European Consortium for Mathematics in Industry

28th ECMI Modelling Week
Final Report

19.07.2015—26.07.2015
Lisboa, Portugal

Group 5

Patient-specific blood flow modelling

Anastasiia Bozhok
Grenoble INP (ENSIMAG)
Gaetano Formato
Lappeenranta University of Technology
Christoph Sadée
University College Dublin
Milena Mandic
University of Novi Sad
Pavel Iliev
University of Sofia
Janvier Ukwizagira
Lappeenranta University of Technology

Instructor: Dr. Jorge Tiago
CEMAT, IST-ULisboa

Abstract

The purpose of this work is to develop a 2D mathematical model of blood flow and to forecast some cardiovascular diseases behavior such as aneurysms and stenosis. In order to complete our model, we should consider the variability in design and size of cardiovascular systems between individuals.

We compute a blood flow model based on Stokes and Navier-Stokes equations. Furthermore, we would like to simulate the behaviour of aneurysms and stenosis in the cardiovascular system by using the model made up. Our primary interests are in the mechanical properties of the system, which will be the principal target of the models discussed in this paper.

Finally, we would like to minimize the misfit between the computed blood flow and the real cross-sections data of specific patients by controlling the choice of the inflow boundary conditions.

5.1 Introduction

Medical images allow us to obtain good representations of blood vessels, even in pathological cases. However concerning the dynamics of the blood itself, medical images can only give us some sparse velocity measurements. Medical doctors would make a good use of accurate blood flow simulations in predicting either the evolution of certain pathologies, or the effect of some therapies. Our project consists in finding a model which allows one to reconstruct the blood flow in the complete domain by using the information obtained from the velocity measurements.

We have largely ignored the enormous and medically very important subject of cardiovascular pharmacology. Drug delivery and action is undoubtedly influenced by haemodynamics but little is known about these interactions.

5.2 Cardiovascular system and diseases

The cardiovascular system (Fig. 5.1) is composed of the heart, which pumps the blood through the body and a network of blood vessels which transport blood to the body and drain it from the body tissues to the heart. All parts of the system work together, but will be considered individually. Blood is ejected from the heart in discrete pulses under relatively high pressure into the main arteries (at a lower pressure in the pulmonary circulation than in the systemic circulation) where it flows through a network of branching arteries of decreasing size to the arterioles and then the capillaries where it delivers oxygen and nutrients to the tissues and removes carbon dioxide. Blood is collected from the capillaries through merging venues and returns to the heart at low pressure through a network of veins.

The most common diseases, affecting the cardiovascular system are aneurysm's (Fig. 5.3) and stenosis' (Fig. 5.2).

A stenosis is an obstruction of blood flow caused by the development of plaque of atherosclerosis.

An aneurysm is a gradual dilation of an arterial segment over a period of years. The aneurysm wall stretches and becomes thinner and weaker than normal arterial walls. This phenomenon could cause the rupture of the vein wall, causing massive haemorrhage, which is often lethal.

5.3 Model

Assumptions

In order to develop a 2D mathematical model for blood flow, we fix some assumptions:

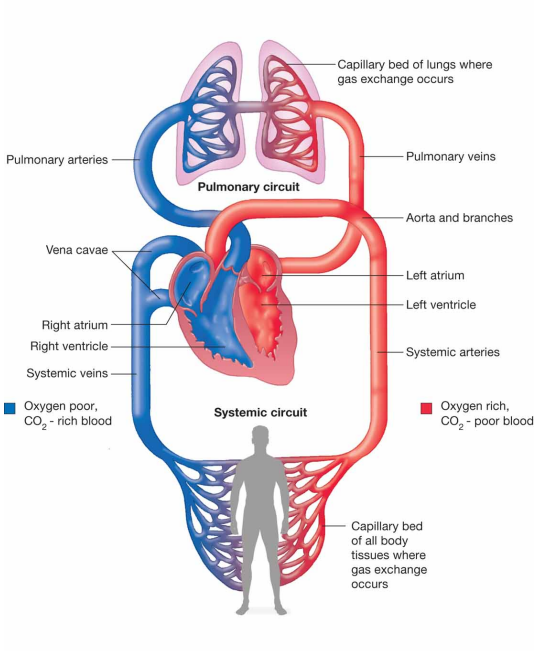


Figure 5.1: A sketch of the cardiovascular system.

1) Blood is modeled as homogeneous fluid flow .

2) We consider in our simulation the stationary case. Therefore the pulsatile nature of blood is neglected .

3) The temperature is constant at 37 °C.

4) Blood is assumed to be a Newtonian fluid wirh constant viscosity equal to 3.5 mPa · s .

5) Blood density is considered to be constant $1.06 \cdot 10^{\ 3}$ Kg/m^3 .

6) We assume the non-slip boundary at the artery wall, that is, the velocity is set to be zero in every direction.

7) The Reynolds number for blood is 100 – 400. This corresponds to a non turbulent motion.

8) We consider blood to be an incompressible fluid. From the mass conservation equation, we get:

$$\nabla \cdot \mathbf{u}(x; y) = 0$$

where $\mathbf{u}$ is the 2D velocity field of the fluid.

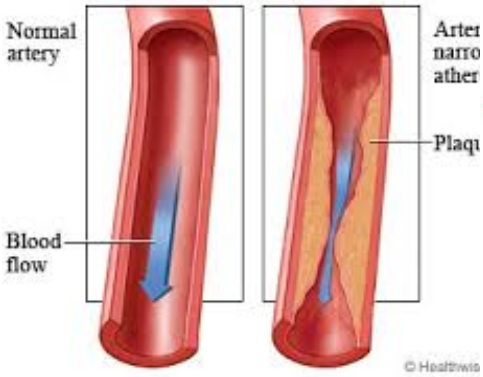


Figure 5.2: Stenosis development

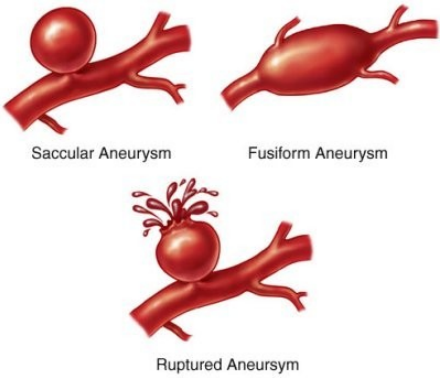


Figure 5.3: Aneurism development

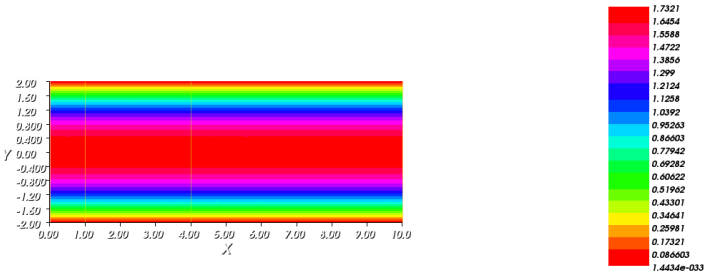


Figure 5.4: Steady state solution to non-stress Stokes problem with Dirichlet boundary condition on one boarder.

Mathematical model for blood flow

In order to obtain patient specific blood flow from measurements we have to define a model for the blood flow. Under the assumptions that have been discussed in the previous chapter the for steady blood flow is expressed by the steady Navier-Stokes equations. Any type of flow is generally described by Navier-Stokes equation (PDE). The assumptions are used to adjust the PDE to fit our model. The first two equation in (5.1) correspond to linear momentum and mass conservation respectively. Here $\mathbf{u}$ denotes the velocity field, p is the pressure, ρ and μ are density and viscosity of the blood. We assume that Ω represents the fluid domain, and the boundaries of Ω consist of Γ_{Wall}, which represents the inside wall of the artery and two artificial boundaries Γ_{In} and Γ_{Out}. Γ_{In} corresponds to the border of inlet flow and Γ_{Out} to the border of outlet flow of the considered section of the artery. We consider Dirichlet boundary condition on Γ_{In} and Neumann boundary condition on Γ_{Out} and no-slip Dirichlet boundary condition on Γ_{Wall}.

$$\begin{cases} \rho(\mathbf{u}\cdot\nabla\mathbf{u}) - \nabla\cdot\mathbf{T}(\mathbf{u},p) = 0 \quad \text{on } \Omega \\ \nabla\cdot\mathbf{u} = 0 \quad \text{on } \Omega \\ \mathbf{u} = 0 \quad \text{on } \Gamma_{Wall} \\ \mathbf{n}\cdot\mathbf{T} = 0 \quad \text{on } \Gamma_{Out} \\ \mathbf{u} = \mathbf{c} \quad \text{on } \Gamma_{In} \end{cases} \tag{5.1}$$

Using the Cauchy stress principle

$$\mathbf{T}(\mathbf{u},p) = -p\mathbf{I} + 2\mu D(\mathbf{u}) \tag{5.2}$$

where

$$D(\mathbf{u}) = \frac{\bigtriangledown\mathbf{u} + \bigtriangledown\mathbf{u}^T}{2} \tag{5.3}$$

and when $\nabla\cdot\mathbf{u} = 0$, it can be shown that

$$\text{div}(D(\mathbf{u})) = \Delta\mathbf{u} \tag{5.4}$$

hence, the first equation in (5.1) takes the form:

$$\rho(\mathbf{u}\cdot\nabla\mathbf{u}) - \mu\Delta\mathbf{u} + \nabla p\mathbf{I} = 0 \quad \text{on } \Omega \tag{5.5}$$

We use this system to predict the behaviour of the velocity magnitudes and pressure in the artery for cases of stenosis and aneurusm. The function $\mathbf{c}$, defining the inlet flow is set to be a parabolic function for simulations, while in reality it depends on patient specific blood flow, it's importance will be discussed later.

Weak Formulation

To obtain a weak formulation of the system of differential equations (5.1) we define the function spaces

$$\mathbb{S} = \{\mathbf{u} \in [H^1(\Omega)],\ |\ \mathbf{u} = \mathbf{c} \quad \text{on } \Gamma_{In},\ \mathbf{u} = \mathbf{0} \quad \text{on } \Gamma_{Wall}\} \tag{5.6}$$

$$\mathbb{V} = \{\mathbf{u} \in [H^1(\Omega)],\ |\ \mathbf{u} = \mathbf{0} \quad \text{on } \Gamma_{In} \cup \Gamma_{Wall}\} \tag{5.7}$$

The weak form of (5.1) is to find $\mathbf{u} \in \mathbb{S}$ and $p \in Q, (Q = L^2(\Omega))$ such that for every $\mathbf{v} \in \mathbb{V}$ and $q \in Q$

$$\int_\Omega \rho(\mathbf{u}\cdot\nabla\mathbf{u})\cdot\mathbf{v}\,d\mathbf{x} + \int_\Omega \mu\nabla\mathbf{u} : \nabla\mathbf{v}\,d\mathbf{x} - \int_\Omega p\nabla\cdot\mathbf{v}\,d\mathbf{x} - \int_\Omega q\nabla\cdot\mathbf{u}\,d\mathbf{x} = 0 \tag{5.8}$$

This expression is obtained by multiplying the two governing equations in (5.1) by test functions $\mathbf{v}$ and q, respectively, integrating over the domain Ω and subtracting one equation from the other. Because the integral is a linear functional it can be further simplified to the form

$$\int_\Omega [\rho(\mathbf{u}\cdot\nabla\mathbf{u})\cdot\mathbf{v} + \mu\nabla\mathbf{u} : \nabla\mathbf{v} - p\nabla\cdot\mathbf{v} - q\nabla\cdot\mathbf{u}] = 0 \tag{5.9}$$

We denote this problem (WP). Obtaining the weak form of the system (5.1) is a key part in the present study. Solving (WP) form leads to weaker restrictions for the smoothness of the solution than solving directly the differential form (5.1). We define finite dimensional subspaces $\mathbb{V}_h \subset \mathbb{V}$ and $Q_h \subset Q$ and take finite dimensional approximations $\mathbf{u}_h \in \mathbb{V}_h$ and $p_h \in Q_h$:

$$\mathbf{u}_h = \sum_{j=1}^{N_u} u_j \phi_j \quad , \phi_j \in \mathbb{V}_h \tag{5.10}$$

$$p_h = \sum_{l=1}^{N_p} p_l \psi_l \quad , \psi_l \in Q_h \tag{5.11}$$

We use finite element spaces $\mathbb{V}_h$ and Q_h corresponding to the Taylor-Hood elements (P2,P1). One method for numerically solving (5.9) is by replacing $\mathbf{u}$ and q in (5.9) with their finite approximations (5.10) and (5.11) and obtain a discrete system. But since the expression in the integral is non-linear the discrete system is not going to be a linear system of equations. In the present study the numerical algorithm that we use to solve the weak form of (5.1) is an iterative method based on the Newton method that we will discuss in details in the next chapter. This method allows us to solve a non-linear problem that models the blood flow and hence to solve the optimization problem.

5.4 The Newton Method

The numerical algorithm that we use to solve (5.9) is based on the classical Newton's method for solving non-linear equations based on Taylor series expansion. Let

$$F(\mathbf{u}, p) = \int_\Omega [\mathbf{v} \cdot \rho(\mathbf{u} \cdot \nabla \mathbf{u}) \ + \mu \nabla \mathbf{v} \cdot \nabla \mathbf{u} \ - \nabla \mathbf{v} \cdot p\mathbf{I} \ - q \nabla \cdot \mathbf{u}] \tag{5.12}$$

The problem can be formulated as: find (**u**,p) such that $F(\mathbf{u}, p) = 0$. This is a non-linear problem and for the formulation of the numerical algorithm that we use to solve it we define the Gateaux derivative of F for all the directions $(\delta\mathbf{u}, \delta p)$ as

$$F'(\mathbf{u}, p).(\delta\mathbf{u}, \delta p) = \frac{\delta F}{\delta \mathbf{u}}(\mathbf{u}, p).(\delta\mathbf{u}) + \frac{\delta F}{\delta p}(\mathbf{u}, p).(\delta p) \tag{5.13}$$

where

$$\frac{\delta F}{\delta \mathbf{u}}(\mathbf{u}, p).(\delta\mathbf{u}) = \lim_{\epsilon \to 0} \frac{F(\mathbf{u} + \epsilon\delta\mathbf{u}, p) - F(\mathbf{u}, p)}{\epsilon} \tag{5.14}$$

and

$$\frac{\delta F}{\delta p}(\mathbf{u}, p).(\delta p) = \lim_{\epsilon \to 0} \frac{F(\mathbf{u}, p + \epsilon\delta p) - F(\mathbf{u}, p)}{\epsilon} \tag{5.15}$$

Therefore the Gateaux derivative of (5.12) takes the form

$$F'(\mathbf{u},p).(\delta\mathbf{u},\delta p) = \int_\Omega [\mathbf{v}\cdot\rho(\delta\mathbf{u}\cdot\nabla\mathbf{u}) + \mathbf{v}\cdot\rho(\mathbf{u}\cdot\nabla\delta\mathbf{u}) + \mu\nabla\mathbf{v}\cdot\nabla\delta\mathbf{u} - q\nabla\cdot\delta\mathbf{u} - \nabla\mathbf{v}\cdot\delta p\mathbf{I}] \tag{5.16}$$

The Newton algorithm can be formulated as the following iterative process

1. Choose $(\mathbf{u}_0, p_0) \in (\mathbb{V}, Q)$
2. For i=1 ... n
   (a) Solve $F'(\mathbf{u}_i, p_i).(\mathbf{w}_i, q_i) = F(\mathbf{u}_i, p_i)$
   (b) $(\mathbf{u}_{i+1}, p_{i+1}) = (\mathbf{u}_i, p_i) - (\mathbf{w}_i, q_i)$

   Break when $||(\mathbf{w}_i, q_i)|| < \epsilon$

For the initial guess $(\mathbf{u}_0, p_0)$ we use the solution obtained from solving the weak form of the linear Stokes equations:

$$\int_\Omega [\mu\nabla\mathbf{u} : \nabla\mathbf{v} - p\nabla\cdot\mathbf{v} - q\nabla\cdot\mathbf{u}] = 0 \tag{5.17}$$

Then for each following step we solve linear problem for $(\mathbf{w}_i, q_i)$. In order to solve (5.17) the linear problem for each step we use finite elements method in the subspaces $(\mathbb{V}_h, Q_h)$, defined in the previous chapter.

5.5 Time dependent Stokes Equation

Consider the following time dependent Stokes equation:

$$\rho\frac{\partial\mathbf{u}}{\partial t} - \mu\nabla^2\mathbf{u} + \nabla p\mathbf{I} = 0 \tag{5.18}$$

This is a simplified version of Navier-Stokes equation, neglecting $\mathbf{u}\cdot\nabla\mathbf{u}$. The time derivative can be discretized using the Crank-Nicolson scheme:

$$\frac{\rho}{\tau}\left(\mathbf{u}^{n+1} - \mathbf{u}^n\right) + \frac{1}{2}\left(-\mu\nabla^2\mathbf{u}^{n+1} + \nabla p^{n+1}\mathbf{I}\right) + \frac{1}{2}\left(-\mu\nabla^2\mathbf{u}^n + \nabla p^n\mathbf{I}\right) = 0 \tag{5.19}$$

The same is done for $\nabla\cdot\mathbf{u} = 0$

$$\frac{1}{2}\nabla\cdot\mathbf{u}^{n+1} + \frac{1}{2}\nabla\cdot\mathbf{u}^n = 0 \tag{5.20}$$

The next step is to derive the weak form:

$$\int\left[\frac{\rho}{\tau}\left(\mathbf{u}^{n+1} - \mathbf{u}^n\right) + \frac{1}{2}\left(-\mu\nabla^2\mathbf{u}^{n+1} + \nabla p^{n+1}\mathbf{I}\right) + \frac{1}{2}\left(-\mu\nabla^2\mathbf{u}^n + \nabla p^n\mathbf{I}\right)\right]\cdot\mathbf{v}d\Omega - \int\left[\frac{1}{2}\nabla\cdot\mathbf{u}^{n+1} + \frac{1}{2}\nabla\cdot\mathbf{u}^n\right]\cdot q d\Omega = 0 \tag{5.21}$$

Note that the following holds:

$$\int_{\Omega} \mathbf{v}\nabla^2 u^{n+1} d\Omega = \left[\cancel{\int_{\Gamma} u\nabla\mathbf{v} \cdot \mathbf{v} d\Gamma} - \int_{\Omega} \nabla\mathbf{u} \cdot \nabla\mathbf{v} d\Omega \right] \tag{5.22}$$

Hence applying Equation (5.22) to Equation (5.21) and grouping $n+1$ and n terms together yields:

$$\begin{aligned} &\int \frac{\rho}{\tau}\mathbf{u}^{n+1} \cdot \mathbf{v} + \frac{1}{2}\mu\nabla\mathbf{u}^{n+1} \cdot \nabla\mathbf{v} - \frac{1}{2}p^{n+1}\nabla \cdot \mathbf{v} - \frac{1}{2}q\nabla \cdot \mathbf{u}^{n+1} d\Omega + \\ &\int -\frac{\rho}{\tau}\mathbf{u}^{n} \cdot \mathbf{v} + \frac{1}{2}\mu\nabla\mathbf{u}^{n} \cdot \nabla\mathbf{v} - \frac{1}{2}p^{n}\nabla \cdot \mathbf{v} - \frac{1}{2}q\nabla \cdot \mathbf{u}^{n} d\Omega = 0 \end{aligned} \tag{5.23}$$

Equation (5.23) is now the final version which can be numerically solved using FreeFEM++.

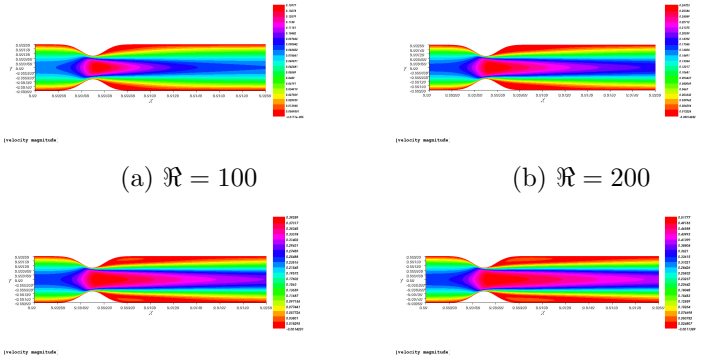
5.6 Results

We would like to investigate the behaviour of the velocity magnitude and the blood pressure for stenosis and aneurysm. As stated before, we use parabolic function $\mathbf{c}$ for the Dirichlet boundary condition at the inlet flow and it's magnitude c is determined from the relation $c = \Re\mu/\rho l$, where l is the diameter of the artery. Figures 5.5 and 5.6 show plots of the velocity magnitude and pressure respectively for four different values of Reynolds numbers for domain Ω that resembles the case of stenosis (the direction of the blood flow is from left to right). By increasing the Reynolds number, the magnitude of the velocity field increases and the pressure in the region where the artery is narrow decreases. One can see from (Fig. 5.5 (c) and (d)) that regions close to the boundary after the stenosis have lower velocity magnitudes.

From the pressure plots one can conclude that higher velocities of the blood leads to bigger pressure drops and hence the affected part of the vessel can be narrowed even more. Furthermore, the magnitude of these pressure drops is not linearly dependent of the velocity of the inlet flow.

Now we simulate two different rates of stenosis. Figure 5.7 compares the magnitudes of the velocity fields for two cases: diameter of stenosis $= 0.4l$ (on the left side) and diameter of stenosis $= 0.3l$ (on the right side). We see that in the case where the stenosis is more narrow the velocity magnitude is greater. Again there can be seen regions after the stenosis with lesser velocity magnitudes and they are much bigger in the second case.

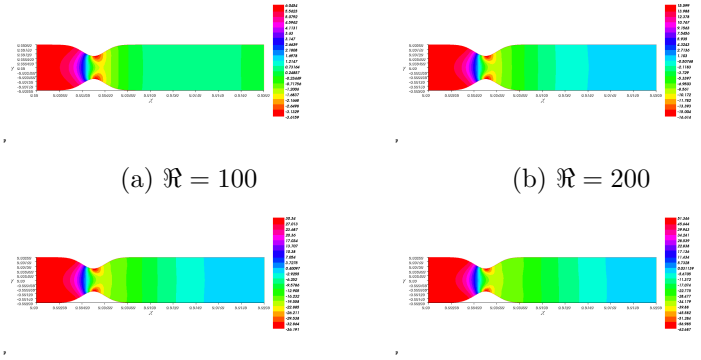
On (Fig. 5.8), the pressures for the cases from (Fig. 5.7) are compared in the same order. We see that in the second case the magnitudes of the pressure drops in the narrowed part are precisely two times more than in the first case. In the current study we do not model the artery as an elastic



(a) $\Re = 100$ (b) $\Re = 200$

(c) $\Re = 300$ (d) $\Re = 400$

Figure 5.5: Blood flow velocity magnitude for different Reynolds numbers.



(a) $\Re = 100$ (b) $\Re = 200$

(c) $\Re = 300$ (d) $\Re = 400$

Figure 5.6: Blood flow pressure for different Reynolds numbers.

body, but we can conclude that once there is a region where the artery is narrow there are pressure drops and therefore the risk of the affected part getting even more narrow.

Let us now investigate in more detail, the behaviour of the blood flow after the stenosis. In (Fig. 5.8) the x coordinate of the velocity field $\mathbf{u}$ for one of the previous cases is shown. We have previously seen that after the stenosis there are isolated regions with comparably smaller velocities. From (Fig. 5.8) one can see that, in these regions, the x coordinate of $\mathbf{u}$ is negative. This means that we observe regions where the blood flows in a direction opposite to the direction of the general blood flow. This is an indication of the formation of vortices and we can conclude that in a non-ideal case turbulent behavior of the blood flow after the stenosis can be observed.

All of the previous results are obtained fot the case of rotational sym-

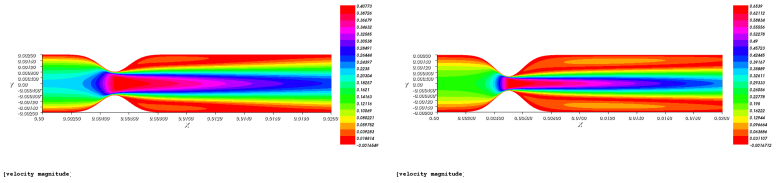


Figure 5.7: Blood flow velocity magnitude for different rates of stenosis.

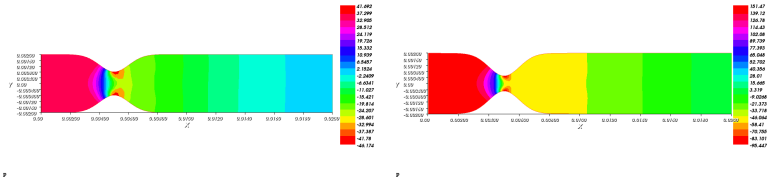


Figure 5.8: Blood flow pressure for different rates of stenosis.

metry, now let's consider the case when there is a shift between the upper and the lower narrowings. Figure 5.10 shows the blood flow velocity magnitude for a simulated non-symmetrical case of stenosis. One can see from the figure that the behaviour of the blood flow with this slight shift changes considerably. Since in reality there is always non-symmetry it is important to be able to obtain results for such cases. In all simulations for the case of stenosis the numerical algorithm based on the Newton method shows fast convergence.

Lastly, we simulate the blood flow for a domain that resembles the other pathological case we are interested in investigating - the case of aneurysm. Plot for the blood flow velocity magnitude is shown in Figure 5.11. The domain in this case consists of one boundary Γ_{In} in the lower right corner of the figure with Dirichlet boundary condition and two boundaries Γ_{Out} in the lower left and upper right corners with Neumann boundary conditions. One can see that in this domain, the blood flow has complex behaviour, however fast convergence of the numerical algorithm was again observed.

Time dependent Stokes Equation

A time varying source was placed on the lower boundary in order to simulate the pulsating blood flow into the artery. This is basically a time dependent Dirichlet boundary. The boundaries are described as follows:

$$\mathbf{n} \cdot \mathbf{T} = 0 \text{ on } \Gamma_{Out}$$
$$\mathbf{u} = A \sin^2(wt)$$

The evolution of the solution can be seen in Figure 5.12

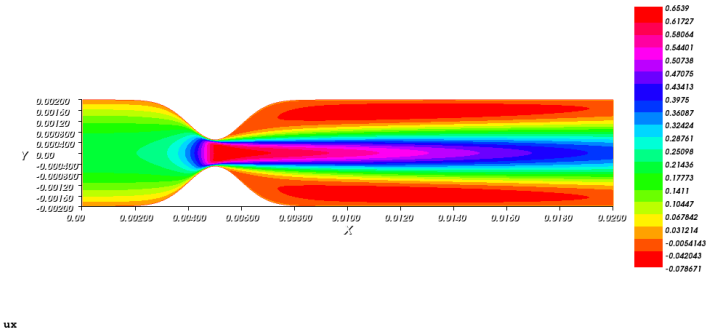


Figure 5.9: x coordinate of the velocity field.

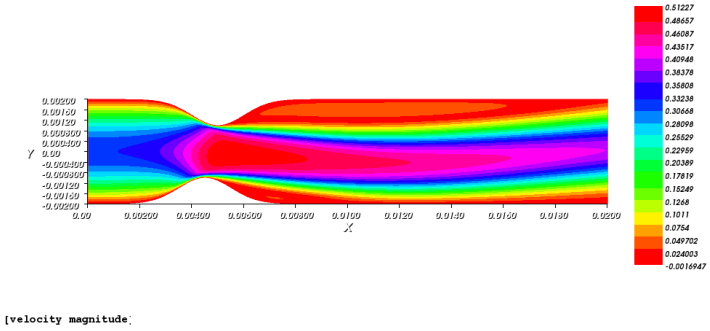


Figure 5.10: Blood flow velocity magnitude for non-symmetryc case of stenosis.

5.7 The Observation Error Minimization

Problem Motivation

In practice, patient-specific simulations of the aneurysms and stenosises can be constructed by using computational fluid dynamic techniques and image-based vascular models through segmentation of medical images from MRI (magnetic resonance imaging) or CTA (computed tomography angiography) (Fig. 5.13).

Moreover, some imaging techniques, mainly MRI-based, allow not only to obtain the geometry of the computational domain but also to measure velocity of the blood along cross-sections of the domain (Fig. 5.14). We want to use the velocity data from those extra observations to set a model and obtain a numerical simulation which can be considered reliable.

It is clear that the impact of the choice of the boundary conditions on simulation can be highly important in matters of reliability. Therefore,

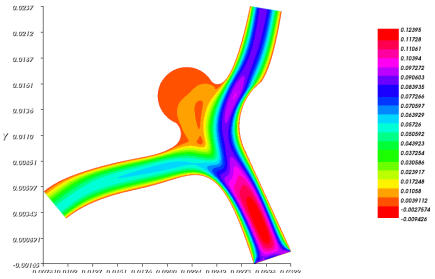


Figure 5.11: Blood flow velocity magnitude in case of aneurysm.

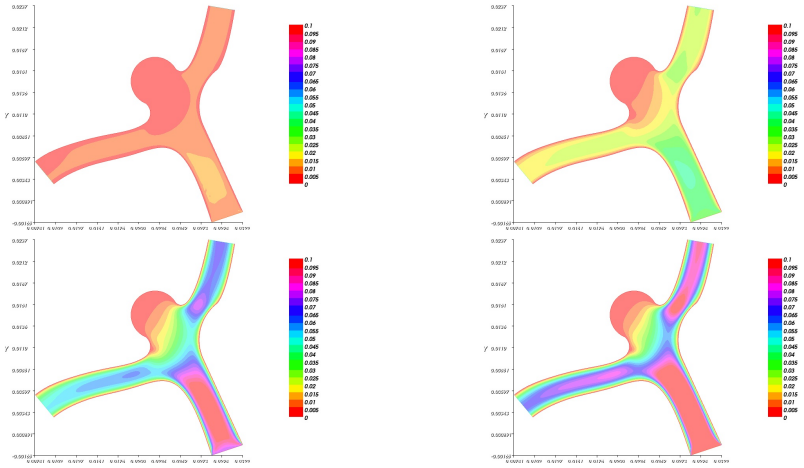


Figure 5.12: Pulsating in-flow of blood

we consider a control of the velocity values at the inflow initial Dirichlet boundary of the approximated computational domain.

The associated optimization problem aims to minimize the misfit between the given cross-sections velocity data and the computed solution, thus, reducing the uncertainty associated to the reconstructed domain.

Problem Formulation

We first simplify the problem discussed in the previous chapters from the case of Navier-Stokes equations to the case of 2D Stokes equation by cause of the limited time allotted in the context of the conference. This simplification reduces both the running time of error minimization program and implementation difficulties. However, the same theory could be applied as well to a more general model based on Navier-Stokes equations.

The weak form of the corresponding Stokes model is to find $u \in \mathbb{S}_u$ and $p \in L^2(\Omega)$, such that

$$\int_\Omega \left[\mu \nabla \mathbf{u} : \nabla \mathbf{v} \; - p \nabla \cdot \mathbf{v} \; - q \nabla \cdot \mathbf{u}\right] \, d\mathbf{x} = 0 \tag{5.24}$$

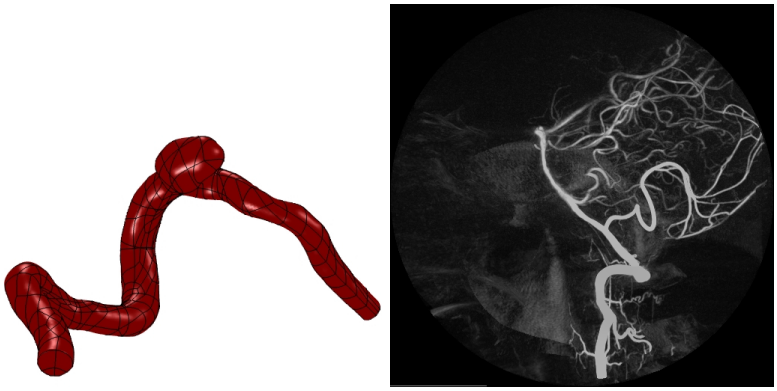


Figure 5.13: Segmentation from Medical Image. Blood through a brain artery with an aneurysm. Image from CTA.

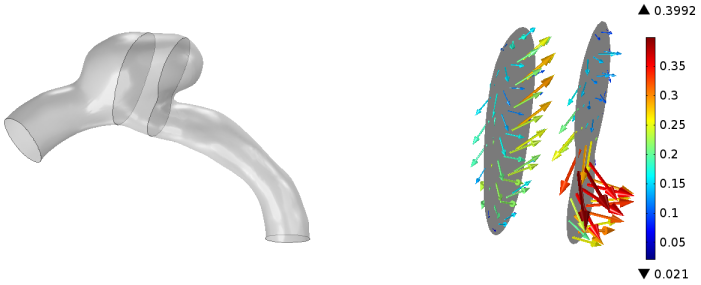


Figure 5.14: The Geometry and the velocity of the matter captured by 3D MRI.

We denote this problem (WPS).

We expect to know the velocity cross-sections data in some internal observation lines that are parallel to Γ_{In}. We denote the union of such lines as Ω_{Obs}.

Our goal is to find a vector of the initial Dirichlet boundary condition on Γ_{In} such that a given objective function J, which depends on $\mathbf{u}, p$ is minimized.

Thus, the error minimization problem can be formulated as follows:

$$\begin{array}{rl} \text{Minimize} & J(\mathbf{u}, p, \mathbf{c}) \\ \mathbf{c} \in A_{ad} & \\ \text{subject to} & (WPS) \end{array} \tag{5.25}$$

where

$\mathbf{c}$ is the control parameter, consisting of the unknown velocity profiles at Dirichlet boundary Γ_{In},

A_{ad} is the admissible function space for the control function and

J is the considered cost functional such that:

$$J(\mathbf{u}, p, \mathbf{c}) = ||\mathbf{u}(\mathbf{c}) - \mathbf{u}_{Obs}||_{L^2(\Omega_{Obs})} + \alpha||\nabla\mathbf{u}(\mathbf{c})||^2_{L^2(\Gamma_{In})} \tag{5.26}$$

The first term of the cost functional J is responsible for minimizing the misfit between the observation data and the computed solution.

The second term is a regularization term that is included in the minimization in order to ensure smoothness of the solution at the controlled boundary Γ_{In}. α is a free parameter that needs to be tuned empirically.

The reason to introduce this additional penalization information is that the problem of fitting the observation data is known to be ill-posed and generally requires some form of regularization. In this work, we use Tikhonov regularization which is essentially a trade-off between fitting the data and smoothness of the solution at the controlled boundary.

Discretization and Algorithm

To discretize the governing equations (WPS), we apply a finite element discretization. We choose a triangulation of the domain and the discretization of the functional spaces. We use linear finite elements for the velocities and the pressure and replace the functions by their approximations similarly to the case of Navier-Stokes equations. We also discretize the cost function 5.26.

The discretized optimization problem may now be written as

$$\begin{array}{rl} \text{Minimize} & \hat{J}^h(\mathbf{c}) \\ \text{subject to} & \mathbf{c} \in A_{ad} \end{array} \tag{5.27}$$

with

$$\hat{J}^h(\mathbf{c}) = J^h(\mathbf{u}^h(\mathbf{c}), p^h(\mathbf{c}), \mathbf{x}^h(\mathbf{c}), \mathbf{c}) \tag{5.28}$$

Here, the velocities and pressure, $\mathbf{u}^h(\mathbf{c})$ and $p^h(\mathbf{c})$, are implicit functions of the control vector $\mathbf{c} \in A_{ad} \subset \mathbb{R}^2$. These implicit functions are defined as the solution of the discretized version of the Stokes problem (WPS) with $\mathbf{c}$ as Dirichlet boundary condition. This is referred to as the black-box or nested analysis and design (NAND) approach.

In (5.25), the velocities and pressures as well as the control vector are optimization variables. This is referred to as the all-at-once or simultaneous analysis and design (SAND) approach.

SAND formulations combined with sequential quadratic programming (SQP) methods is a very good approach, however, they require more implementation efforts than the gradient-free method CMAES that we use to solve the error minimization problem (5.27).

CMAES (Covariance Matrix Adaptation Evolution Strategy) is an a derivative free optimizer implementing an evolutionary algorithm. This algorithm works with a normal multivariate distribution in the parameters

space and try to adapt its covariance matrix using the information provided by the successive function evaluations. Although gradient-based methods are usually faster than the CMAES, we have chosen the latter because it is a convenient tool provided by FreeFem++.

On the whole, the algorithm to perform our optimization problem is given as following steps:

Algorithm 1: Blood flow modeling with observation error minimization

1: Get the observation data $\mathbf{u}_{obs}$ (from a simulation).
2: Initialize $\mathbf{c}$ randomly on Dirichlet boundary Γ_{In}.
3: While stopping criteria not satisfied:
   (a) Solve no-stress Stokes problem (WPS) with $\mathbf{c}$ fixed as a Dirichlet boundary condition.
   (b) Evaluate the cost functional $J(\mathbf{c})$, measuring the misfit between the solution obtained above and observation data.
   (c) Perform an optimization step by updating $\mathbf{c}$. CMAES optimizer is used in the current work.

For the purpose of simplifying implementation, we performed the error minimization analyses by assuming the considered part of cardiovascular system to be rectangular.

Observation Data

In the context of the current work, the real observation data obtained from MRI were not available, instead, we simulated the velocity data on observation lines by running the solver of the Stokes problem (WPS) with an inclined parabola as an initial Dirichlet boundary condition $\mathbf{c}$.

The formula for this parabola is given by:

$$u|_{\Gamma_{In}} = \left(-\frac{1}{2}(-y^2 + l^2)\cos\left(\frac{\pi}{6}\right);\ \ \frac{1}{2}(-y^2 + l^2)\sin\left(\frac{\pi}{6}\right) \right)$$

with l – rectangular domain height.

The result of this simulation at observation lines (the solution $\mathbf{u}|_{\Omega(Obs)}$) was afterwards saved and referred as the observation data. This procedure provides the first step of the Algorithm (Alg. 1). The magnitude plot of the obtained simulated data is shown on the (Fig. 5.15).

Although data sets obtained by real medical imaging technique often contain noise and artifacts caused by signal degradation, those potential errors are out of the scope of the current work.

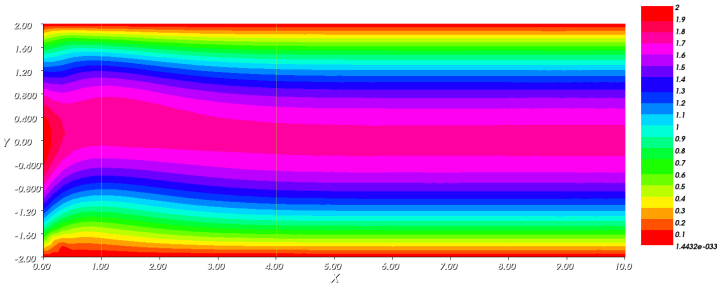


Figure 5.15: The simulated solution reffered as the observation data.

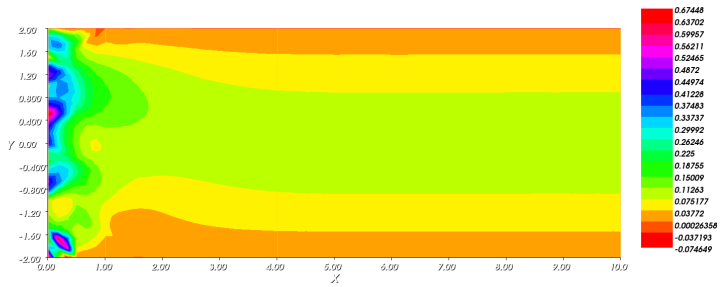


Figure 5.16: The solution obtained with an initial randomized Dirichlet boundary condition **c**.

Optimization Procedure and Results

The optimization procedure is started by a random initialization of the Dirichlet boundary control parameter **c**, which is the second step of the Algorithm (Alg. 1). The result of running the solver of the Stokes problem with such an initialization is shown in (Fig. 5.16).

In order to evaluate the minimal error solution obtained from a proposed optimal control algorithm, we compare this solution with a simulated data referred here as a ground truth solution $\mathbf{u}_{Sim}$.

For a numerical evaluation, three types of errors are introduced:

- Error over the whole domain:

$$\epsilon_{\Omega} = \frac{||\mathbf{u}_{Sim} - \mathbf{u}||_{L^2(\Omega)}}{||\mathbf{u}_{Sim}||_{L^2(\Omega)}}$$

- Error on the inflow (control) boundary

$$\epsilon_{\Gamma_{In}} = \frac{||\mathbf{u}_{Sim} - \mathbf{u}||_{L^2(\Gamma_{In})}}{||\mathbf{u}_{Sim}||_{L^2(\Gamma_{In})}}$$

- Error on the observation lines

$$\epsilon_{\Omega_{Obs}} = \frac{\sum_i ||\mathbf{u}_{Sim} - \mathbf{u}||_{L^2(\Gamma_{Obs_i})}}{\sum_i ||\mathbf{u}_{Sim}||_{L^2(\Gamma_{Obs_i})}}, \quad \text{where } \cup_i \Gamma_{Obs_i} = \Omega_{Obs}$$

As it was mentioned above, the free parameter α, responsible for the weight of the regularization term, is needed to be tuned empirically. The influence of the choice of α on the error values is demonstrated in the Table below (Fig. 5.17).

	ϵ_Ω	$\epsilon_{\Gamma_{In}}$	$\epsilon_{\Omega_{Obs}}$
$\alpha = 0$	2.94	0.125	0.005
$\alpha = 10^{-3}$	0.43	0.034	0.004
$\boldsymbol{\alpha = 10^{-2}}$	**0.286**	**0.050**	**0.002**
$\alpha = 10^{-1}$	2.072	0.616	0.058

Figure 5.17: Error values with different reguarization weight parameter α. Number of iterations 500 for all the cases. Number of fitness evalutions 7000.

In the case of small α or $\alpha = 0$ the solution at the inflow boundary is not smooth enough and, even though the error on the observation lines may be small, the error over the whole domain shows that the computed optimized solution is far away from the simulated solution. The velocity magnitude plots for $\alpha = 0$ and $\alpha = 10^{-3}$ are shown on the (Fig. 5.18) and (Fig. 5.19). Empirically obtained optimal regularization term weight is $\alpha = 10^{-2}$ which is confirmed by the smallest errors, producing the following values: $\epsilon_\Omega = 0.286$, $\epsilon_{\Gamma_{In}} = 0.050$, $\epsilon_{\Omega_{Obs}=0.002}$ The velocity magnitude plots for $\alpha = 10^{-1}$ and $\alpha = 10^{-2}$ are shown on the (Fig. 5.20) and (Fig. 5.21).

5.8 Conclusions

In the current work we have presented a 2D mathematical model based on the Navier-Stokes equations for steady state blood flow inside a segment of an artery. We proposed an iterative numerical method for obtaining a solution of the non-linear problem, based on the Newton method. Furthermore we have investigated the behavior of the blood flow for simulated cases of the two most common cardiovascular diseases - aneurysm and stenosis. We have shown that the numerical method we use allows us to obtain results for non-symmetric cases resembling the case of stenosis and also for complex domain that resembles the case of aneurysm. Also we have formulated a minimization problem for the recovery of boundary conditions from given cross-sections measurements. However, due to time limitations the optimization problem was based on the simplified case of a Stokes flow. A

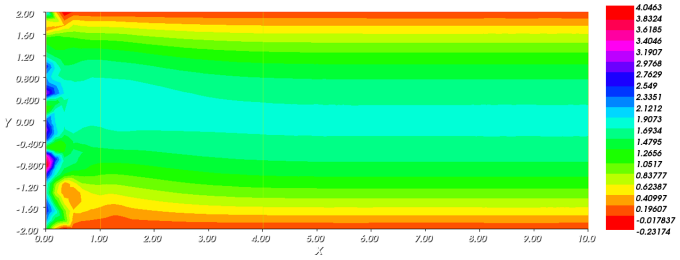


Figure 5.18: Minimization with no regularization term imposed ($\alpha = 0$).

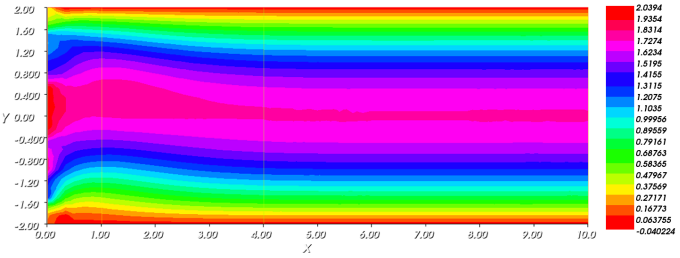


Figure 5.19: Minimization with small-weight regularization term ($\alpha = 10^{-3}$).

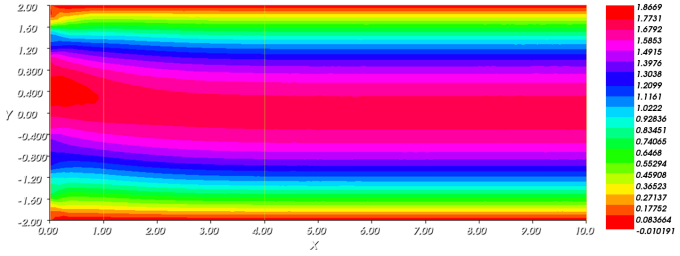


Figure 5.20: Minimization with big-weight regularization term ($\alpha = 0.1$).

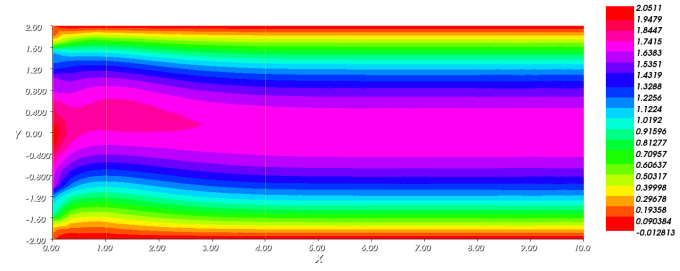


Figure 5.21: Minimization with $\alpha = 0.01$.

numerical algorithm for the minimization problem was also proposed and simulations were performed. Results from the carried simulations led us to a conclusion that the minimization algorithm allows us to recover simulated data with a sufficient error from only two cross-section measurements. For future work one should consider a 3D model for the blood flow without the assumption of the blood being a Newtonian fluid and a minimization problem for Navier-Stokes flow.

Bibliography

[1] Abraham, F and Behr, M. Shape optimization in steady blood flow: A numerical study of non-newtonian effects. In *Computer Methods in Biomechanics and Biomedical Engineering, Vol.8*, 127–137. 2005.

[2] author, U. Tikhonov regularization. https://en.wikipedia.org/wiki/Tikhonov_regularization.

[3] Formaggia, L; Quarteroni, A and Veneziani, A. *Cardiovascular Mathematics: Modeling and simulation of the circulatory system*, volume 1. Springer Science & Business Media, 2010.

[4] Hansen, N. The cma evolution strategy: A tutorial 2011.

[5] Hecht, F. New development in freefem++. *J Numer Math* **20**(3-4), 251–265, 2012. ISSN 1570-2820.

[6] Tiago, J; Gambaruto, A and Sequeira, A. Patient-specific blood flow simulations: setting dirichlet boundary conditions for minimal error with respect to measured data. *Mathematical Modelling of Natural Phenomena* **9**(06), 98–116, 2014.

.1 Code

Navier-Stokes Stenosis.edp

```
1 // Stationary incompressible Navier-Stokes Equation
    with Newton method.
2
3 // build the  Mesh
4 real L = 0.020, l = 0.004;
5 real l2=l/2;
6
7 real ag =0.0010 , bg=L/4 , cg=0.001, shiftparam = 0*
    L/40;
8 real numb = 40;
9
```

```
border gamma1(tt=0,L) { x=tt ; y=-l2+ag*exp(  - ( (
  (tt-L/2 + bg + shiftparam)^2 )/(2*cg^2) )  ); label
  =1; }
border gamma4(tt=-l2,l2) { x=L ; y=tt; label=3; }
border gamma2(tt=L,0) { x=tt ; y=l2-ag*exp(  - ( ( (
  tt-L/2 + bg)^2 )/(2*cg^2) )   ); label=1; }
border gamma3(tt=l2,-l2) { x=0 ; y=tt; label=2; }

mesh Th=buildmesh(gamma1(numb*3)+gamma2(numb*3)+
  gamma3(numb)+gamma4(numb));
plot(Th);


// macro operators
macro Grad(u1,u2) [ dx(u1),dy(u1), dx(u2),dy(u2)]//
macro UgradV(u1,u2,v1,v2) [ [u1,u2]'*[dx(v1),dy(v1)]
   , [u1,u2]'*[dx(v2),dy(v2)] ]//
macro div(u1,u2)  (dx(u1)+dy(u2))//


//  FE Space
fespace Xh(Th,P2);fespace Mh(Th,P1);
Xh u1,u2,v1,v2,du1,du2,u1p,u2p;
Mh p,q,dp,pp;

Xh psi, phi;

real c ; // velocity of in flow
Xh vel, vor;

// Physical properties
real mu = 0.0035, rho = 1060;
real Vmean;
real Re = 200;
c = Re*mu/(rho*l);

// Initial guess for (u1, u2) with B.C.
solve Stokes ([u1,u2,p],[v1,v2,q],solver=UMFPACK) =
    int2d(Th)(  mu*(Grad(u1,u2)'*Grad(v1,v2) )
              - div(u1,u2)*q - div(v1,v2)*p
          )
  + on(1,u1=0,u2=0)
  + on(2,u1=(-c*(y^2)+c*(l2^2))/(l2^2),u2=0);

//plot([u1,u2] );


solve streamlines(psi,phi) =
      int2d(Th)( dx(psi)*dx(phi) + dy(psi)*dy(phi))
```

```
    +  int2d(Th)( -phi*(dy(u1)-dx(u2)))
//    +  on(1,psi=0)
//   +  on(2,psi=0)
//   +  on(3,psi=0)
;

plot(psi,wait=1);

vel = sqrt(u1^2+u2^2);
plot(vel, fill=1, value=1);
plot([u1,u2]);


// stop test for Newton
real eps=1e-6;

verbosity = 2;

int n;
real err=0;
for( n=0;n<20;n++) // Newton Loop
{
  solve Oseen([du1,du2,dp],[v1,v2,q]) =
        int2d(Th) (  mu*(Grad(du1,du2)'*Grad(v1,v2))
          )
                  + rho*UgradV(du1,du2, u1, u2)'*[v1,v2]
                  + rho*UgradV( u1, u2,du1,du2)'*[v1,v2]
                  - div(du1,du2)*q - div(v1,v2)*dp
                  - 1e-8*dp*q // stabilization term
                  )
      - int2d(Th) (  mu*(Grad(u1,u2)'*Grad(v1,v2) )
                  + rho*UgradV(u1,u2, u1, u2)'*[v1,v2]
                  - div(u1,u2)*q - div(v1,v2)*p
                  - 1e-8*p*q
                  )
      + on(1,du1=0,du2=0)
      + on(2, du1=0, du2=0)
  ;

  u1[] -= du1[];
  u2[] -= du2[];
  p[]  -= dp[];
  real Lu1=u1[].linfty,  Lu2 = u2[].linfty , Lp = p[].linfty;
  err= du1[].linfty/Lu1 + du2[].linfty/Lu2 + dp[].linfty/Lp;
```

```
97
98    cout << n << " err = " << err << endl;
99    if(err < eps) break; // converge
100   if( n>3 && err > 10.) {
101     cout << " not converge " << endl;
102     break;
103   } //  Blowup ?
104
105   //  plot([u1,u2]);
106 }
107
108 streamlines;
109 plot(cmm="streamlines", psi,wait=1);
110
111 vel = sqrt(u1^2+u2^2);
112 plot(cmm=" [velocity magnitude]", vel, fill=1, value
    =1);
113 //plot([u1,u2]);
114 plot (cmm="p" ,p, fill=1, value=1);
115
116 vor = abs(dx(u2)-dy(u1));
117 plot (cmm="vortex" ,vor, fill=1, value=1);
118
119 Xh u1n = u1 / vel;
120 Xh u2n = u2 / vel;
121
122 plot(cmm=" [u1n,u2n]", [u1n,u2n]);
```

Time-dependent Stokes equation.edp

```
1 border f1(t=8.277579470048165,3.122414148982325){x=t
    ;y=0.002358371815112*t^5 -
2 0.065655720087747*t^4 + 0.729627361800546*t^3 -
    4.08197654392561*t^2 + 11.856566776041069*t -
3 11.709295757087336; label=1;}
4 border f2(t=0.859452741468984,-0.981740566055334){x=
     (0.341400771151288^(1./2.))*cos(t) +
    7.952959033840158; y=
5 (0.341400771151288^(1./2.))*sin(t)
    +4.461148557368587; label=1;}
6 border f3(t=0.529128582979014,2*pi
    -1.952566680840888) {x=((1.281091616344505)^(1./2.)
    )*cos(t)+8.756105018206105; y =
7 ((1.281091616344505)^(1./2.))*sin(t) +
    5.954110697006632; label=1;}
8 border f5(t=12.880766163863337,11.008092762240402){x
    =t;y= -2.1851851851852*t + 27.97381491146139; label
```

```
   =1;}
9  border f6(t=4,11.41942696370471){x=t;y
   =0.000141915629507*t^7 - 0.007078659134381*t^6 +
   0.14697826936389*t^5 -
10 1.653476604852379*t^4 + 10.936743462092085*t^3 -
   42.81088073494685*t^2 +
11 92.93661032718971*t - 86.13879673351471; label=1;}
12 border f7(t=11.265838454747,9.733174187725922){x=t;
   y= 1.863904540127596*t^2 -
13 37.203740366653165*t + 192.0595346933998; label=1;}
14 border f8(t=11.688120335844724,12.5) {x=t; y=
   0.818080357142857*t^3 - 26.75577731092437*t^2 +
15 291.4957983193277*t - 1051.6302521008404; label=1;}
16 border f9(t=2*pi
   -2.739677193250460,2.225248502466470) {x=
   (4.763099785545185^(1./2.))*cos(t) +
17 13.016633203294978; y= (4.763099785545185^(1./2.))*
   sin(t)+4.772829646050528; label=1;}
18 border g1(t=3.122414148982325,4){x=t;y
   =-1.200267257468735*t + 5.932898265695426;}
19 border g2(t=11.41942696370471,12.880766163863337){x=
   t;y = 0.332421039275865*t - 4.454882158508418;label
   =2;}
20 border g4(t=12.5, 11.265838454747){x=t; y=
   -0.164567526760309*t + 11.347313687969192;}
21
22
23 plot(f1(50)+f2(50)+f3(50)+f5(50)+f6(50)+f7(50)+f8
   (50)+g1(50)+g2(50)+g4(50)+f9(50));
24
25
26 mesh Th1=buildmesh(f1(50)+f2(10)+f3(50)+f5(30)+f6
   (70)+f7(30)+f8(35)+f9(25)+g1(20)+g2(20)+g4(20));
27
28 mesh Th=movemesh(Th1,[0.001*2.5*x,0.001*2.5*y]);
29
30 plot(Th);
31
32
33 //macro operators
34 macro Grad(u1,u2) [dx(u1),dy(u1), dx(u2),dy(u2)]//
35 macro UgradV(u1,u2,v1,v2) [ [u1,u2]'*[dx(v1),dy(v1)]
     , [u1,u2]'*[dx(v2),dy(v2)] ]//
36 macro div(u1,u2)  (dx(u1)+dy(u2))//
37
38 //macro Gradn(u1n,u2n) [ dx(u1n),dy(u1n), dx(u2n),dy
   (u2n)]//
39 //macro UgradVn(u1n,u2n,v1,v2) [ [u1n,u2n]'*[dx(v1),
   dy(v1)] , [u1n,u2n]'*[dx(v2),dy(v2)] ]//
```

```
//macro divn(u1n,u2n)  (dx(u1n)+dy(u2n))//

//  FE Space
fespace Xh(Th,P2);fespace Mh(Th,P1);
Xh u1,u2,v1,v2,du1,du2,u1p,u2p;
Mh p,q,dp,pp;

Xh vel;

real c; // velocity of in flow

//real tau=0.01; //time

// Physical properties
real mu = 0.0035, rho = 1060;
real Vmean;
real Re = 100;
real Lc=0.003425;
c = Re*mu/(rho*Lc);
cout << c << endl;

real t;
real T=2;
real n=200;
real tau=T/n;

real[int]viso(2);
for(int ii=0;ii<viso.n;ii++)
  viso[ii]=ii*0.1;


Xh u1o=0,u2o=0,po=0,u1plot,u2plot;
problem Stokes ([u1,u2,p],[v1,v2,q],solver=UMFPACK)
  =
    int2d(Th)( (rho/tau)*[u1,u2]'*[v1,v2]+0.5*( mu*(
      Grad(u1,u2)'*Grad(v1,v2) )
                  - div(u1,u2)*q - div(v1,v2)*p))+
       int2d(Th)( -(rho/tau)*[u1o,u2o]'*[v1,v2
         ]+0.5*( mu*(Grad(u1o,u2o)'*Grad(v1,v2) )
                  - div(u1o,u2o)*q - div(v1,v2)*po)
                 )

    + on(1,u1=0,u2=0)
  //+ on(2,u1=0,u2=0);
  //+ on(2,u1=-c*(N.x),u2=-c*(N.y));
  + on(2,u1=-c*(N.x)*sin(10*t)*sin(10*t),u2=-c*(N.y)
    *sin(10*t)*sin(10*t));


```

```
t=0;
for (int iii = 0; iii < n; iii++){
  t+=tau;
  cout << "t" << t << endl;

  u1o=u1;
  u2o=u2;
  po=p;

  Stokes;
  vel = sqrt(u1^2+u2^2);

  u1plot=u1/70;
  u2plot=u2/70;


  //plot([u1plot,u2plot],ps="Magnetude"+iii+".jpg");
  plot([u1plot,u2plot]);
  //plot(vel, fill=1, value=1,ps="Plot"+iii+".jpg",
    viso=viso(0:viso.n-1));
  //plot(vel,fill=1,value=1,ps="Plot.jpg", viso=viso
    (0:viso.n-1));
}
```

Optimization.edp

```
// Observation Error Minimization for Stationnary
   imcompressible Stokes Equation with Newton method.

// ----------------------------------------
// I. Load modules:
// ----------------------------------------

load "ff-cmaes"

// ----------------------------------------
// II. Global variables:
// ----------------------------------------

// Mesh properties
real L = 10., l = 4.;
real l2=l/2;
mesh Th;

//  FE Space
fespace Xh(Th,P2);
fespace Mh(Th,P1);
```

```
Xh u1,u2,v1,v2,du1,du2,u1p,u2p;
Mh p,q,dp,pp;
Xh boundaryC1,boundaryC2;

// Physical properties
real mu = 0.8, rho = 0.5;
real Vmean = 1.;
real Re = 100;

// Simulation and Observation properties and variables (for optimization part)
real c = 0.5; // velocity of in flow
real alpha = 0.01; // weight of the regularization term
int numObs=2;

// Arrays numNodesInflowBoundary, nodesObsLines, etc. don't have a lot of physical/mathematical sence,
// they are introduced only to read/compare observations at the observation lines
int       numNodesInflowBoundary; // number of dots on the inflow boundary line
int[int]   nodesInflowBoundary(0); // to get indices of inflow boundary dofs
int[int]    numNodesObsLines(numObs); // (cnumber) number of dots on the observation lines
int[int,int] nodesObsLines(numObs, 0); // (cMatrix) // to get indices of observation lines dofs

Xh u1Sim, u2Sim; // stores the data from observation lines
real [int, int ] uObsMat1 (numObs,2000); // another format / stores the data from observation lines (x component)
real [int, int ] uObsMat2   (numObs,2000); // another format / stores the data from observation lines (y component)

real[int] control1, control2;

// Plotting properties
int itNum = 0;
int plotEvery = 5000; // intermediate plots

// ----------------------------------------
// III. Mesh
// ----------------------------------------

// build the  Mesh
```

```
57 border gamma1(tt=0,L) { x=tt ; y=-l/2; label=1; }
58 border gamma4(tt=-l2,l2) { x=L ; y=tt; label=3; }
59 border gamma2(tt=L,0) { x=tt ; y=l/2; label=1; }
60 border gamma3(tt=l2,-l2) { x=0 ; y=tt; label=2; }
61 border gamma6(tt=l2,-l2){ x=4 ; y=tt; label=5;}
62 border gamma5(tt=l2,-l2){ x=1 ; y=tt; label=4;}
63
64 int[int] labels = [2,4,5];
65
66 Th=buildmesh(gamma1(L*2)+gamma2(L*2)+gamma3(l*2)+
     gamma4(l*2)+ gamma5(l*2)+gamma6(l*2));
67 plot(Th);
68
69 // ---------------------------------------
70
71 Xh[int] xh (2);
72 xh[0] = x;
73 xh[1] = y;
74 int n = Xh.ndof; // velocity dfom
75
76 // ---------------------------------------
77 // IV. Auxilary functions
78 //     (GO TO to the next chapter to skip the details
     of implementation)
79 // ---------------------------------------
80
81 // ---------------------------------------
82 // IV.a Standard macro operators
83 // ---------------------------------------
84
85 macro Grad(u1,u2) [ dx(u1),dy(u1), dx(u2),dy(u2)]//
86 macro UgradV(u1,u2,v1,v2) [ [u1,u2]'*[dx(v1),dy(v1)]
     , [u1,u2]'*[dx(v2),dy(v2)] ]//
87 macro div(u1,u2)  (dx(u1)+dy(u2))//
88
89 // ---------------------------------------
90 // IV.b Functions and macros needed
91 // ---------------------------------------
92
93 // the function return the indeces of nodes located
94 // at the border with a label 'label'
95 func int getNodesOnLine(int iLine)
96 {
97   int label = labels(iLine);
98
99   // to generate a vector with 1 on boundary dofs
100  varf von(u,v)= on(label,u=1);
101  real[int] on1=von(0,Xh, tgv=1); // one on boundary
       dofs
```

```
102
103   int numNodesOnBorder = on1.sum; // number of nodes
        on the line 'label'
104   int[int] nodesOnBorder(numNodesOnBorder);
105
106   int boundaryCounter = 0;
107   for (int i=0;i<n;++i)
108   {
109    if( on1[i])
110     {
111       nodesOnBorder[boundaryCounter]=i;
112       boundaryCounter++;
113     }
114   }
115
116   cout << "# nodes " << Xh.ndof << "; "
117      << "# nodes on the border " << numNodesOnBorder
           << endl;
118   cout << "Indeces and positions of nodes at the
        observation line "
119      << iLine << ": " << endl;
120   for (int n = 0; n < numNodesOnBorder; n++) {
121     cout << nodesOnBorder[n];
122     for (int d = 0; d < 2; d++)
123       cout << "  " << xh[d][][nodesOnBorder[n]]; //
            x, y
124     cout << endl;
125   }
126
127   // ---------------------------------------
128   // save the results to global arrays
129
130   // Inflow boundary
131   if (iLine == 0) {
132     nodesInflowBoundary.resize(numNodesOnBorder);
133     nodesInflowBoundary = nodesOnBorder;
134     numNodesInflowBoundary = numNodesOnBorder;
135   }
136
137   // Observation lines
138   if (iLine >= 1) {
139     if (nodesObsLines.m < numNodesOnBorder)
140       nodesObsLines.resize (numObs, numNodesOnBorder
            );
141     nodesObsLines(iLine-1,:) = nodesOnBorder;
142     numNodesObsLines(iLine-1) = numNodesOnBorder;
143   }
144
145   return 0;
```

```
146 }
147
148 // solve Stokes equation
149 macro solvePDE(control)
150 {
151   convertControlToBoundary(control); /*changes
      boundaryC1,boundaryC2*/
152
153   /* Initial guess for (u1, u2) with B.C.*/
154   solve Stokes ([u1,u2,p],[v1,v2,q],solver=UMFPACK)
       =
155       int2d(Th)(
156         (1/Re)*(Grad(u1,u2)''*Grad(v1,v2) )
157          - div(u1,u2)*q - div(v1,v2)*p
158            )
159     + on(1,u1=0,u2=0)
160       + on(2,u1=boundaryC1,u2=boundaryC2);
161
162   /* change u1 and u2 on control */
163 }[u1,u2]//
164
165 // the function convert a vector control into
166 // compatible for the current program format of
    boundary
167 macro convertControlToBoundary(control)
168 {
169   splitControlVectorIntoTwoVectors(control, control1
      , control2);
170   for (int n = 0; n < numNodesInflowBoundary; n++) {
171       boundaryC1[][nodesInflowBoundary[n]] =
          control1[n];
172           boundaryC2[][nodesInflowBoundary[n]] =
              control2[n];
173   }
174 }[boundaryC1, boundaryC2]//
175
176 // auxiliary function for convertControlToBoundary(
    control)
177 func int splitControlVectorIntoTwoVectors(
178   /* input  */  real[int] & control,
179   /* output */  real[int] & control1, real[int] &
      control2)
180 {
181   int obsLength = control.n/2;
182   control1.resize(obsLength);
183   control2.resize(obsLength);
184
185   for (int i = 0; i < obsLength; i++)
186   {
```

```
187     control1[i] = control[i];
188     control2[i] = control[i + obsLength];
189   }
190
191   return 0;
192 }
193
194 // plot the current solution u
195 func int plotSolution()
196 {
197   Xh mag=sqrt(u1^2+u2^2); // for a magnitude plot
198   plot([u1,u2],value=1);
199   plot(mag,value=1, fill=1);
200   return 0;
201 }
202
203 // run the simulation with an inclined parabola at
    the inflow border
204 func int runSimulation() {
205
206   real[int] cInitX(numNodesInflowBoundary);
207   real[int] cInitY(numNodesInflowBoundary);
208   real[int] cInit2(2*numNodesInflowBoundary);
209
210   real xx, yy;
211   for (int n = 0; n < numNodesInflowBoundary; n++) {
212         yy = xh[1][][nodesInflowBoundary[n]];
213       //xx = xh[0][][nodesInflowBoundary[n]];
214
215     cInitX(n) = (-c*(yy^2)+c*(l2^2))*cos(-pi/6);
216         cInitY(n) = -(-c*(yy^2)+c*(l2^2))*sin(-pi/6)
            ;
217
218     cInit2(n) = cInitX(n);
219     cInit2(n+numNodesInflowBoundary) = cInitY(n);
220   }
221
222   solvePDE(cInit2);
223   plotSolution();
224
225   return 0;
226 }
227
228 // save the results of simulation as observation
229 func int getObservationsFromSimulation()
230 {
231   real yy;
232     for (int k = 0; k<numObs; k++){
233     for (int j = 0; j < numNodesObsLines[k]; j++) {
```

```
234         uObsMat1(k,j)= u1[][nodesObsLines(k,j)];
235         uObsMat2(k,j)= u2[][nodesObsLines(k,j)];
236       }
237     }
238
239     u1Sim = u1;
240     u2Sim = u2;
241     return 0;
242 }//[u1Sim,u2Sim,uObsMat1,uObsMat2]
243
244 // compute errors on the whole domain, inflow and on
      observation lines
245 func real[int] computeError() {
246     real[int] error(3);
247
248     error(0) = int2d(Th)
249         (
250         sqrt(
251             ( u1Sim - u1 )^2
252           + ( u2Sim - u2 )^2
253           )
254         /
255         sqrt(
256             ( u1Sim )^2
257           + ( u2Sim )^2
258           )
259         );
260
261     error(1) = int1d(Th,gamma3)
262         (
263         sqrt(
264             ( u1Sim - u1 )^2
265           + ( u2Sim - u2 )^2
266           )
267         /
268         sqrt(
269             ( u1Sim )^2
270           + ( u2Sim )^2
271           )
272         );
273
274     error(2) = int1d(Th,gamma5)
275         (
276         sqrt(
277             ( u1Sim - u1 )^2
278           + ( u2Sim - u2 )^2
279           )
280         /
281         sqrt(
```

```
            ( u1Sim )^2
          + ( u2Sim )^2
          )
        )
           + int1d(Th,gamma6)
        (
        sqrt(
            ( u1Sim - u1 )^2
          + ( u2Sim - u2 )^2
          )
        /
        sqrt(
            ( u1Sim )^2
          + ( u2Sim )^2
          )
        );
  return error;
}

func real[int] setInitialControlToZero() {
  itNum = 0;
  real[int] cInit(2*numNodesInflowBoundary);
  for (int n = 0; n < numNodesInflowBoundary; n++) {
    cInit(n) = 0;
    cInit(n+numNodesInflowBoundary) = 0;
  }
  return cInit;
}

// returns J = ||u_Obs - u|| + alpha * || grad(u)|_c
   ||^2
func real evaluateCostFunctionalJ() {

  real cost=0;

  Xh du1x = dx(u1);
  Xh du1y = dy(u1);
  Xh du2x = dx(u2);
  Xh du2y = dy(u2);

  for (int k = 0; k < numObs; k++)
  {
     for (int j=0;j<numNodesObsLines[k];j++)
   {
    cost +=
      abs(
        sqrt(
          (uObsMat1(k,j) - u1[][nodesObsLines(k,j)])
            ^2
```

```
329           + (uObsMat2(k,j) - u2[][nodesObsLines(k,j)])
                ^2
330           )
331         )
332         + alpha *
333         abs(
334             (du1x[][nodesInflowBoundary[j]])^2
335           + (du1y[][nodesInflowBoundary[j]])^2
336           +
337             (du2x[][nodesInflowBoundary[j]])^2
338           + (du2y[][nodesInflowBoundary[j]])^2
339         )
340       ;
341     }
342   }
343
344   return cost;
345 }
346
347 // ----------------------------------------
348 // V. Cost Function
349 // ----------------------------------------
350
351 func real J(real[int] & control)
352 {
353   // ----------------------------------------
354
355   // solves PDE (Stokes here) with the inflow
        boundary condition 'control'
356   solvePDE(control);
357   // evaluate the cost functional J using
358   //     * the solution obtained by solvePDE(control
        ) above
359   //   and * the observation data (obtained from a
        simulation here)
360   real cost = evaluateCostFunctionalJ();
361
362   // ----------------------------------------
363   // print and plot:
364
365   cout << "J ="<< cost << " control =" << control1
        (0) << " " << control1(1) << "...\n" ;
366   if (itNum % plotEvery == 0) plotSolution();
367   itNum++;
368
369   // ----------------------------------------
370
371   return cost;
372 }
```

```
373
374 // ----------------------------------------
375 // VI. Main program
376 // ----------------------------------------
377
378 // ----------------------------------------
379 // VI.0) Get indeces of nodes at observation lines
    and the inflow border.
380 //     It is a technical solution with no
    mathematical sence in it.
381
382   for (int i=0 ; i<=numObs; i++){
383     getNodesOnLine(i);
384   }
385
386 // ----------------------------------------
387 // VI.1) SIMULATION
388
389   runSimulation();
390   getObservationsFromSimulation();
391
392 // ----------------------------------------
393 // VI.2) OPTIMIZATION
394
395   // We start from cInit (the control, initial flow)
       being a rendom vector around 0
396   real[int] cInit = setInitialControlToZero();
397   // The function cmaes performs optimization given
398   //     * a cost function J (solvePDE with cInit +
      compute the cost functional)
399   // and * a control vector cInit
400   real min = cmaes(J,cInit,stopTolFun=1e-3,
      stopMaxIter=500,seed=1);
401
402 // ----------------------------------------
403 // VI.3) RESULTS
404
405   cout << "minimal value is J = " << min << endl;
406   plotSolution();
407   real[int] error = computeError();
408   cout << "Error whole domain: " << error(0) << endl
      ;
409   cout << "Error inflow boundary: " << error(1) <<
      endl;
410   cout << "Error observation lines: " << error(2) <<
       endl;
```