pressure.

Let us now look at the result of the error estimate to the method described above. Assume that $\mathbf{u}$ and $\mathbf{p}$ are the analytical solutions of the problem, and let $u_{\Delta t} = u_{1,\dots}, u_n$ be some sequence of functions. From [8] *the Semi Implicit*

Projection Method satisfies the following error estimate;

$$||\mathbf{u}_{\Delta t} - v_{\Delta t}||_{L^\infty(L^2(\Omega)^d)} + ||\mathbf{u}_{\Delta t} - v^*_{\Delta t}||_{L^\infty(L^2(\Omega)^d)} \leq C(\mathbf{u}, \mathbf{p}, T)\, \Delta t \tag{4.25}$$

$$||\mathbf{p}_{\Delta t} - p_{\Delta t}||_{L^\infty(L^2(\Omega))} + ||\mathbf{u}_{\Delta t} - v^*_{\Delta t}||_{L^\infty(H^1(\Omega)^d)} \leq C(\mathbf{u}, \mathbf{p}, T)\, \Delta t^{\frac{1}{2}} \tag{4.26}$$

where v^* is the tentative velocity and v satisfies the continuity equation, from the algorithm above.

$$||\cdot||_{L^2(\Omega)} = \big(\Delta t \sum_{k=0}^{n} \big|\big|\, \cdot^k \big|\big|^2_\Omega\big)^{\frac{1}{2}} \text{ and } ||\cdot||_{L^\infty(\Omega)} = max\big(\, \big|\big|\, \cdot^k \big|\big|_\Omega\big)$$

Note that the estimates are time-dependent. Methods like the one shown above are time- marching techniques. In basic this means that the approximate solutions converges for each time-step towards a better calculated solution.

4.3 Mixed methods and The Stokes Problem

Consider a simplification of the Navier-Stokes equations, namely the Stokes problem. Much of the mathematical theory and understanding of numerical solutions of the Navier-Stokes equations have been developed for this problem,

$$\begin{aligned} -\Delta u - \nabla p &= f \text{ in } \Omega & (4.27)\\ \nabla \cdot u &= 0 \text{ in } \Omega. & (4.28) \end{aligned}$$

The Stokes Equations (4.27) and (4.28), is a saddle-point problem requiring special care in order to satisfy a stable discretization in space. The idea is to employ *Mixed Finite Elements*. The term mixed elements refers to the selection of different finite elements for the various unknown. By selecting a particular spatial discretization for the Stokes problem and put it into the block system (4.18), we can ensure that the linear system is invertible.

The question is obviously. What conditions are required to determine the unknown **u** and p uniquely? From [10] we have the following properties on the matrices N and Q in (4.18),

- N is positive definite
- $\text{Ker}(Q) = \{0\} \Leftrightarrow \sup_{v_h \in V_h} \int_\Omega p_h \nabla \cdot v_h > 0$

gives a solvable system. But we also need a stability criteria. Stability in the sense that the *Schur complement* $Q^T N^{-1} Q$, does not get singular as $h \to 0$. Here h represents the scale of the discretization. To obtain stable algebraic equations

for *the Stokes problem*, the finite element spaces V_h and Q_h should be chosen such that they satisfy the *Brezzi conditions*,

$$\inf_{p_h \in Q_h} \sup_{v_h \in V_h} \frac{\int_\Omega p_h \nabla \cdot v_h}{||v_h||_1 \, ||p_h||_0} \geq \beta > 0 \tag{4.29}$$

and,

$$a(v_h, v_h) \geq D \, ||v_h||_1^2 \quad \forall v_h \in V_h. \tag{4.30}$$

Here β and D are independent of the grid parameters, and $a(v_h, v_h) = (\nabla v_h, \nabla v_h)$. $V_h \subset H_0^1$, $Q_h \subset L_0^2$ and v_h, p_h are discrete solutions. $||\cdot||_1 = ||\cdot||_{H^1}$ and $||\cdot||_0 = ||\cdot||_{L^2}$.

Provided that the *Brezzi conditions* are fulfilled, the following error estimate is satisfied [10]:

$$||v_h - v||_1 + ||p_h - p||_0 \leq C(h^k \, ||v||_{k+1} + h^{l+1} \, ||p||_{l+1}). \tag{4.31}$$

Here the constant C is independent of the spatial mesh parameter and the velocity field. The exact solutions are v, p and v_h,p_h are the the approximate solutions. The order of piecewise continuous polynomials are given by k and l respectively for velocity and pressure. Convergence in L^2 norm is one degree higher in v than p, it follows from (4.31) that $k = l + 1$ is optimal.

4.3.1 Mixed Formulation of Navier-Stokes Equations

The properties outlined in the previous section for the Stokes problem are advantageous now in the study of the more complicated Navier-Stokes equations. However, we must show caution. The discretization of the convection term in the Navier-Stokes equations can lead to unstable equations. By taking advantage of stabilization techniques like *upwinding* or *artificial diffusion* we can get past this problem.

Suppose now that for any t, the Brezzi conditions (4.29) and (4.30) is satisfied. Then the system of differential algebraic equations (4.16), (4.17) will have a solvable and stable solution.

For the mixed finite element algorithm we have selected the scheme given in Chapter 6.2.2 and Equation (6.11) to perform simulations of incompressible viscous fluid flow. A closer look at the implementation of the mixed element algorithm, see Appendix A.3.

4.3.2 Results and Error Estimate Mixed Method

We will apply mixed finite elements to the simulations of the incompressible Navier-Stokes equations. This approach allows us to implement with correct boundary conditions that are compatible to the original system (4.1) and (4.2), unlike what we could with the projection algorithm (Chapter 4.2.1). The use of the mixed formulation with a combination of elements for velocity and pressure that satisfies the Babuska-Brezzi condition, ensures a stable and solvable system of algebraic equations.

When solving problems like (4.16) and (4.17) called saddle-point problems or maximum/minimum problems, we require efficient solvers. Since this are complex coupled systems. The use of preconditioning (see Chapter 4.5) and the numerical software tool FEniCS (see Chapter 5), simplifies the implementation considerably. Without this tools would the implementation of the mixed algorithm become dramatically more difficult.

Let us now look at the result of the error estimate for the Navier-Stokes quations by the mixed method. Assume that v and p are the analytical solutions of the problem. The approximate solutions are given by v_h and p_h.

$$||v_h - v||_1 + ||p_h - p||_0 \leq C_v(h^k \, ||v||_{k+1} + h^{l+1} \, ||p||_{l+1}). \tag{4.32}$$

Note that the constant C_v is now dependent of the velocity field. This is different from the error estimate we saw in (4.31), otherwise everything is the same. Here h describes the grid parameter.

4.4 Iterativ Methods

Consider now a linear system arising from a finite element discretization of a partial differential equation. Where the system contains a large number of unknown, and the matrix A_h has a structure that reflects the PDE operator. In this context direct methods will be too time-consuming, and in some cases impossible. Instead of solving the linear system directly, we now consider *iterative solution methods*. This is based on computing a sequence of approximations $\{u^n\}_{n=1}^{\infty}$, such that $\lim_{n\to\infty} u^n = u$. Where u is the solution of $Au = b$.

4.4.1 The Richardson Iteration

Our goal is to solve the linear system,

$$Au = b. \tag{4.33}$$

The simplest way to solve (4.33) iteratively, is to reformulate it as a fixed point iteration,

$$u = u - Au + b = (I - A)u + b.$$

This define *the Richardson iteration*,

$$u^n = u^{n-1} - (Au^{n-1} - b). \tag{4.34}$$

We now define the *error*,

$$e^n = u^n - u \tag{4.35}$$

and the *residual*

$$r^n = b - Au^n. \tag{4.36}$$

A relation between the current error and the current residual is given by,

$$Ae^n = r^n.$$

Note that the definition of the residual is independent of the actual solution u.

An important question is whether the Richardson iteration converges to the solution? The iteration is convergent if the error decreases at each step. By inserting (4.34) into (4.35) and use that $b = Au$,

$$e^n = u^{n-1} - (Au^{n-1} - Au) - u = (I - A)(u^{n-1} - u) = (I - A)e^{n-1}. \tag{4.37}$$

This means that we require,

$$||e^n|| \leq ||I - A|| \left|\left|e^{n-1}\right|\right|. \tag{4.38}$$

The Richardson iteration is convergent if the *convergence rate* $||I - A|| < 1$.

The next property we need to consider is a stopping criterion for the method. In many cases we want to stop the iteration when we are arbitrary close to the solution, to avoid infinite iteration. The error equation (4.35) will generally not help us in an applied problem, since we of course not know the actual solution u. However, the residual defined in (4.36), is independent of u. Most iterative methods terminate as soon as the residual is sufficiently small. One frequently used stopping criterion that we also apply in this thesis is,

$$\frac{||r^t||}{||r^0||} < \epsilon. \tag{4.39}$$

Here ϵ is a prescribed number. The iteration will stop when we reach the predicted accuracy measured by a discrete norm of the residual, or exceeded the maximum number of iterations.

The effectiveness of the solution strategy is dependent on the development of iterative solvers. It turns out that Richardson iteration has poor performance for the discretization of PDEs. The *spectrum* of these operators (matrices) are in most cases unbounded, where the spectrum can be seen as the operators set of eigenvalues. The cure for this problem will be introduced in the next section.

4.5 Preconditioning

The *convergence rate* of iterative methods for discretized PDEs, are dependent on the *spectrum* of the operator A_h. If the spectrum of A_h is unbounded, it follows that the problem is ill-conditioned, and *the condition number* of A_h tend to infinity as the grid spacing h tend to zero. This weakens the convergence rate of the iterative methods when the grid is refined. In our case with discretization of partial differential equations *the condition number* is defined as,

$$\kappa(A_h) = \|A_h\| \cdot \|A_h^{-1}\| = \left| \frac{\lambda_{max}(A_h)}{\lambda_{min}(A_h)} \right|. \tag{4.40}$$

The ratio between the largest and smallest eigenvalue of the operator/matrix. Here λ represents an eigenvalue of the operator.

In order to improve the convergence rate of iterative methods, a linear system is *preconditioned* by multiplying both sides of (4.33) by a nonsingular matrix B, that approximates A^{-1} such that,

$$BAu = Bb. \tag{4.41}$$

By a properly chosen B, the spectrum and the condition number of the operator $\kappa(BA)$ is bounded.

The preconditioned Richardson iteration has the following form,

$$u^n = u^{n-1} - B(Au^{n-1} - b). \tag{4.42}$$

If the norm $||I - BA||$ is small, it speeds up the *convergence*, and the iterative method is improved.

The four main elementary iterative methods Jacobi method, Gauss-Seidel method, successive overrelaxation method (SOR), and symmetric successive overrelaxation method (SSOR) can fit into the framework of preconditioned Richardson iteration.

One example is the Jacobi iteration. Let $B = D^{-1}$, where $D = diag(A)$ from (4.42), and we can evaluate the convergence of the method $||I - D^{-1}A|| < 1$. For a more detailed derivation of Jacobi and the other classical methods see [14]. However, they are building blocks in the more advanced multigrid algorithm (see Chapter 4.6).

4.5.1 Abstract Motivation

Let us look a little more abstract of the process of designing a preconditioner. The theoretical basis of this section and the previous section, is from [16],[15] and [4].

Consider the following abstract formulation, where V is a Hilbert space.

Find $u \in V$ such that,

$$Au = f. \tag{4.43}$$

Here the right hand side $f \in V^*$, and V^* is the dual space of V. The unknown $u \in V$.

Assume that $A : V \to V^*$ is a bounded invertible linear differential operator such that,

$$A \in L(V, V^*) \text{ and } A^{-1} \in L(V^*, V).$$

Observe that the operator A maps functions in V out of its own space. This creates difficulties such as, unbounded spectrum, and in the discrete case the iterative methods will not converge properly.

The key tool to this unbounded problems is to introduce a *preconditioner* B. The preconditioner is an operator $B : V^* \to V$ such that,

$$B \in L(V^*, V) \text{ and } B^{-1} \in L(V, V^*).$$

Then,

$$BA : V \xrightarrow{\text{A}} V^* \xrightarrow{\text{B}} V$$

is a mapping V to itself. Therefor the preconditioned system,

$$BAx = Bf, \quad \text{with } BA \in L(V, V) \tag{4.44}$$

can be solved by a iterativ method. The system will have a convergence rate bounded by the condition number where the condition number is defined,

$$\kappa(BA) = \|BA\|_{L(V,V)} \|BA^{-1}\|_{L(V,V)} \leq D. \tag{4.45}$$

Preconditioned Stokes Problem

From earlier we know that the Stokes problem (4.27) and (4.28), must satisfy the Brezzi conditions (4.29) and (4.30), for a stable discretization. We can write the coefficient operator for the Stokes problem as,

$$A_h = \begin{bmatrix} -\Delta & -grad \\ div & 0 \end{bmatrix} \tag{4.46}$$

where $A_h : (H_0^1(\Omega)^d \times L_0^2(\Omega)) \longrightarrow (H^{-1}(\Omega)^d \times L_0^2(\Omega))$.

The following preconditioner B_h, is known to be a good choice [16],

$$B_h = \begin{bmatrix} (-\Delta)^{-1} & 0 \\ 0 & I \end{bmatrix} \tag{4.47}$$

where $B_h : (H^{-1}(\Omega)^d \times L_0^2(\Omega)) \longrightarrow (H_0^1(\Omega)^d \times L_0^2(\Omega))$.

4.6 Algebraic Multigrid Method

In a search for an order optimal method for large systems of linear equations, we will briefly present a *multigrid method*. This since we have applied such a method in parts of the implementation, see Appendix A.1 and Chapter 6.
Multigrid methods are among the fastest numerical algorithms for the solution of large sparce systems of linear equations [3]. In this type of algorithms a given problem is solved by integrating different levels of resolution into the solution process. During this process the solution contributions at the different levels are combined appropriately together to form the final solution. Relaxed variants of the elementary iterative methods are used at the different levels.

From theory [15], the downside to all classical iterative solvers when applied to large sparce linear systems are that they cannot efficiently reduce the low frequency (smooth) errors. However they often succeed in eliminating the high frequency errors. The multigrid algorithms take advantage of this property.

After a few relaxations all high frequency modes of the error is eliminated and the smooth modes dominate the remaining error. Such that

$$Ae^n = r^n$$

where e^n and r^n are given in (4.35) and (4.36) respectively. Then

$$u = u^n + e^n.$$

Chapter 5

Software tools

This chapter provides a presentation of some software programs applied in the implementations. Because of the time it took to develop an understanding in order to take advantage of these various software applications, it felt appropriate to devote a separate chapter for this topic. We will throughout the chapter illustrate the applied work by some screenshots. For a more complete and detailed description, we refer to the individual software-applications website.

5.1 The Vascular Modeling Toolkit (VMTK)

Modeling of flow in blood vessels is today a major research field, and a lot of effort is put into computer simulations of such kinds of problems. In order to create favorable simulations, there is nothing better than working with real patient data. In our case this is data from the scanning of the human head, and we want to recreate a 3D volumetric image of this data-set. It is in this context we want to apply the VMTK software tool.

The Vascular Modeling Toolkit is a collection of libraries and tools for 3D reconstruction, geometric analysis, mesh generation and surface data analysis for image-based modeling of blood vessels [1].

The construction of 3D images are in itself a separate field. It is very complicated to reproduce a precise geometry of such a complex surface. However, in this thesis, we assume that we have found a good enough re-creation of the scan, and does the work from this point of view. Since the main focus is to perform simulations on a given blood vessel, we ignore the fact concerning whether a blood vessel is rendered 100 percent correct. We convert the image into a 3D volume grid by a built-in mesh generator, and then apply the finite element method.

Below we have described the application of VMTK, and the process with some figures. Find a complete reference to publications at David Steinman's and Luca Antiga's homepages, see also *http://www.vmtk.org/*.

- In the first step VMTK reads the image, and displays it on the screen, see Figure 5.1. From this we can find the so called volume of interest (VOI), and enlarge this VOI for further analysis. Observe from the Figure that the scan is taken from the top of the head and down, and we can observe the "*eyes*" in the upper part of the Figure. It is possible to select which level curve of the skull we want to study further. This is done by scrolling the mouse on the computer. At this level in the process all the work is done on a *.vti* image file (VTK format file).

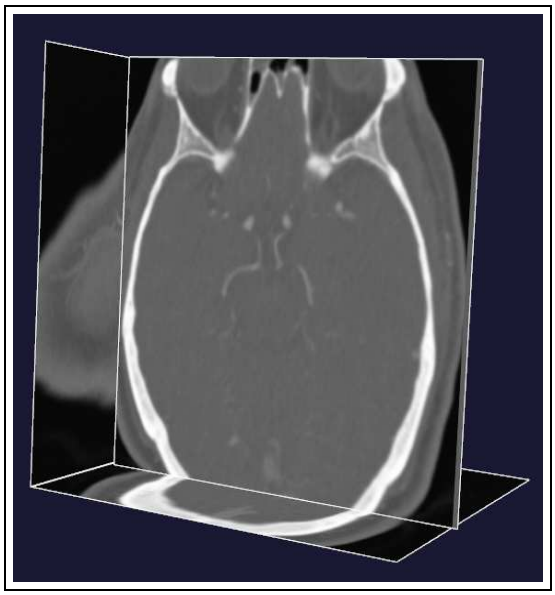


Figure 5.1: A scan of a human head, given from a medical data-set. Observe the blood vessels in the middle of a selected level set.

- The next step is very interesting. Here we select the segments of the scan that we will analysis further. By selecting a vascular segment between two points, we get out the info we want. Observe from Figure 5.2, the two red dots where we have selected a segment and ignored the side branches. Repeating this operation for different segments, and then merging everything into a single surface.

Figure 5.2: In the process of building up a 3D surface. Here we have merged together some selected segments.

- At a point when we are pleased with our 3D surface, we wish to perform some additional operations. This involves clipping and smoothing of the generated 3D surface model. Figure 5.3 shows an example of how such a model may look.

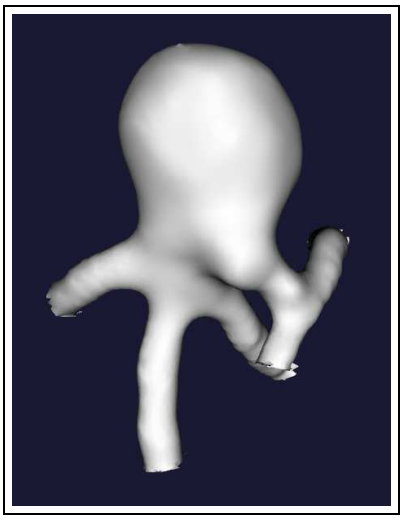


Figure 5.3: A 3D surface model.

- The final step will now be to generate a mesh out of the 3D surface model in Figure 5.3. The 3D surface will then be converted into a Dolfin

accepted .xml format. See Figure 5.4 for the generated mesh version of the final surface.

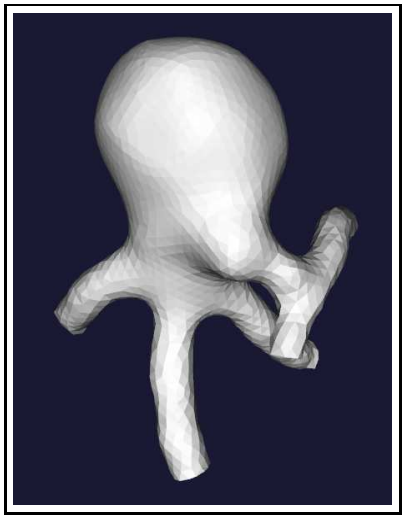


Figure 5.4: The generated mesh version of the final 3D surface model.

5.2 FEniCS Project

FEniCS is a software tool for solving partial differential equations, and is a collaboration project between a number of universities including Simula Research Laboratory. The programming language applied in this thesis is Python, and Dolfin is the Python interface of FEniCS. Where Dolfin is a problem solving environment for differential equations.

Automation of the finite element method and finite element simulations are strengths of the FEniCS software tool. It is designed such that the transformation from an abstract mathematical version of a PDE problem, to a discrete weak form of the problem shall be uncomplicated to implement.

FEniCS has plenty of built-in features which specifies the implementation on a readily understood manner. The creation of suitable function spaces and finite elements are two examples. Further, FEniCS have features that trivial creates mesh (2D and 3D), defining the corresponding boundary conditions (Dirichlet, Neumann etc) and assembling of bilinear form a, and linear form L of a finite element formulation. All this to achieve a short and precise computer code. We would refer to [12] for a good introduction to FEniCS. In the Appendix you can find computer code where we have applied this tool.

5.3 MESHBUILDER

Meshbuilder is a software tool applied to set boundary conditions on generated surface mesh. The program has a geometric interface, where we can loading .xml files into it. Dolfin, the interface of FEniCS has support for such .xml files, and VMTK generates surface mesh. We will use Meshbuilder to set the various boundary conditions we want for our problems. In Figure 5.5 we can see the use of Meshbuilder.

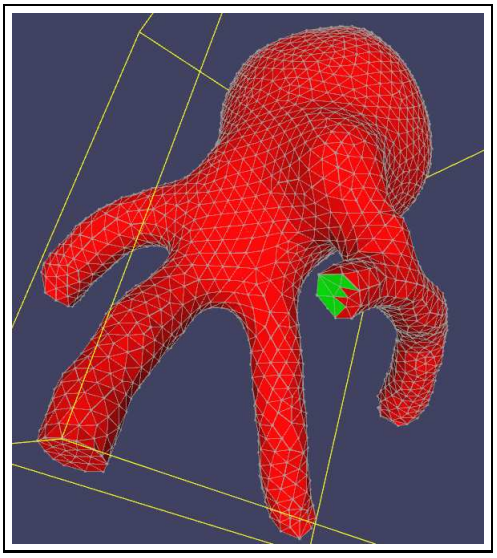


Figure 5.5: Defines the boundary conditions on the generated mesh from VMTK. Note the green color field on the surface, indicating that there should be a certain boundary condition on the outflow.

5.4 GMSH

Gmsh is a 3D finite element grid generator which we have used to create simple test models for the development of the implementation. The software is easy to learn and is design to provide a fast and user-friendly meshing tool with an advanced visualization capabilities. Gmsh has four main applications geometry, mesh, solver and post-processing. In the thesis we have made use of the two first parts, the geometric representation of the model and then generated a mesh model. The specification of input to create a simple model is done either interactively using the graphical user interface or using Gmsh's own scripting language.

Chapter 6

Simulations and results

This chapter concerns the simulations and results of the implementations. In the first part the objective is to find a fast and well adapted solution algorithm for two Poisson examples. The inclusion of preconditioners and alternative solvers, is a necessity to accelerate the computation time on selected algorithms. In the next section we have solved *Poisson problems* with different types of preconditioners and solvers.

Next, we move forward to solve *the time dependent Stokes problem* and *the Navier-Stokes problem*, carried out by manufactured solutions explained in Chapter 3.2. We perform simulations by applying this method in $2D$ and $3D$. The aim is to verify the implementations. The focus will be to identify the convergence and the CPU time of the solutions when we change the grid parameter h.

The main purpose is not only to solve the problems, but also to evaluate and compare the performance of the numerical methods described in Chapter 4. In particular, investigate the *CPU (central processing unit) time* and *accuracy*, and discuss the advantages and disadvantages. The last part of this chapter deals with simulation of flow through an aneurysm.

6.1 Poisson Problems

The aim of this section is solving the Poisson 2D equations (6.1) and (6.3), with different solution algorithms, and varying number of unknown. Then we compare the CPU-time required to solve the two problems. The equations below are interesting to study since they are important building blocks in a number of solution algorithms for the Navier Stokes equations.

Problem 1:

$$-\Delta u = f\,, \text{ in } \Omega = (0,1)\,\mathrm{x}\,(0,1) \tag{6.1}$$

$$u0 = 0\,, \text{ on } \partial\Omega \tag{6.2}$$

Problem 2:

$$u - \epsilon \Delta u \quad = \quad f \text{ , in } \Omega = (0,1)\,\mathrm{x}\,(0,1) \text{ and } \epsilon > 0. \tag{6.3}$$

The corresponding variational problems are:

Find $u \in H_0^1$ such that,

$$\int_\Omega \nabla u \cdot \nabla v \, dx \quad = \quad \int_\Omega f \, v \, dx, \quad \forall \; v \in H_0^1. \tag{6.4}$$

Find $u \in H_0^1$ such that,

$$\int_\Omega u \cdot v \, dx + \int_\Omega \epsilon \, \nabla u \cdot \nabla v \, dx \quad = \quad \int_\Omega f \, v \, dx, \quad \forall \; v \in H_0^1. \tag{6.5}$$

Here we have chosen $f = sin(x)$, and we are experimenting with different values of ϵ in (6.5)

Since the assignment deals with both direct and iterative algorithms, we require a stopping criterion for the iterative methods. We defines a tolerance of 10^{-5} on the relative residual, as a stopping criterion.

6.1.1 Implementation Packages

The algorithms are implemented in Python, and for the iterative solvers we have applied the linear algebra library *Epetra* and the preconditioner library *AztecOO*. These libraries can easily be imported independently into a Python script from PyTrilinos (see http://trilinos.sandia.gov/packages/pytrilinos/index.html). We have additionally imported the *ML* package for designing a multilevel preconditioner.

In the program code part below we have in advance assembled a bilinear form $a(\cdot,\cdot)$, and a linear form $L(\cdot)$, from a Poisson variational problem. From this variational problem we have created a matrix A and a vector b. In the code, it is shown how we use the matrix A to create a multilevel preconditioner, and then solves the system with a conjugate gradient solver.

```
from PyTrilinos import Epetra, AztecOO, ML
from dolfin import *

# apply the linear algebra library Epetra
parameters["linear_algebra_backend"] = "Epetra"
```

```
A_epetra = down_cast(A).mat()
b_epetra = down_cast(b).vec()
x_epetra = down_cast(U.vector()).vec()

# Sets up the parameters for a multilevel preconditioner
MLList = {"max levels"        : 30,
          "output"            : 1,
          "smoother: type"    : "ML symmetric Gauss-Seidel",
          "aggregation: type" : "Uncoupled",
          "ML validate parameter list" : False
}

# Create the preconditioner
ml_prec = ML.MultiLevelPreconditioner(A_epetra, False)
ml_prec.SetParameterList(MLList)
ml_prec.ComputePreconditioner()

# Create solver and solve the system
Solver = AztecOO.AztecOO(A_epetra, x_epetra, b_epetra)
Solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.AZ_cg)
Solver.SetPrecOperator(ml_prec)
Solver.Iterate(1550, 1e-5)
```

For the complete implementation, see Appendix A.1.

6.1.2 Simulations

The calculations are done with respect to Lagrange P_1 finite elements. The numbers in the tables indicate the CPU-time, and number of iterations to solve the problem respectively. The combination of the choice of solvers and preconditioners is given in the top row in each table.

Table for *Problem 1*:

The computer calculations are performed with a MacBook Intel Core 2 Duo, with a 2.4 GHz processor.

unknown	cg/amg /it	amg/it	cg/none/it	cg/ilu/it	cg/jacobi/it	LU-fact(UMFP.)
60^2	0.0106 /5	0.0254 /5	0.0177 /129	0.02632 /29	0.02507 /129	0.03567
120^2	0.0380 /5	0.0327 /4	0.1284 /262	0.09113 /55	0.14114 /262	0.17109
240^2	0.2078 /6	0.1983 /6	1.3981 /532	0.8140 /107	1.67193 /532	0.9047
480^2	0.8276 /6	0.7473 /5	13.7544 /1080	6.6827 /196	14.5691 /1080	5.31924
960^2	2.9581 /5	3.1004 /5	1550+ it	51.2733 /395	1550+ it	46.1354

Tables for *Problem 2*:

$\epsilon = 1.0$

unknown	cg/amg /it	amg/it	cg/none/it	cg/ilu/it	cg/jacobi/it	LU-fac(UMFP.)
60^2	0.03824 /10	0.03815 /10	0.03177 /233	0.04161 /49	0.03389 /194	0.06716
120^2	0.07125 /11	0.07041 /10	0.22227 /427	0.14010 /89	0.20961 /373	0.16738
240^2	0.39955 /14	0.39999 /13	2.18913 /836	1.18226 /170	1.95466 /704	0.90075
480^2	1.86998 /15	1.80611 /13	1550+ it	10.0509 /323	18.4078 /1335	5.29668
960^2	9.67421 /19	9.57739 /17	1550+ it	78.6696 /629	1550+ it	48.2385

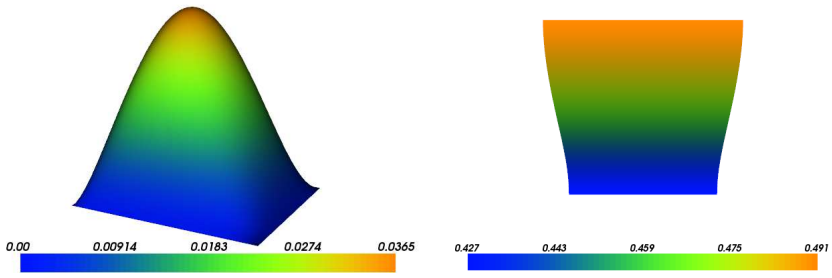
$\epsilon = 0.001$

unknown	cg/amg /it	amg/it	cg/none/it	cg/ilu/it	cg/jacobi/it	LU-fac(UMFP.)
60^2	0.00626 /3	0.0078 /3	0.00648 /32	0.01063 /6	0.0050 /24	0.0425
120^2	0.02987 /4	0.0333 /4	0.0313 /60	0.04121 /11	0.0317 /49	0.1781
240^2	0.16572 /5	0.17678 /5	0.3095 /117	0.2761 /21	0.3037 /99	0.9092
480^2	0.7008 /5	0.7345/5	3.027 /229	2.0069 /41	2.8648 /204	5.555
960^2	3.5171 /6	3.5584 /6	24.177 /454	14.8800 /81	24.565 /419	43.458

$\epsilon = 0.0001$

unknown	cg/amg /it	amg/it	cg/none/it	cg/ilu/it	cg/jacobi/it	LU-fac(UMFP.)
60^2	0.0036 /1	0.0037 /1	0.0025 /13	0.0073 /2	0.00264 /7	0.0680
120^2	0.0189 /2	0.02085 /2	0.0124 /21	0.0332 /4	0.01238 /13	0.1725
240^2	0.1073 /3	0.12837 /3	0.1038 /38	0.2087 /7	0.08835 /25	0.8814
480^2	0.4688 /3	0.4926 /3	0.9773 /72	1.2771 /13	0.75638 /51	5.3402
960^2	2.5392 /4	2.5559 /4	7.5495 /139	8.0635 /24	6.462 /104	47.0009

The solution of *Problem 1* is to the left, and the solution of *Problem 2* is to the right.



(a) Solution problem 1. (b) Solution problem 2.

6.1.3 Results

The idea was to find out if any preconditioners accelerated the convergence of our two Poisson problems (6.1) and (6.3). From the simulations and the results in the tables, we see clearly that the algebraic multigrid (amg) preconditioner provides by far the fastest convergence. The combination of a algebraic multigrid preconditioner, and a conjugate gradient solver achieved a reduction of the overall CPU-time. The CPU-time was much lower compared with the other elementary

iterative methods and the direct solver. We observe this tendency very clearly, when we increase the number of unknown.

Note that the CPU-time is approximately linear as we increase the number of unknown for the advanced cg/amg algorithm. We have an order optimal method $O(n)$. This is a positive observation in order to solve larger and more complex algebraic systems later. The other simpler solvers do not share this linear property. Figure 6.1 demonstrates the linear evolution of cg/amg method versus cg/ilu method, based on the results from the table for *Problem 1*. We can also observe this nonlinear behavior in the other elementary methods (Jacobi, direct method, etc).

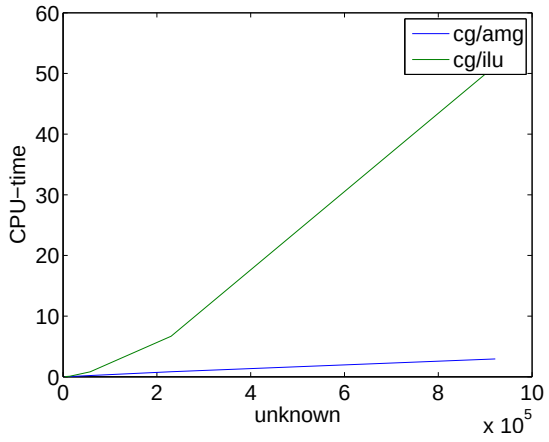


Figure 6.1: Comparison of CPU-time for cg/amg and cg/ilu for increased number of unknown. Note the linearity in cg/amg method.

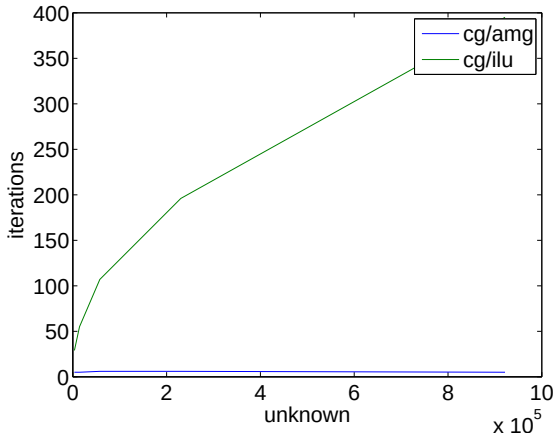


Figure 6.2: Comparison the number of iterations in cg/amg and cg/ilu. Note constant number of iterations in cg/amg method.

The number of iterations remains constant in cg/amg method. This is not the case with the other methods. Figure 6.2 shows the difference between cg/amg method and cg/ilu method.

For a detailed overview of the implementation for the problems in this section, see Appendix A.1.

6.2 Manufactured solutions

The artificial solutions in 2D and 3D were chosen respectively in Chapter 3.2 as,

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} sin(y) \\ sin(x) \end{bmatrix} \tag{6.6}$$

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} sin(y) \\ sin(x) \\ 0 \end{bmatrix} \tag{6.7}$$

and,

$$\nabla p = -p_{grad}. \tag{6.8}$$

Solving the following system,

$$\begin{aligned} \mathbf{u}_t + k(\mathbf{u}\cdot\nabla\mathbf{u}) - \nu\Delta\mathbf{u} - \nabla p &= \mathbf{f} \\ \nabla\cdot\mathbf{u} &= 0. \end{aligned} \tag{6.9}$$

The parameter $k = 0$ or 1, in front of the nonlinear term acts like a switch, depending on whether we solve *the time dependent Stokes Problem* or *the Navier Stokes Problem*. The source term $\mathbf{f}$ is calculated in Chapter 3.2, and viscosity is described by ν.

6.2.1 The time dependent Stokes Problem

The solution scheme for the projection algorithm is described in detail in Chapter 4.2.1. For the mixed element method we choose the following scheme,

$$\begin{aligned} u - \epsilon\Delta u - \nabla p &= f \\ \nabla\cdot u &= 0 \\ u &= 0 \end{aligned} \tag{6.10}$$

here ϵ includes time and viscosity.

Results from simulations of *the projection method* and *mixed element method*, are given in the tables below. $P_1 - P_1$ and $P_2 - P_1$ describes the combination of choice of Lagrange finite elements. $P_2 - P_1$ (Taylor Hood elements) describes second order Lagrange elements (CG2) in the velocity field, and first-order Lagrange elements in pressure (CG1). The boundary conditions are given as Dirichlet boundary condition, and the initial condition $u0 = 0$. In the case of stationary solutions, it takes time to achieve the correct solution. Numerical experiments indicate that the stationary solution is obtained at $T = 1.0$, when the time discretization $\Delta t < 0.2$. The viscosity parameter is set to $\nu = 1.0$. Implementation of the numerical methods can be found in Appendix A.2, A.3.

Projection Method 2D problem

$P_1 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	4	8	16	32	64
0.2	4.61×10^{-3} / 0.70	4.13×10^{-3} / 0.72	4.02×10^{-3} / 0.98	3.99×10^{-3} / 1.59	3.99×10^{-3} / 4.07
0.1	1.41×10^{-3} / 1.35	1.10×10^{-3} / 1.44	1.02×10^{-3} / 1.78	9.97×10^{-4} / 2.86	9.93×10^{-4} / 8.67

$P_2 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	4	8	16	32	64
0.2	4.72×10^{-3} / 0.84	4.18×10^{-3} / 0.98	4.03×10^{-3} / 1.61	3.99×10^{-3} / 4.60	3.98×10^{-3} / 20.60
0.1	1.32×10^{-3} / 1.45	1.08×10^{-3} / 2.00	1.01×10^{-3} / 3.00	9.95×10^{-4} / 9.82	9.92×10^{-4} / 45.15

Observations: We achieve convergence when the grid is refined. $P_1 - P_1$ elements and $P_2 - P_1$ elements give almost identical error estimates, but the CPU-time is faster for $P_1 - P_1$ elements.

Projection Method 3D problem

$P_1 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	2	4	8
0.2	5.37×10^{-3} / 0.79	5.60×10^{-3} / 1.25	5.16×10^{-3} / 6.28
0.1	2.35×10^{-3} / 1.44	1.60×10^{-3} / 2.72	1.47×10^{-3} / 13.63

$P_2 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	2	4	8
0.2	6.65×10^{-3} / 1.79	5.59×10^{-3} / 10.78	5.10×10^{-3} / 169.8
0.1	1.91×10^{-3} / 3.67	1.67×10^{-3} / 23.38	1.46×10^{-3} / 360.8

Observations: Convergence when the grid is refined. We get the same tendency as in the 2D case. Note that the CPU time is excessively longer for $P_2 - P_1$ elements.

Mixed element method 2D problem

$P_2 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	8	16	32	64
0.2	6.80×10^{-8} / 0.52	5.98×10^{-9} / 1.46	4.72×10^{-9} / 6.40	4.75×10^{-9} / 41.50
0.1	6.85×10^{-8} / 1.42	4.42×10^{-9} / 3.12	2.80×10^{-10} / 14.0	1.76×10^{-11} / 91.0
0.01	6.85×10^{-8} / 9.40	4.42×10^{-9} / 28.02	2.80×10^{-10} / 127.60	1.76×10^{-11} / 858.0

Mixed element method 3D problem

$P_2 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	2	4	8
0.2	1.19×10^{-5} / 2.27	1.29×10^{-6} / 14.77	9.54×10^{-8} / 283.15
0.1	1.19×10^{-5} / 4.06	1.29×10^{-6} / 32.32	9.58×10^{-8} / 623.89
0.01	1.19×10^{-5} / 37.04	1.29×10^{-6} / 298.20	9.58×10^{-8} / 5 801

Observation: We achieve convergence when the grid is refined for both problems.

Results:

The purpose was to verify the implementation of the time dependent Stokes problem. The tables summarized indicates that we have obtained this goal for both methods. They converge with respect to the L2 errornorm.

If we go into more detail and compares the two methods, we observe that the projection algorithm solves the problems faster for identical parameter values. However, the accuracy is a lot better in the mixed element algorithm compared with the projection algorithm. In fact, the accuracy is so much better in the mixed algorithm that we can solve the problem on a coarse grid, and still achieve a better error estimate at a faster CPU time. We demonstrate this on the basis of the tables in the 2D example. For the parameters, $N = 8$ and $\Delta t = 0.1$ the mixed method has an error estimate of about $\sim 10^{-8}$, computed at a CPU-time= 1.42. Compared to the projection method with, $N = 64$ and $\Delta t = 0.1$ need a CPU- time= 8.67 to reach an error estimate of approximately $\sim 10^{-3}$. The same tendency exists also in the 3D case.

For the projection algorithm has $P_1 - P_1$ and $P_2 - P_1$ elements almost the same error estimate. However, in general the CPU-time for $P_1 - P_1$ elements is faster, and this is demonstrated even more clearly for the 3D problem.

6.2.2 The Navier Stokes Problem

For the mixed element method we select the following scheme for the Navier-Stokes problem,

$$\begin{aligned} u + dt\,(u_0 \cdot \nabla u) - \epsilon \Delta u - \nabla p &= f \\ \nabla \cdot u &= 0 \\ u &= 0 \end{aligned} \tag{6.11}$$

here ϵ includes time and viscosity, and u_0 is the solution of the previous time step. All parameters are as in the previous section, but we have gained an extra convection term.

Simulations of *the projection method* and *mixed element method* are given below.

Projection Method 2D problem

$P_1 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	4	8	16	32	64
0.2	4.74×10^{-3} / 0.65	4.16×10^{-3} / 0.70	4.01×10^{-3} / 0.93	3.98×10^{-3} / 1.65	3.97×10^{-3} / 4.17
0.1	1.23×10^{-3} / 1.27	9.40×10^{-4} / 1.38	8.64×10^{-4} / 1.87	8.47×10^{-4} / 2.94	8.44×10^{-4} / 9.00

$P_2 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	4	8	16	32	64
0.2	4.73×10^{-3} / 0.74	4.17×10^{-3} / 0.98	4.02×10^{-3} / 1.53	3.98×10^{-3} / 4.87	3.97×10^{-3} / 21.70
0.1	1.18×10^{-3} / 1.42	9.35×10^{-4} / 1.97	8.63×10^{-4} / 3.10	8.47×10^{-4} / 10.60	8.44×10^{-4} / 47.60

Observation: Convergence is achieved. CPU-time is faster for $P_1 - P_1$ elements.

Projection Method 3D problem

$P_1 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	2	4	8
0.2	5.43×10^{-3} / 0.82	5.71×10^{-3} / 1.43	5.26×10^{-3} / 7.67
0.1	2.41×10^{-3} / 1.60	1.49×10^{-3} / 2.84	1.33×10^{-3} / 16.68

$P_2 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	2	4	8
0.2	6.80×10^{-3} / 2.22	5.69×10^{-3} / 14.59	5.20×10^{-3} / 195.3
0.1	1.69×10^{-3} / 4.70	1.53×10^{-3} / 31.70	1.31×10^{-3} / 428.9

Observation: Convergence when the grid is refined. We get the same tendency as in the 2D case. Note that the CPU time is excessively larger in $P_2 - P_1$ elements.

Mixed element method 2D problem

$P_2 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	8	16	32	64
0.2	5.43×10^{-7} / 0.59	3.56×10^{-8} / 1.65	4.81×10^{-9} / 7.20	3.86×10^{-9} / 44.60
0.1	5.42×10^{-7} / 1.17	3.45×10^{-8} / 3.50	2.17×10^{-9} / 15.70	1.36×10^{-10} / 98.17
0.01	5.42×10^{-7} / 9.40	3.45×10^{-8} / 31.0	2.17×10^{-9} / 142.0	1.36×10^{-10} / 893.0

Observation: We achieve convergence when the grid is refined.

Mixed element method 3D problem

$P_2 - P_1$ Taylor Hood

The numbers in the table indicate the error and CPU time respectively.

Δt \ N	2	4	8
0.2	8.03×10^{-5} / 2.48	7.39×10^{-6} / 19.52	5.20×10^{-7} / 320.88
0.1	8.03×10^{-5} / 5.33	7.39×10^{-6} / 43.10	5.19×10^{-7} / 704.0
0.01	8.03×10^{-5} / 47.23	7.38×10^{-6} / 390.13	5.19×10^{-7} / 6 400

Observation: Convergence when the grid is refined.

Results:

The implementation of the Navier-Stokes problem is verified for both methods. We achieved convergence with respect to the L2 errornorm. We got the same tendency as from the time dependent Stokes results i.e the projection algorithm solves the problems faster, but with less accuracy.

To summarize the two problems based on the results, we observe that the Navier-Stokes problem has a slightly longer CPU- time. However, this is expected since we have added a convection term. The error is of the same order in the projection method for both the time dependent Stokes Problem and the Navier-Stokes problem. But for the mixed method, we can suggest a slightly better error estimate for the time dependent Stokes simulation versus the Navier-Stokes simulation.

If we add up, the results of the simulations conclude that the implementation is correct for the projection method and the mixed element method. Simulation results favor the mixed method, if we require the best possible accuracy of our solutions. However, if we want a fast computed approximate solution, the projection algorithm is preferable. In the next section we will try to improve the CPU time of the mixed method by employing a block preconditioner.

6.2.3 Block Preconditioner

The idea of introducing a block preconditioner is to improve the CPU-time. We have seen from the results in Chapter 6.1.3 that the algebraic multigrid preconditioner achieved a reduction of the overall CPU-time, for the Poisson problems. The aim of this sections will be to accomplish something similar for the system (6.9).

The preconditioning will be carried out on the mixed element method, as this method gave the best results in the previous section. From [16] an efficient block preconditioner for the system (6.9) with k=0 is on the form:

$$B = \begin{bmatrix} (I - \epsilon\Delta)^{-1} & 0 \\ 0 & (-\Delta)^{-1} + \epsilon I \end{bmatrix} \tag{6.12}$$

The program code for the creation of a block preconditioner of this type, see Appendix A.3.

Mixed element method 2D problem preconditioned

$P_2 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	4	8	16	32	64
0.2	1.01×10^{-6} / 1.22	6.80×10^{-8} / 3.09	5.98×10^{-9} / 10.5	4.72×10^{-9} / 42.3	4.75×10^{-9} / 197.5
0.1	1.01×10^{-6} / 2.65	6.85×10^{-8} / 6.22	4.42×10^{-9} / 21.8	2.80×10^{-10} / 88.17	*out of memory*
0.01	1.01×10^{-6} / 21.6	6.85×10^{-8} / 52.4	4.42×10^{-9} / 180.0	*out of memory*	*out of memory*

Mixed element method 3D problem preconditioned

$P_2 - P_1$

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	2	4	8
0.2	1.19×10^{-5} / 4.37	1.29×10^{-6} / 55.9	9.54×10^{-8} / 466.6
0.1	1.19×10^{-5} / 9.58	1.29×10^{-6} / 125.9	9.58×10^{-8} / 1 002.0
0.1	1.19×10^{-5} / 81.2	1.29×10^{-6} / 881.5	*no result*

Results:

The goal was to increase efficiency and reduce the CPU-time of the mixed algorithm, by applying a preconditioner. Theory and results from Chapter 6.1.3 with the plot in Figure 6.1 suggests this idea. However, if we compare the results from the tables of the mixed algorithm in Chapter 6.2.1 with the results of this section, we can witness that the CPU time has increased. The calculations does not support the theory.

A possible explanation may be, a lack of computer strength. Accordingly, the computer that was used "ran out of memory" before we could evaluate linear systems of a size where the order optimal algorithm would have been superior. Another explanation may be inadequate implementation. Since the libraries we used for solving the Poisson problems do not support block preconditioning, we could not apply them here.

6.3 Aneurysm Simulation

We will in this section perform simulations of flow through a blood vessel containing an aneurysm. Due to the limited time frame for this thesis, this is not completely realistic simulations of a blood flow. However, it demonstrates a verified discretization of the Navier-Stokes equations by the Finite Element Method.

Geometry

Figure 6.3 shows the mesh of a blood vessel containing an aneurysm. The mesh is based on a medical data-set. We have applied VMTK (Chapter 5.1) to reconstruct the surface and generated a mesh, from the medical data-set.

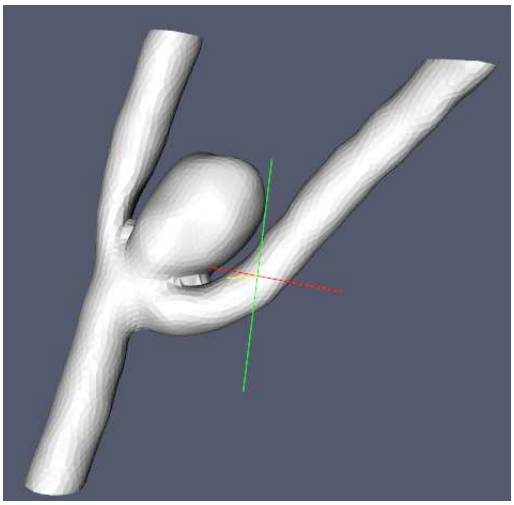


Figure 6.3: A generated surface mesh of a blood vessel containing an aneurysm

Boundary Conditions

In numerical simulations of blood flow problems, it is a challenge to define appropriate boundary conditions. To have a well-posed Navier-Stokes problem we must choose the boundary conditions in a satisfactory manner. It can be discussed that in this context with our simplified problem with respect to the choice of boundary conditions, has violated this.

We define $\partial\Omega$ to be the surface of Figure 6.3. Where $\partial\Omega = \Gamma_0 \bigcup \Gamma_1 \bigcup \Gamma_2 \bigcup \Gamma_w$. In our simulations we choose a known (Dirichlet condition) velocity profile on the inflow boundary Γ_0, see Figure 6.4;

$$\mathbf{v}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 + sin(t) \end{bmatrix}. \tag{6.13}$$

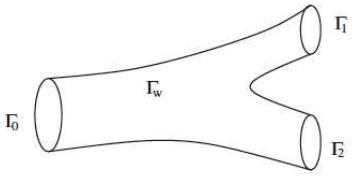


Figure 6.4: boundaries

For the outflow boundaries Γ_1 and Γ_2, we have homogeneous pressure condition $p = 0$. On the rigid vascular wall Γ_w, we assume that the velocity profile is trivial i.e no slip conditions,

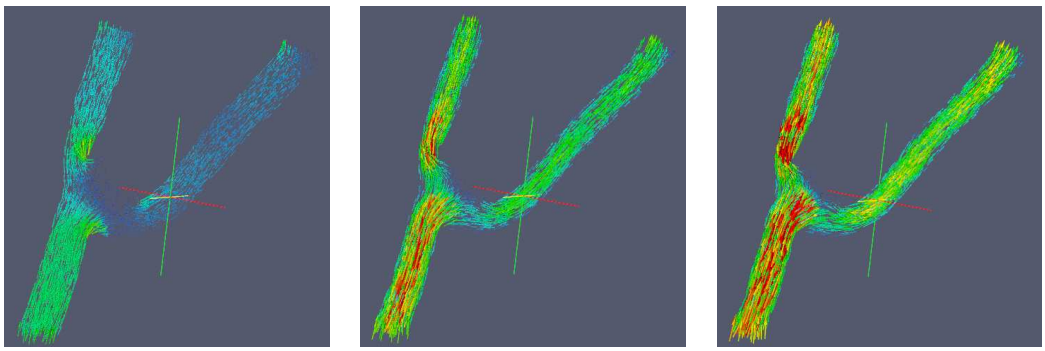
$$\mathbf{v}_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{6.14}$$

We have applied the software tool Meshbuilder (Chapter 5.3), to set the different boundary conditions. Note that the numerical solution will be greatly dependent on the choice of boundary conditions included above.

6.3.1 Simulation

From results in Chapter 6.2.2, we choose to carry out the projection simulations with $P_1 - P_1$ elements. Since this gave almost identical error estimates, but the CPU-time was faster with $P_1 - P_1$ elements.

The simulation of the velocity field is shown in the figures below. The figures describing the velocity field presented at three different times.



(a) Initial velocity field $t = 0.0$. (b) Velocity field at $t = 0.4$. (c) Velocity field at $t = 1.0$.

The visualization application *paraview(http://www.paraview.org/)* has been used for the analysis of the simulations.

6.3.2 Result

Observe that the velocity field evolves as time passes. The red color in the figures indicates that the velocity rate increases proportional with time. This is a good match since the z component of $\mathbf{v}_1$ will continue to rise with time, as long as $t \leq \frac{\pi}{2}$.

The simulations are performed with the projection algorithm derived in Chapter 4.2. With $\Delta t = 0.1$ and stop time $T = 1.0$, gave a CPU-time= 1 178 seconds. With $\Delta t = 0.1$ and stop time $T = 3.0$, gave a CPU-time= 3 216 seconds.

Unfortunately we were not able to perform simulations with the mixed finite element algorithm. Since the computer "ran out of memory". We can assume that the mixed algorithm would have had a favorable numerical solution, justified by the results produced earlier in this paper.

Chapter 7

Conclusion and Further Research

We have presented two numerical strategies for solving the Navier-Stokes equations, describing the motion of an incompressible Newtonian viscous fluid. A projection approach and a mixed finite element approach. Based on these strategies, we have implemented two different finite element solvers. Comparison and verification of these finite element algorithms, have been an important objective in the thesis.

The projection algorithm (Chapter 4.2) splits the complicated Navier-Stokes equations into simpler standard partial differential equations. As we can see from the results of Chapter 4.2.2, this leads to a fast and efficient solution algorithm, where we may represent the velocity and pressure by same order of basis functions. The problem with our projection strategy is the incorporation of unnatural boundary conditions on the pressure, that can result in large errors close to the boundary. A discussion of this issue is given in Chapter 4.2.2.

The other approach, the mixed finite element discretizing avoids the problem with unnatural boundary conditions. However, it requires that we solve a more complex system of ordinary differential equations with respect to time. This system of equations, stated in (4.16) and (4.17), ensures stability and has a solution if the Brezzi conditions (Chapter 4.3) are fulfilled.

We have verified our implementations with a number of test methods. As we can see from Chapter 3, the correct flow structure has been obtained in our simulations. From the results in Chapter 6.2.2, both algorithms achieved convergence with respect to the L2 errornorm. The tables from Chapter 6, show that the projection algorithm solves the problems more efficiently, and thus use a shorter processor time (CPU-time) than the mixed finite element algorithm does for identical parameter values. However, the accuracy turns out to be a lot better for the mixed finite element algorithm compared with the projection algorithm. We can even solve a problem on a coarse grid with the mixed algorithm, that will require less processor time (CPU-time) and still achieve a better accuracy, than on a fine grid with the projection algorithm. Let us demonstrate with results from the tables;

Projection Method 2D problem

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	16	32	64
0.2	4.01×10^{-3} / 0.93	3.98×10^{-3} / 1.65	3.97×10^{-3} / 4.17
0.1	8.64×10^{-4} / 1.87	8.47×10^{-4} / 2.94	8.44×10^{-4} / 9.00

Mixed element method 2D problem

The numbers in the table indicate the error and CPU time respectively.

$\Delta t \setminus$ N	16	32	64
0.2	3.56×10^{-8} / 1.65	4.81×10^{-9} / 7.20	3.86×10^{-9} / 44.60
0.1	3.45×10^{-8} / 3.50	2.17×10^{-9} / 15.7	1.36×10^{-10} / 98.2

Based on the tables, we conclude that the mixed algorithm is beneficial when we require the best possible accuracy. On the other hand, if we search for a fast computed approximated solution, the projection algorithm is preferable.

The implementations in this thesis is done with the programming language Python, and the finite element simulations are performed with the FEniCS project library DOLFIN. The FEniCS framework is a solving environment for partial differential equations. Without the help of this software framework, would parts of the implementations have been considerably more complex to execute. Since the FEniCS project is still under development, it has been complicated to apply such a sophisticated system with limited access of documentation. The process to understand and be able to apply the other software tools discussed in Chapter 5, have also been quite time-consuming. However, the pleasure of working with real medical data and be able to reconstruct surfaces of blood vessels and then perform calculations on these, have been a privilege.

In a search for a more efficient solution algorithm, we introduced iterative methods and preconditioning in Chapter 4.4. Then we performed simulations on some Poisson problems, based on this theory. As we can see from the results of Chapter 6.1.3, we found a combination of an algebraic multigrid preconditioner and a conjugate gradient solver, that reduced the CPU-time significantly. We got an order optimal method $O(n)$, see Figure 6.1. The purpose was to carry out something similar for the more advanced system (6.9), by introducing a block preconditioner (Chapter 6.2.3). Due to the limited time frame we were not able to solve this problem. However, this is an interesting question in our further research. A possible explanation may be the lack of computer strength, since the

computer ran out of memory. A discussion of the problem is given in Chapter 6.2.3.

There had also been convenient with a more extensive perspective on the aneurysm simulations in Chapter 6.3. By performing several simulations with a wider setup, and more realistic input data. Again because of the limited time, we could not go deeper into this subject. We have after all a very valuable basis for further research on a topic that is in great development these days.

Appendix A

Implementation

A.1 Poisson Problem

```
"""
This program solve Poisson 2D problem with different solution
algorithms.
From commandline:
    N=
    method= iterativ or direct
    bla bla
"""
from PyTrilinos import Epetra, AztecOO, TriUtils, ML
from dolfin import *
import time
import sys

# input data from command line
def get_command_line_arguments():
    dict = {}
    if len(sys.argv) == 1: return dict
    for a in sys.argv[1:]:
        key, value = a.split('=')
        dict[key] = value
    return dict

cl_args = get_command_line_arguments()

if cl_args.has_key("method"):
    if cl_args.get("method") == "iterativ":
        parameters["linear_algebra_backend"] = "Epetra"


N = 10
if cl_args.has_key("N"):
    N = int(cl_args["N"])
```

```
# Create mesh and define function space
mesh = UnitSquare(N,N)
V = FunctionSpace(mesh, "CG", 1)

# Define boundary conditions
#u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
u0 = Function(V)

# define the Dirichlet boundary
class Boundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary

u0_boundary = Boundary()
bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
v = TestFunction(V)
u = TrialFunction(V)
f = Expression('sin(x[1])')
U = Function(V)

# Define matrix A and vector b
A = 0
b = 0

alt = 1
if cl_args.has_key("alt"):
    alt = int(cl_args["alt"])
    if alt == 1:
        a = dot(grad(u), grad(v))*dx
        L = f*v*dx
        # Assemble symmetric matrix and vector
        A, b = assemble_system(a, L, bc)
    elif alt == 2:
        a = u*v*dx + 0.0001*dot(grad(u), grad(v))*dx
        L = f*v*dx
        # Assemble symmetric matrix and vector
        A, b = assemble_system(a, L)
else:
    print "ERROR : Select alt=1 OR alt=2"

A_epetra = down_cast(A).mat()
b_epetra = down_cast(b).vec()
x_epetra = down_cast(U.vector()).vec()

# Sets up the parameters for ML using a python dictionary
MLList = {"max levels"        : 30,
          "output"            : 1,
          "smoother: type"    : "ML symmetric Gauss-Seidel",
          "aggregation: type" : "Uncoupled",
```

```
        "ML validate parameter list" : False
}

if cl_args.has_key("method"):
    if cl_args.get("method") == "iterativ":
        if cl_args.has_key("solver") and cl_args.has_key("precond"):
            if cl_args.get("solver") == "cg" and cl_args.get("
                precond") == "amg":
                # Create the preconditioner
                ml_prec = ML.MultiLevelPreconditioner(A_epetra,
                    False)
                ml_prec.SetParameterList(MLList)
                ml_prec.ComputePreconditioner()
                #Create solver and solve the system
                t0 = time.time()
                Solver = AztecOO.AztecOO(A_epetra, x_epetra,
                    b_epetra)
                Solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.
                    AZ_cg)
                Solver.SetPrecOperator(ml_prec)
                Solver.Iterate(1550, 1e-5)
                t1 = time.time()
                print "TIME USED FOR SIMULATION ", t1-t0
                #plot(U)
                #plot(mesh)
                #Hold plot
                #interactive()
            if cl_args.get("solver") == "none" and cl_args.get("
                precond") == "amg":
                # Create the preconditioner
                ml_prec = ML.MultiLevelPreconditioner(A_epetra,
                    False)
                ml_prec.SetParameterList(MLList)
                ml_prec.ComputePreconditioner()
                #Create solver and solve the system
                t0 = time.time()
                Solver = AztecOO.AztecOO(A_epetra, x_epetra,
                    b_epetra)
                Solver.SetPrecOperator(ml_prec)
                Solver.Iterate(1550, 1e-5)
                t1 = time.time()
                print "TIME USED FOR SIMULATION ", t1-t0
                #plot(U)
                #plot(mesh)
                #Hold plot
                #interactive()
            if cl_args.get("solver") == "cg" and cl_args.get("
                precond") == "jacobi":
                t0 = time.time()
                Solver = AztecOO.AztecOO(A_epetra, x_epetra,
                    b_epetra)
```

```
        Solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.
            AZ_cg)
        Solver.SetAztecOption(AztecOO.AZ_precond, AztecOO.
            AZ_Jacobi)
        Solver.Iterate(1550, 1e-5)
        t1 = time.time()
        print "TIME USED FOR SIMULATION ", t1-t0
        #plot(U)
        #plot(mesh)
        #Hold plot
        #interactive()
    if cl_args.get("solver") == "cg" and cl_args.get("
        precond") == "ilu":
        t0 = time.time()
        Solver = AztecOO.AztecOO(A_epetra, x_epetra,
            b_epetra)
        Solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.
            AZ_cg)
        Solver.SetAztecOption(AztecOO.AZ_precond, AztecOO.
            AZ_dom_decomp)
        Solver.SetAztecOption(AztecOO.AZ_subdomain_solve,
            AztecOO.AZ_ilu)
        Solver.Iterate(1550, 1e-5)
        t1 = time.time()
        print "TIME USED FOR SIMULATION ", t1-t0
        #plot(U)
        #plot(mesh)
        #Hold plot
        #interactive()
    if cl_args.get("solver") == "cg" and cl_args.get("
        precond") == "none":
        t0 = time.time()
        Solver = AztecOO.AztecOO(A_epetra, x_epetra,
            b_epetra)
        Solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.
            AZ_cg)
        Solver.SetAztecOption(AztecOO.AZ_precond, AztecOO.
            AZ_none)
        Solver.Iterate(1550, 1e-5)
        t1 = time.time()
        print "TIME USED FOR SIMULATION ", t1-t0
        #plot(U)
        #plot(mesh)
        #Hold plot
        #interactive()
    if cl_args.get("solver") == "gmres" and cl_args.get("
        precond") == "none":
        t0 = time.time()
        Solver = AztecOO.AztecOO(A_epetra, x_epetra,
            b_epetra)
```

```
                Solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.
                    AZ_gmres)
                #Solver.SetAztecOption(AztecOO.AZ_precond, AztecOO.
                    AZ_dom_decomp)
                #Solver.SetAztecOption(AztecOO.AZ_subdomain_solve,
                    AztecOO.AZ_ilu)
                Solver.Iterate(1550, 1e-5)
                t1 = time.time()
                print "TIME USED FOR SIMULATION ", t1-t0
                #plot(U)
                #plot(mesh)
                #Hold plot
                #interactive()


        else:
            print "ERROR : Select solver ex: cg, none AND precond ex
                : amg, jacobi, ilu"

    if cl_args.get("method") == "direct":
        t0 = time.time()
        solve(A, U.vector(), b, "lu")
        t1 = time.time()
        print "TIME USED FOR SIMULATION ", t1-t0
        #Plot solution and mesh
        #plot(U)
        #plot(mesh)
        #Hold plot
        #interactive()

else:
    print "ERROR : Select method, iterativ OR direct"
```

A.2 Projection Algorithm

```
from dolfin import *
from PyTrilinos import Epetra, AztecOO, TriUtils, ML
import sys
import pylab
import time

# set parameters from command-line
bryter = 0.0
p_grad = 1.0

bcv = []
bcp = []
bc = [bcv,bcp]

from arguments import get_command_line_arguments
cl_args = get_command_line_arguments()

N = 2
if cl_args.has_key("N"):
    N = int(cl_args["N"])

# create mesh and define function space
mesh = UnitSquare(N,N)

if cl_args.has_key("problem"):
    if cl_args.get("problem") == "testproblem3d" or cl_args.get("
        problem") == "testproblem3dtime":
        mesh = UnitCube(N,N,N)
    if cl_args.get("problem") == "anu":
        # Read mesh and subdomains from file
        mesh = Mesh("../anudata/dog_assignment.xml.gz")
        subdomains = MeshFunction("uint", mesh, "../anudata/
            dog_assignment_bc.xml")
        boundary_func1 = Expression(("0.0","0.0","1.0+sin(t)"))
        boundary_func2 = Expression(("0.0","0.0","0.0"))
        boundary_func3 = Expression("0.0")
        #boundary_func.t = t

V = VectorFunctionSpace(mesh, "CG", 1)
Q = FunctionSpace(mesh, "CG", 1)


# define boundary conditions and initial condition
if cl_args.has_key("problem"):
    if cl_args.get("problem") == "testproblem2d":
        boundary_func = Expression(("sin(x[1])","sin(x[0])"))
        #boundary_func.t = t
    if cl_args.get("problem") == "testproblem3d":
```

```
        print "kommer inn i 3D tidsuavhengig"
        boundary_func = Expression(("sin(x[1])","sin(x[0])","0.0"))
        #boundary_func.t = t
    if cl_args.get("problem") == "testproblem3dtime":
        print "kommer inn i 3D tidsavhengig"
        boundary_func = Expression(("sin(x[1])*sin(t)","sin(x[0])*
            sin(t)","0"))
        boundary_func.t = 0

class Boundary(SubDomain):
    def inside(self, x, on_boundary):
        if on_boundary:
            return True

if cl_args.get("problem") == "anu":
    bc_v1 = DirichletBC(V, boundary_func2, subdomains, 3)
    bc_v2 = DirichletBC(V, boundary_func1, subdomains, 1)
    bc_p = DirichletBC(Q, boundary_func3, subdomains, 2)
    bcv.append(bc_v1)
    bcv.append(bc_v2)
    bcp.append(bc_p)

else:
    boundary = Boundary()
    bc_v = DirichletBC(V, boundary_func, boundary)
    bcv.append(bc_v)

u0 = Function(V)
p0 = Function(Q)

#define time conditions
delta_t = 0.1
nuu = 1.0
if cl_args.has_key("delta_t"):
    delta_t = float(cl_args["delta_t"])

if cl_args.has_key("nuu"):
    nuu = float(cl_args["nuu"])

dt = Constant(delta_t)
NUU = Constant(nuu)
epsilon = dt*NUU
epsiloninv = (1.0/epsilon)


# define correct function f
f = None
if cl_args.has_key("problem"):
    if cl_args.get("problem") == "testproblem2d":
        if cl_args.get("conv") == "on":
            bryter = 1.0
```

```
            f = Expression(("sin(x[0])*cos(x[1]) + nuu*sin(x[1]) -
                p_grad","sin(x[1])* cos(x[0]) + nuu* sin(x[0])"))
        else:
            f = Expression(("nuu*sin(x[1]) - p_grad","nuu*sin(x[0])
                "))

    if cl_args.get("problem") == "testproblem3d":
        if cl_args.get("conv") == "on":
            bryter = 1.0
            f = Expression(("sin(x[0])*cos(x[1]) + nuu*sin(x[1]) -
                p_grad","sin(x[1])* cos(x[0]) + nuu* sin(x[0])",
                "0.0"))
        else:
            f = Expression(("nuu*sin(x[1]) - p_grad","nuu*sin(x[0])
                ","0.0"))
            #boundary_func.t = t
    if cl_args.get("problem") == "testproblem3dtime":
        if cl_args.get("conv") == "on":
            bryter = 1.0
            f = Expression(("sin(x[1])*cos(t)+sin(t)*sin(t)*sin(x
                [0])*cos(x[1])+nuu*sin(x[1])*sin(t)- p_grad","sin(x
                [0])*cos(t)+sin(t)*sin(t)*sin(x[1])*cos(x[0])+nuu*sin
                (x[0])*sin(t)", "0.0"))
            f.t = 0
        else:
            f = Expression(("sin(x[1])*cos(t)+nuu*sin(x[1])*sin(t)-
                p_grad","sin(x[0])*cos(t)+nuu*sin(x[0])*sin(t)",
                "0.0"))
            f.t = 0

if cl_args.get("problem") == "anu":
    f = Function(V)
else:
    f.nuu = nuu
    f.p_grad = p_grad

# define variational problem
Us = Function(V)
U = Function(V)
u = TrialFunction(V)
v = TestFunction(V)

Ps = Function(Q)
P = Function(Q)
p = TrialFunction(Q)
q = TestFunction(Q)


a0= dot(u,v)*dx + epsilon*inner(grad(u), grad(v))*dx + bryter*(
    epsilon/NUU)*dot(dot(u0,grad(u)),v)*dx
L0= dot((u0 + f*(epsilon/NUU)),v)*dx + dot(grad(p0), v)*dx
```

```
a1= -dot(grad(p), grad(q))*dx
L1= -div(Us)*q*dx

a2= dot(u,v)*dx
L2= dot(Us, v)*dx + dot(grad(Ps), v)*dx

a3= dot(p, q)*dx
L3= dot(Ps, q)*dx + p0*q*dx

T = 3.0
t = 0
time_array = []
feil = []

t0 = time.time()

# Output file
out_file = File("hastighets_f.pvd")
while t < T:
    print "time =", t
    if cl_args.get("problem") == "anu":
        boundary_func1.t=t
        moro=0.0
    else:
        boundary_func.t = t

    f.t = t

    A0, b0 = assemble_system(a0, L0, bcv)
    solve(A0, Us.vector(), b0)

    A1, b1 = assemble_system(a1, L1, bcp)
    solve(A1,Ps.vector(), b1)

    A2, b2 = assemble_system(a2, L2)
    solve(A2, U.vector(), b2)

    A3, b3 = assemble_system(a3, L3)
    solve(A3, P.vector(), b3)

    if cl_args.get("problem") == "anu":
        out_file << U
    else:
        # verify
        u_exact = interpolate(boundary_func, V)
        E = u_exact - U
        norm = sqrt( assemble(dot(E,E)*dx, mesh = mesh) )
        feil.append(norm)

    u0.assign(U)
```

```
    p0.assign(P)
    t += delta_t

t1 = time.time()
print "TIME USED FOR SIMULATION ", t1-t0
```

A.3 Mixed Finite Elements Algorithm

```
"""
we solve here the following equation:
    u_t + bryter * u * grad(u) - k*div grad(u) + grad(p) = f    in
        omega
                            div(u) = 0    in omega
                                u  = 0    in delta_omega

We study time dependent Stokes/Navier Stokes(if bryter=1)
input data from commandline:
    N= 1,2,3 ...
    problem= testproblem2d, testproblem3d, testproblem3dtime or
        meshproblem
    precond=amg or none
    delta_t=
    nuu=
    conv= on or off
"""

from dolfin import *
from PyTrilinos import Epetra, AztecOO, TriUtils, ML
import SaddlePrec2, Krylov, MinRes
import sys
import pylab
import time

# set parameters from command-line
bryter = 0.0
p_grad = 1.0

bcv = []
bcp = []
bc = [bcv,bcp]

from arguments import get_command_line_arguments
cl_args = get_command_line_arguments()

N = 2
if cl_args.has_key("N"):
    N = int(cl_args["N"])

# create mesh and define function space
mesh = UnitSquare(N,N)

if cl_args.has_key("problem"):
    if cl_args.get("problem") == "testproblem3d" or cl_args.get("
        problem") == "testproblem3dtime":
        mesh = UnitCube(N,N,N)

    if cl_args.get("problem") == "anu":
```

```
        mesh = Mesh("../anudata/dog_assignment.xml.gz")
        subdomains = MeshFunction("uint", mesh, "../anudata/
            dog_assignment_bc.xml")
        boundary_func1 = Expression(("0.0","0.0","1.0+sin(t)"))
        boundary_func2 = Expression(("0.0","0.0","0.0"))
        boundary_func3 = Expression("0.0")
        #boundary_func.t = t

V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
mixed = V + Q


# define boundary conditions and initial condition
if cl_args.has_key("problem"):
    if cl_args.get("problem") == "testproblem2d":
        boundary_func = Expression(("sin(x[1])","sin(x[0])"))
    if cl_args.get("problem") == "testproblem3d":
        boundary_func = Expression(("sin(x[1])","sin(x[0])","0.0"))
    if cl_args.get("problem") == "testproblem3dtime":
        boundary_func = Expression(("sin(x[1])*sin(t)","sin(x[0])*
            sin(t)","0"))
        boundary_func.t = 0

class Boundary(SubDomain):
    def inside(self, x, on_boundary):
        if on_boundary:
            return True

if cl_args.get("problem") == "anu":
    bc_v1 = DirichletBC(V, boundary_func2, subdomains, 3)
    bc_v2 = DirichletBC(V, boundary_func1, subdomains, 1)
    bc_p = DirichletBC(Q, boundary_func3, subdomains, 2)
    bcv.append(bc_v1)
    bcv.append(bc_v2)
    bcp.append(bc_p)

else:
    boundary = Boundary()
    bc_v = DirichletBC(V, boundary_func, boundary)
    bcv.append(bc_v)

u0 = Function(V)
p0 = Function(Q)

#define time conditions
delta_t = 0.1
nuu = 1.0
if cl_args.has_key("delta_t"):
    delta_t = float(cl_args["delta_t"])
```

```
if cl_args.has_key("nuu"):
    nuu = float(cl_args["nuu"])

dt = Constant(delta_t)
NUU = Constant(nuu)
epsilon = dt*NUU
epsiloninv = (1.0/epsilon)

# define correct function f
f = None
if cl_args.has_key("problem"):
    if cl_args.get("problem") == "testproblem2d":
        if cl_args.get("conv") == "on":
            bryter = 1.0
            f = Expression(("sin(x[0])*cos(x[1]) + nuu*sin(x[1]) -
                p_grad","sin(x[1])* cos(x[0]) + nuu* sin(x[0])"))
        else:
            f = Expression(("nuu*sin(x[1]) - p_grad","nuu*sin(x[0])
                "))
    if cl_args.get("problem") == "testproblem3d":
        if cl_args.get("conv") == "on":
            bryter = 1.0
            f = Expression(("sin(x[0])*cos(x[1]) + nuu*sin(x[1]) -
                p_grad","sin(x[1])* cos(x[0]) + nuu* sin(x[0])",
                "0.0"))
        else:
            f = Expression(("nuu*sin(x[1]) - p_grad","nuu*sin(x[0])
                ","0.0"))
    if cl_args.get("problem") == "testproblem3dtime":
        if cl_args.get("conv") == "on":
            bryter = 1.0
            f = Expression(("sin(x[1])*cos(t)+sin(t)*sin(t)*sin(x
                [0])*cos(x[1])+nuu*sin(x[1])*sin(t)- p_grad","sin(x
                [0])*cos(t)+sin(t)*sin(t)*sin(x[1])*cos(x[0])+nuu*sin
                (x[0])*sin(t)", "0.0"))
            f.t = 0
        else:
            f = Expression(("sin(x[1])*cos(t)+nuu*sin(x[1])*sin(t)-
                p_grad","sin(x[0])*cos(t)+nuu*sin(x[0])*sin(t)",
                "0.0"))
            f.t = 0

if cl_args.get("problem") == "anu":
    f = Function(V)
else:
    f.nuu = nuu
    f.p_grad = p_grad


# define variational problem
u, p_scale = TrialFunctions(mixed)
```

```
v, q = TestFunctions(mixed)

a = dot(u,v)*dx + epsilon*inner(grad(u), grad(v))*dx + p_scale*div(v
    )*dx + div(u)*q*dx + bryter*(epsilon/NUU)*dot(dot(u0,grad(u)),v)*
    dx
L = dot((epsilon/NUU)*f + u0,v)*dx

if cl_args.has_key("precond"):
    if cl_args.get("precond") == "amg":
        parameters["linear_algebra_backend"] = "Epetra"
        g = Constant(0)
        # create preconditioner AMG
        uu = TrialFunction(V)
        vv =  TestFunction(V)
        pp_scale = TrialFunction(Q)
        qq =  TestFunction(Q)

        kk = dot(uu, vv)*dx +  epsilon*inner(grad(uu), grad(vv))*dx
        ll = epsiloninv*pp_scale*qq*dx
        mm = dot(grad(pp_scale),grad(qq))*dx
        L00 = dot((epsilon/NUU)*f + u0,vv)*dx
        L11 = qq*g*dx

        KK, b00 = assemble_system(kk,L00,bcv)
        LL, b11 = assemble_system(ll,L11)
        MM, b11 = assemble_system(mm,L11)

        B = SaddlePrec2.SaddlePrec2(KK, LL, MM)

        U = Function(V)
        P = Function(Q)
        U_size = U.vector().size()

T = 0.3
t = 0
time_array = []
feil = []

t0 = time.time()
# Output file
out_file = File("hastighets_f.pvd")
while t < T:
    print "time =", t
    if cl_args.get("problem") == "anu":
        boundary_func1.t=t
    else:
        boundary_func.t = t

    f.t = t

    A, b = assemble_system(a, L, bcv)
```

```
    if cl_args.has_key("precond"):
        if cl_args.get("precond") == "amg":
            x = Vector(b.size())
            x.zero()
            x, e, iter  = Krylov.CGN_BABA(B, A, x, b, 10e-12, True,
                1000)
            print "Sqrt of condition number of BABA  ", sqrt(e[len(e
                )-1]/e[0])
            U.vector()[:] = x.array()[0:U_size]

    else:

        U = Function(V)
        P = Function(Q)

        solve(A, U.vector(), b)
        solve(A, P.vector(), b)
        #file = File("A_%d.m"% float(delta_t))
        #file << A

    if cl_args.get("problem") == "anu":
        out_file << U
    else:
        # verify
        u_exact = interpolate(boundary_func, V)
        E = u_exact - U
        norm = sqrt( assemble(dot(E,E)*dx, mesh = mesh) )
        feil.append(norm)

    u0.assign(U)
    p0.assign(P)
    t += delta_t

t1 = time.time()
print "TIME USED FOR SIMULATION ", t1-t0
```

Bibliography

[1] *http://www.vmtk.org/.*

[2] Susanne C. Brenner and L.Ridgway Scott, *The mathematical theory of finite element methods*, Springer, 2008.

[3] Are Magnus Bruaset and Aslak Tveito, *Numerical solution of partial differential equations on parallel computers*, Springer, 2006.

[4] Howard Elman, David Silvester, and Andy Wathen, *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*, Oxford University Press, 2005.

[5] Lawrence C. Evans, *Partial differential equations*, American Mathematical Society, 1998.

[6] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer, *Numerical simulation in fluid dynamics: A practical introduction*, Society for Industrial and Applied Mathematics, 1998.

[7] Clay Mathematics Institute, *Millennium prize problems.*

[8] P. Minev J.L. Guermond and Jie Shen, *An overview of projection methods for incompressible flows.*

[9] Anders Logg Kent-Andre Mardal and Garth Wells, *Automated scientific computing.*

[10] Ragnar Winther Kent-Andre Mardal and Hans Petter Langtangen, *Numerical methods for incompressible viscous flow.*

[11] Pijush K. Kundu and Ira M. Cohen, *Fluid mechanics*, McGraw- Hill Book Co., 2008.

[12] H. P. Langtangen, *A fenics tutorial.*

[13] Hans Petter Langtangen, *Computational partial differential equations*, Springer, 2002.

[14] Tom Lyche, *Lecture notes for inf-mat 4350*, (2008).

[15] Kent-Andre Mardal, *Solving linear systems*.

[16] Kent-Andre Mardal and Ragnar Winther, *Preconditioning discretizations of system of partial differential equations*.

[17] Frank M. White, *Viscous fluid flow*, McGraw- Hill Book Co., 1991.