A non-Newtonian model for computational fluid dynamics simulations of blood flows

Project report for

Advanced Programming for Scientific Computing

A.Y. 2021 - 2022

Professor: Luca Formaggia

Supervisors: Africa P. C., Fedele M., Fumagalli I., Regazzoni F.

Michele Precuzzi

Luca Caivano

POLITECNICO
MILANO 1863

Contents

1. Introduction

As medicine progresses, computational fluid dynamics simulations of blood flow are becoming of fundamental importance in the understanding of cardiovascular diseases.
Blood is composed of plasma, which consists mostly of water, and of blood cells whose diameter is approximately 10^{-3} cm, whereas that of most arteries and veins is about 10^{-1} cm. This is why blood in the systemic and pulmonary circulation is often considered to be Newtonian, that is, the shear stress is proportional to the rate of shear and the viscosity is the constant of proportionality. However in the smallest arteries, such as coronary arteries, where the dimension of the blood cells becomes more relevant with respect to the dimension of the vessel, non-Newtonian blood rheology is more appropriate ([19], [13]) and becomes of fundamental importance in small vessels like the capillaries ([10], ([16]). When treated as a Non-Newtonian fluid, the viscosity of blood varies with shear rate and this increases the complexity of the numerical model.
In addition, when one wants to perform a CFD simulation, the employment of appropriate boundary condition is of fundamental importance. For a pulsatile flow in a cylindrical domain, it is well known that the velocity profile in a section is given by the so called "Womersley profile", described by J.R. Womersley in his work [24] . Often, in this kind of simulations, medical images are used to provide boundary condition data. Doppler ultrasound may be used to estimate the flow rates at different times ([9]). Most of inflow Dirichlet boundary condition are imposed reconstructing a velocity profile starting from those flow rates measurements. For vessels with a large diameter (e.g. the ascending aorta) a uniform profile velocity imposition is considered a good approximation, and for vessels with a very small diameter (e.g. capillaries) a parabolic profile is often used ([19]): those are indeed limit cases of the Womersley profile, respectively for big and small values of the Womerlsey number. However, for intermediate values that can be found in small arteries or veins, these approximations are not appropriate and so the Womersley profile has to be computed and imposed in order to better represent the physiological boundary conditions.
We developed our project in the framework of life^x: a high-performance Finite Element library mainly focused on mathematical models and numerical methods for cardiac applications. Our aim was to deepen the study of blood flow in small vessels, implementing the two features described above: the treatment of blood as a non-Newtonian fluid (coupling the Carreau model with the Navier-Stokes equations) and the possibility of imposing Womerlsey profile for the velocity as a Dirichlet boundary condition at the inlet, solving the so called "inverse Womersley problem".

2. Mathematical modeling

Among all the various modules implemented in the life^x library, our project is completely developed inside the fluid dynamics module, which implements a solver for incompressible Navier-Stokes equations based on Galerkin Finite Elements Methods. The class offers high flexibility to the user about many parameters as the type of formulation, stabilization techniques, type of preconditioner to solve the linear system and many other features.
In order to better understand the fluid dynamics module, we present in the next session the Navier-Stokes equations, together with the weak formulation, Galerkin projection methods and finite dimensional spaces.

2.1. The Incompressible Navier-Stokes equations

Navier-Stokes equations constitute the main mathematical model to describe the motion of viscous fluid substances. This system consists in a set of partial differential equations that mathematically express conservation of momentum and conservation of mass for Newtonian or more general fluids, coupled with the continuity equation. The system is usually closed through a thermodynamic relation between pressure and temperature but for the purpose of this project this relation is replaced with the assumption of constant temperature. Let us now introduce the Navier-Stokes equations for incompressible fluid in convective form:

$$\begin{cases} \rho\frac{\partial \mathbf{u}}{\partial t} + \rho(\mathbf{u}\cdot\nabla)\mathbf{u} - \nabla\cdot\sigma = \mathbf{f} & \text{in } \Omega\times(0,T) \\ \nabla\cdot\mathbf{u} = 0 & \text{in } \Omega\times(0,T) \\ \mathbf{u} = \mathbf{g} & \text{on } \Gamma_D\times(0,T) \\ \sigma\mathbf{n} = \mathbf{h} & \text{on } \Gamma_N\times(0,T) \\ \mathbf{u}(\mathbf{x},0) = \mathbf{u}_0(\mathbf{x}) & \text{in } \Omega\times\{0\} \end{cases} \tag{2.1}$$

The system is solved in the spatial domain $\Omega \subset \mathbb{R}^d$, $d = 2, 3$ for all the time instants in the interval $[0, T)$ where $T \in \mathbb{R}^+$. The main unknowns are the fluid velocity field $\mathbf{u} : \Omega \times [0, T) \to \mathbb{R}^d$ and the pressure $p : \Omega \times [0, T) \to \mathbb{R}$. σ is called stress tensor and it is usually a function of $p, \mathbf{u}$ and the dynamic viscosity μ. Thanks to the incompressibility assumptions, the fluid density ρ is constant along flow trajectories and, for the purpose of this project, the density is supposed constant in space and time. In the momentum conservation equation we find also the external volume force $\mathbf{f} : \Omega \times [0, T) \to \mathbb{R}^d$. Moreover we define here the kinematic viscosity $\nu = \frac{\mu}{\rho}$ and the total pressure $p_T = p + \frac{1}{2}|\mathbf{u}|^2$. System (2.1) is valid for a generic fluid. Under the assumption of Newtonian fluid the stress tensor can be written in the form $\sigma = -p\mathbf{I} + 2\mu\varepsilon(\mathbf{u})$, where $\mathbf{I}$ is the $d \times d$ identity tensor and $\varepsilon(\mathbf{u}) = \frac{1}{2}(\nabla\mathbf{u} + (\nabla\mathbf{u})^T)$ is the strain rate tensor. In this case system (2.1) becomes:

Momentum equation	Natural boundary condition
$\rho\frac{\partial \mathbf{u}}{\partial t} + \rho(\mathbf{u}\cdot\nabla)\mathbf{u} - \nabla\cdot(\mu\nabla\mathbf{u} - p\mathbf{I}) = \mathbf{f}$	$-p\mathbf{n} + \mu\nabla\mathbf{u}\cdot\mathbf{n} = \mathbf{h}$
Laplacian formulation	
$\rho\frac{\partial \mathbf{u}}{\partial t} + \rho(\mathbf{u}\cdot\nabla)\mathbf{u} - \nabla\cdot(2\mu\varepsilon(\mathbf{u}) - p\mathbf{I}) = \mathbf{f}$	$\boldsymbol{\sigma}\cdot\mathbf{n} = -p\mathbf{n} + 2\mu\varepsilon(\mathbf{u})\cdot\mathbf{n} = \mathbf{h}$
Standard formulation	
$\rho\frac{\partial \mathbf{u}}{\partial t} + \rho(\boldsymbol{\nabla}\times\mathbf{u})\times\mathbf{u} - \nabla\cdot(2\mu\varepsilon(\mathbf{u}) - p_T\mathbf{I}) = \mathbf{f}$	$\boldsymbol{\sigma}_T\cdot\mathbf{n} = -p_T\mathbf{n} + 2\mu\varepsilon(\mathbf{u})\cdot\mathbf{n} = \mathbf{h}$
Total stress formulation	
$\frac{\partial \mathbf{u}}{\partial t} - \nabla\cdot\left(2\nu\varepsilon(\mathbf{u}) - \frac{p}{\rho}\mathbf{I} - \mathbf{u}\otimes\mathbf{u}\right) = \frac{1}{\rho}\mathbf{f}$	$-\rho(\mathbf{u}\cdot\mathbf{n})\mathbf{u} - p\mathbf{n} + 2\mu\varepsilon(\mathbf{u})\cdot\mathbf{n} = \mathbf{h}$
Momentum flux formulation	

Table 2.1: Different formulations for the Navier-Stokes system

$$\begin{cases} \rho\frac{\partial \mathbf{u}}{\partial t} + \rho(\mathbf{u}\cdot\nabla)\mathbf{u} - \nabla\cdot(2\mu\varepsilon(\mathbf{u})) + \nabla p = \mathbf{f} & \text{in } \Omega\times(0,T) \\ \nabla\cdot\mathbf{u} = 0 & \text{in } \Omega\times(0,T) \\ \mathbf{u} = \mathbf{g} & \text{on } \Gamma_D\times(0,T) \\ -p\hat{\mathbf{n}} + 2\mu\varepsilon(\mathbf{u})\hat{\mathbf{n}} = \mathbf{h} & \text{on } \Gamma_N\times(0,T) \\ \mathbf{u}(\mathbf{x},0) = \mathbf{u}_0(\mathbf{x}) & \text{in } \Omega\times\{0\} \end{cases} \tag{2.2}$$

Since we are dealing with an evolutive problem, we need an initial condition and this is provided by the last equation of (2.1). We now consider $\Gamma_D, \Gamma_N \subset \partial\Omega$ s.t. $\Gamma_D \cup \Gamma_N = \partial\Omega$ and $\Gamma_D \cap \Gamma_N = \{\}$. For every time instant in the interval $(0,T)$ we impose on Γ_D a value for the fluid velocity field (Dirichlet boundary condition), while on Γ_N we prescribe the value of the stress on the tangent surface in every point (Neumann boundary condition).

It is worth mentioning that different formulations are possible for the momentum conservation equation and those are associated to different natural Neumann boundary conditions. Different boundary conditions have different meaning, so, depending on the circumstances, one can be more suitable than the others ([17]). In Table 2.1 we report the 4 main formulations, together with the natural Neumann boundary conditions they give rise to.

2.1.1. Weak formulation

In order to proceed with the weak formulation we first make some assumptions on the data, in particular:

$$\mathbf{f} \in L^2(\mathbb{R}^+, [L^2(\Omega)]^d), \quad \mathbf{g} \in L^2(\mathbb{R}^+, [H^1(\Omega)]^d), \quad \mathbf{h} \in L^2(\mathbb{R}^+, [L^2(\Omega)]^d), \quad \mathbf{u_0} \in [H^1(\Omega)]^d \tag{2.3}$$

.

Under these assumptions, a weak formulation of (2.2) can be obtained by proceeding formally, in the canonical way. Let us multiply the momentum conservation equation by a test function $v \in \mathbf{V} = [H^1_{\Gamma_D}(\Omega)]^d$, divide by ρ and integrate in Ω:

$$\int_\Omega \frac{\partial \mathbf{u}}{\partial t} \cdot \mathbf{v} d\Omega - \int_\Omega \nabla \cdot (2\mu\varepsilon(\mathbf{u})) \cdot \mathbf{v} d\Omega + \int_\Omega [(\mathbf{u} \cdot \nabla)\mathbf{u}] \cdot \mathbf{v} d\Omega + \int_\Omega \nabla \tilde{p} \cdot \mathbf{v} d\Omega \quad = \int_\Omega \mathbf{f} \cdot \mathbf{v} d\Omega \tag{2.4}$$

Here $\tilde{p}$ is the pressure p divided by the density ρ and from now on the tilde will be omitted for ease of notation.

Similarly, multiplying the mass conservation equation by a test function $q \in Q = \begin{cases} L^2(\Omega) & \text{if } \Gamma_N \neq \{\} \\ L^2_0(\Omega) & \text{if } \Gamma_N = \{\} \end{cases}$ and integrating on Ω we obtain:

$$\int_\Omega q \nabla \cdot \mathbf{u} d\Omega = 0 \tag{2.5}$$

After integrating by parts the second and the fourth term of (2.4), using the Neumann boundary condition in (2.2) one finally ends up with the weak formulation, which can be stated as follows: Find $\mathbf{u} \in \mathrm{L}^2(\mathbb{R}^+; [\mathrm{H}^1(\Omega)]^d) \cap C^0(\mathbb{R}^+; [\mathrm{L}^2(\Omega)]^d)$, $p \in \mathrm{L}^2(\mathbb{R}^+; Q)$ such that:

$$\begin{cases} \int_\Omega \frac{\partial \mathbf{u}}{\partial t} \cdot \mathbf{v} d\Omega + \mathcal{D}(\mathbf{u}, \mathbf{v}; \mu) + \int_\Omega [(\mathbf{u} \cdot \nabla)\mathbf{u}] \cdot \mathbf{v} d\Omega \\ \qquad - \int_\Omega p \nabla \cdot \mathbf{v} d\Omega = \int_\Omega \mathbf{f} \cdot \mathbf{v} d\Omega + \int_{\Gamma_N} \mathbf{h} \cdot \mathbf{v} d\gamma & \forall \mathbf{v} \in \mathbf{V} \\ \int_\Omega q \nabla \cdot \mathbf{u} d\Omega = 0 & \forall q \in Q \\ \mathbf{u} = \mathbf{g} & \text{on } \Gamma_D \times (0, T) \\ \mathbf{u}(\mathbf{x}, 0) = \mathbf{u}_0(\mathbf{x}) & \text{in } \Omega \times \{0\} \end{cases} \tag{2.6}$$

Where $\mathcal{D}(\mathbf{u}, \mathbf{v})$ is the diffusion term and its form gives rise to three different but equivalent formulations of the weak problem:

$$\mathcal{D}(\mathbf{u}, \mathbf{v}; \mu) = \begin{cases} \int_\Omega \mu \nabla \mathbf{u} \cdot \nabla \mathbf{v} d\Omega & \text{Grad-Grad formulation} \\ \int_\Omega 2\mu\varepsilon(\mathbf{u}) \cdot \nabla \mathbf{v} d\Omega & \text{SymGrad-Grad formulation} \\ \int_\Omega 2\mu\varepsilon(\mathbf{u}) \cdot \varepsilon(\mathbf{v}) d\Omega & \text{SymGrad-SymGrad formulation} \end{cases} \tag{2.7}$$

The SymGrad-Grad formulation is obtained from the standard formulation i.e. system (2.2). Regarding the well-posedness of (2.6), existence of solutions can be proven for both dimensions $d = 2$ and $d = 3$, whereas uniqueness has been proven only in the case $d = 2$ for sufficiently small data. The details are available in many references, for instance in [20].

2.1.2. Space and time discretization

We first proceed with the time discretization. Let us define a set of times $\{t_k\}_{k=0}^{N}$ such that $0 = t_0 < t_1 < \cdots < t_N = T$ and $t_{k+1} - t_k = \Delta t \ \forall k \geqslant 0$, then, given a generic function of space and time $v(x,t)$, let us define $v^n(x) \coloneqq v(x,t_n) \ \forall n \in \{0,1,\dots,N\}$. We now approximate the time derivative in (2.4) with a generic BDF scheme in the following way:

$$\left.\frac{\partial \mathbf{u}}{\partial t}\right|_{t^{n+1}} \approx \frac{1}{\Delta t}\left(\alpha_{BDF}\mathbf{u}^{n+1} - \mathbf{u}^{n}_{BDF}\right) \tag{2.8}$$

where $\alpha_{BDF} > 0$ and $\mathbf{u}^n_{BDF}$ is a suitable function of $\mathbf{u}^n, \mathbf{u}^{n-1}, \dots$
Both quantities depend on the order of the selected BDF scheme.
For the space discretization let us consider the discretized domain $\Omega_h \subset \Omega$ and let $\mathcal{T}_h = \{K_i\}_{i=1}^{N_{el}}$ be the associated mesh. We now select two finite-element spaces $\mathbf{V}_h \subset \mathbf{V}$, $Q_h \subset Q$ uch that they fulfill the following property, called *Inf-Sup condition*:

$$\inf_{q_h \in Q_h} \sup_{\mathbf{v}_h \in \mathbf{V}_h} \frac{\int_{\Omega_h} q_h \nabla \cdot \mathbf{v}_h d\Omega}{\|\nabla \mathbf{v}_h\|_{L^2(\Omega)} \|q_h\|_{L^2(\Omega)}} \geqslant \beta_h > 0 \ \forall h \tag{2.9}$$

Moreover we assume that $\beta_h \xrightarrow{h \to 0^+} \beta > 0$ where h can be interpreted as the size of the elements of the mesh applied to obtain the discrete domain Ω_h.
Let us now apply the discretization both in space and time to (2.6) and use (2.8), moreover define $\mathbf{g}^n_h$, $\mathbf{f}^n_h$ and $\mathbf{h}^n_h$ as the projections of $\mathbf{g}^n$, $\mathbf{f}^n$ and $\mathbf{h}^n$ on $\mathbf{V}_h$ respectively.
Then the weak formulation reads as follows:

Set $\mathbf{u}^0_h$ equal to the projection of $\mathbf{u}_0$ on $\mathbf{V}_h$, then $\forall n \geqslant 0$, given $\mathbf{u}^n_h$, find $\left(\mathbf{u}^{n+1}_h, p^{n+1}_h\right) \in V_h \times Q_h$ such that $\mathbf{u}^{n+1}_h = \mathbf{g}^{n+1}_h$ on Γ_{Dh} and:

$$\begin{cases} \int_{\Omega_h} \frac{1}{\Delta t}\alpha_{BDF}\mathbf{u}^{n+1}_h \cdot \mathbf{v}_h d\Omega_h + \mathcal{D}(\mathbf{u}^{n+1}_h, \mathbf{v}_h; \mu) + \int_{\Omega_h}[(\mathbf{u_*}^{n+1} \cdot \nabla)\mathbf{u}^{n+1}_h] \cdot \mathbf{v}_h d\Omega_h - \int_{\Omega_h} p^{n+1}_h \nabla \cdot \mathbf{v}_h d\Omega_h \\ = \int_{\Omega_h} \frac{1}{\Delta t}\mathbf{u}^n_{hBDF} \cdot \mathbf{v}_h d\Omega_h + \int_{\Omega_h} \mathbf{f}^n_h \cdot \mathbf{v}_h d\Omega_h + \int_{\Gamma_N} \mathbf{h}^{n+1}_h \cdot \mathbf{v}_h d\gamma & \forall \mathbf{v}_h \in \mathbf{V}_h \\ \int_{\Omega_h} q_h \nabla \cdot \mathbf{u}^{n+1}_h d\Omega_h & \forall q_h \in Q_h \end{cases} \tag{2.10}$$

Here $\mathbf{u_*}^{n+1}$ is the advection velocity and it determines whether the problem is linear or not. Indeed we can have:

- $\mathbf{u_*}^{n+1} = \mathbf{u}^{n+1}_h$; in this case the formulation is non-linear and the problem is solved by means of Newton's method.
- $\mathbf{u_*}^{n+1} = \mathbf{u}^{n+1}_{hEXT}$ where $\mathbf{u}^{n+1}_{hEXT}$ is a suitable extrapolation of $\mathbf{u}^{n+1}_h$ based on $\mathbf{u}^n_h, \mathbf{u}^{n-1}_h, \dots$

In this case the formulation is clearly linear.

The fluid dynamics module includes the possibility to apply a stabilization procedure to problem (2.10). A generic stabilization technique consists in adding a term to the weak formulation which is equal to 0 in the continuous formulation, but it is no longer null when we pass to the discretized formulation. One of the most common is the so-called Streamline-Upwind-Petrov-Galerkin (SUPG), whose utility is twofold:

- it allows the user to select finite element spaces V_h, Q_h which are not Inf-Sup stable (i.e. they do not fulfill the Inf-Sup condition (2.9));
- it is suitable for advection-dominated problems, which are known for their unstable nature for many numerical approaches.

However there are two main side effects associated to the SUPG stabilization:

- the stabilization of advection-dominated problem is done through the introduction of an artificial diffusion coefficient, so there is a corruption of the original problem;
- if SUPG is implemented, then the convergence with respect to h is always of order 1, redardless to the degree of the finite element spaces employed in the simulation.

For more details about the SUPG stabilization see [22].
Other stabilization techniques are available in life^x in order to be able to solve problem (2.6) under turbulence conditions. However in this project we are always working under the assumption of laminar flow. This is justified by the fact that, although in the veins of the systemic circulation and in the aorta the Reynolds number can be greater than the threshold value above which, in a straight pipe, the flow would no longer be laminar, the pulsatile nature of blood flow does not allow a full transition to turbulence ([19]). This is not necessarily the case for some pathological conditions, such as carotid stenosis, which are characterized by a narrowing of the vessel lumen and, increasing the complexity of the geometry, the Reynolds number is consequently higher (see e.g. [14]).

2.1.3. Algebraic formulation

This section is devoted to the derivation of the algebraic formulation of (2.6). For simplicity we carry on the analysis without any stabilization term, however all the following argument can be generalized.

First we introduce the following operators:

$$b : \mathbf{V}_h \times Q_h \longrightarrow \mathbb{R} : \quad b(\mathbf{v}_h, p_h) = -\int_{\Omega_h} \nabla \cdot \mathbf{v}_h p_h d\Omega_h$$

$$c : \mathbf{V}_h \times \mathbf{V}_h \times \mathbf{V}_h \longrightarrow \mathbb{R} : \quad c(\mathbf{w}_h, \mathbf{u}_h, \mathbf{v}_h) = \int_{\Omega_h} (\mathbf{w}_h \cdot \nabla)\mathbf{u}_h \cdot \mathbf{v}_h d\Omega_h$$

$$F^n(\mathbf{v}_h) = \int_{\Omega_h} \mathbf{f}_h^n \cdot \mathbf{v}_h d\Omega_h + \int_{\Gamma_N} \mathbf{h}_h^n \cdot \mathbf{v}_h d\gamma$$

and the scalar product in the vector space $L^2(\Omega_h)$:

$$(\mathbf{f}, \mathbf{g})_h := \int_{\Omega_h} \mathbf{f} \cdot \mathbf{g} d\Omega_h \quad \forall\ \mathbf{f}, \mathbf{g}$$

If we sum the two equations of (2.6) we can reformulate the problem in the following way:
Set $\mathbf{u}_h^0$ equal to the projection of $\mathbf{u}_0$ on $\mathbf{V}_h$, then $\forall n \geqslant 0$, given $\mathbf{u}_h^n$, find $\left(\mathbf{u}_h^{n+1}, p_h^{n+1}\right) \in \mathbf{V}_h \times Q_h$ such that $\mathbf{u}_h^{n+1} = \mathbf{g}_h^{n+1}$ on Γ_{Dh} and $\forall (\mathbf{v}_h, q_h) \in \mathbf{V}_h \times Q_h$:

$$\frac{\alpha_{BDF}}{\Delta t}(\mathbf{u}_h^{n+1}, \mathbf{v}_h)_h + \mathcal{D}(\mathbf{u}_h^{n+1}, \mathbf{V}_h; \mu) + c({\mathbf{u}_*}^{n+1}, \mathbf{u}_h^{n+1}, \mathbf{v}_h) - b(p_h^{n+1}, \mathbf{v}_h) + b(q_h, \mathbf{u}_h^{n+1}) = \frac{1}{\Delta t}(\mathbf{u}_{hBDF}^n, \mathbf{v}_h)_h + F^{n+1}(\mathbf{v}_h) \quad (2.11)$$

We proceed introducing a (vector) basis $\left\{\boldsymbol{\phi}_j\right\}_{j=1}^{\mathcal{N}_\mathbf{u}}$ for the space $\mathbf{V}_h$ and a (scalar) basis $\{\psi_k\}_{k=1}^{\mathcal{N}_p}$ for the space Q_h, where we have denoted with $\mathcal{N}_\mathbf{u} = \dim(\mathbf{V}_h)$ and $\mathcal{N}_p = \dim(Q_h)$ the dimensions of the finite element spaces of velocity and pressure, respectively. The solution $(\mathbf{u}_h, p_h)$ of problem (2.11) can be written in the following way:

$$\mathbf{u}_h^{n+1}(\mathbf{x}) = \sum_{j=1}^{\mathcal{N}_u} u_j^{n+1} \boldsymbol{\phi}_j(\mathbf{x}), \quad p_h^{n+1}(\mathbf{x}) = \sum_{k=1}^{\mathcal{N}_p} p_k^{n+1} \psi_k(\mathbf{x}) \quad (2.12)$$

Thanks to the linearity of (2.11) w.r.t. $\mathbf{v}_h$ and q_h, it is enough to impose the equality for every possible choice of the test functions in the sets of basis functions $\left\{\boldsymbol{\phi}_j\right\}_{j=1}^{\mathcal{N}_\mathbf{u}}$ and $\{\psi_k\}_{k=1}^{\mathcal{N}_p}$.
After introducing the matrices:

$$\begin{aligned}
&A \in \mathbb{R}^{\mathcal{N}_\mathbf{u} \times \mathcal{N}_\mathbf{u}} : && A_{ij} = \mathcal{D}\left(\boldsymbol{\phi}_j, \boldsymbol{\phi}_i; \mu\right) \\
&B \in \mathbb{R}^{\mathcal{N}_p \times \mathcal{N}_\mathbf{u}} : && B_{lj} = b\left(\boldsymbol{\phi}_j, \psi_l\right) \\
&N(\mathbf{w}) \in \mathbb{R}^{\mathcal{N}_\mathbf{u} \times \mathcal{N}_\mathbf{u}} : && N(\mathbf{w})_{ij} = c\left(\mathbf{w}, \boldsymbol{\phi}_j, \boldsymbol{\phi}_i\right) \\
&L(\mathbf{w}) \in \mathbb{R}^{\mathcal{N}_\mathbf{u} \times \mathcal{N}_\mathbf{u}} : && L(\mathbf{w})_{ij} = c\left(\boldsymbol{\phi}_j, \mathbf{w}, \boldsymbol{\phi}_i\right) \\
&M \in \mathbb{R}^{\mathcal{N}_\mathbf{u} \times \mathcal{N}_\mathbf{u}} : && M_{ij} = \int_{\Omega_h} \boldsymbol{\phi}_j \cdot \boldsymbol{\phi}_i d\Omega_h
\end{aligned}$$

and the vectors:

$$
\begin{aligned}
U^n &= \left[u_1^n, \ldots, u_{\mathcal{N}_\mathbf{u}}^n\right]^T \\
P^n &= \left[p_1^n, \ldots, p_{\mathcal{N}_p}^n\right]^T \\
F^n &= \left[F^n\left(\phi_1\right), \ldots, F^n\left(\phi_{\mathcal{N}_\mathbf{u}}\right)\right]^T
\end{aligned}
$$

it can be proved that, by (2.12), setting $(\mathbf{v}_h, q_h) = (\boldsymbol{\phi}_i, \psi_i)$ in (2.11) and applying the Newton's method to treat the non-linearity in the advection term we end up with the following algebraic system:

$$
\begin{bmatrix} C^n & B^T \\ -B & S \end{bmatrix} \begin{bmatrix} U^{n+1} \\ P^{n+1} \end{bmatrix} = \begin{bmatrix} \tilde{F}^{n+1} + \frac{M}{\Delta t} U^n \\ 0 \end{bmatrix} \tag{2.13}
$$

where $C^n = \frac{M}{\Delta t} + A + N(U^n) + L(U^n)$ and $\tilde{F}^{n+1} = F^{n+1} + \frac{M}{\Delta t} U^n + N(U^n)U^n$

However, in life[x], problem (2.13) is recast in the following way:

$$
\begin{bmatrix} F & B^T \\ -B & S \end{bmatrix} \begin{bmatrix} \delta\mathbf{U} \\ \delta\mathbf{P} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{bmatrix} \tag{2.14}
$$

where $F, \mathbf{b}_u$ and $\mathbf{b}_p$ can be obtained from (2.13). Then the solution is updated as (U^{n+1}, P^{n+1})=$(U^n, P^n) - (\delta U, \delta P)$.

Because of its complex form, the algebraic system (2.13) the system is generally ill conditioned, so a suitable perconditioner must be employed in order to obtain a solution efficiently.

For this project the SIMPLE preconditioner is employed and it can be written in the following form:

$$
P_{\text{SIMPLE}} = \begin{bmatrix} C^n & 0 \\ -B & -\widetilde{\Sigma} \end{bmatrix} \begin{bmatrix} I & D_n^{-1} B^T \\ 0 & I \end{bmatrix} \tag{2.15}
$$

Where $D_n = diag(C^n)$ and $\widetilde{\Sigma}$ is an approximation of the so called Schur complement. More details about the SIMPLE preconditioner can be found in [18].

2.2. Modeling Non-Newtonian fluids

Blood is a complex fluid composed by many elements, such as red blood cells, white blood cells and platelets, suspended in an aqueous solution of organic molecules, proteins, and salts called plasma. By means of this multicomponent nature, blood exhibits complex rheological properties.

Blood plasma, which consists mostly of water, is a Newtonian fluid, that is, a fluid which satisfies the Newton's law of viscosity: the shear stress is proportional to the rate of shear and the viscosity of the fluid is the constant of proportionality. This is why in computational fluid-dynamics, the assumption of Newtonian flow is generally accepted for blood flow in large-sized arteries, such as the aorta, where the shear rates are high ([10]).
Blood, however, has more complex mechanical properties: those become especially significant where the particles size is comparable with the lumen size. Indeed, as we said, blood is not only composed of plasma but, with a relevant percentage, of other particles and, in particular, of red blood cells (erythrocytes). At low shear rates, the erythrocytes tend to aggregate and those aggregation will increase the viscosity of the blood. On the contrary, if the shear rate is increased, the latter will eventually be high enough to deform the erythrocytes, thus decreasing the viscosity. We remind to [23] for more details about this process. For this reason blood has in general higher viscosity than plasma and, when the volume percentage of red blood cells (known in medicine as hematocrit) rises, the viscosity of the suspension increases and the non-Newtonian behaviour of blood becomes more relevant: this happens especially at very low shear rates. In particular, blood features a so called shear-thinning behaviour, that is, its viscosity decreases with increasing shear rates, reaching a nearly constant value of approximately 0.00345 $\mathrm{Pa} \cdot \mathrm{s}$ only for shear rates grater than 200 s^{-1} ([6]).
This is why non-Newtonian effects has to be taken in account for better accuracy of the simulation when dealing with small vessels and low shear rates (that is the field under investigation in our project).

2.2.1. Carreau model for blood rheology

In a CFD simulation, in order to take into account the non-Newtonian nature of blood, one needs a constitutive equation that describes the relationship between the viscosity and the shear rate. If, as it usually happens for blood flow simulations, only the shear-rate dependent viscosity is considered, one may use as constitutive equation one of the equations developed for the so called "generalised Newtonian fluids" in which a relationship between the scalar viscosity and the rate of strain tensor $\dot{\gamma}$ is introduced. This dependency must be such that the constitutive equation does not depend on the selected coordinate system: this is achieved if and only if the set of variables is a subset of the invariants of the tensor $\varepsilon(\boldsymbol{u})$. The generalised Newtonian constitutive equation reads as $\boldsymbol{\sigma} = -p\boldsymbol{I} + 2\mu\left(II_\varepsilon\right)\varepsilon(\boldsymbol{u})$, where II_ε represents the second invariant of the strain rate tensor. Recalling that for incompressible flows $\mathrm{tr}\left(\varepsilon(\boldsymbol{u})\right) = \nabla \cdot \boldsymbol{u} = 0$, we have $II_\varepsilon = -\frac{1}{2}\left(\mathrm{tr}\left(\varepsilon(\boldsymbol{u})^2\right)\right)$, so if we define the shear rate as $\dot{\gamma} = \sqrt{2\,\mathrm{tr}\left(\varepsilon(\boldsymbol{u})^2\right)}$ the constitutive equation for generalized Newtonian fluids can be written as:

$$\boldsymbol{\sigma} = -p\boldsymbol{I} + 2\mu(\dot{\gamma})\varepsilon(\boldsymbol{u}) \tag{2.16}$$

where the relationship $\mu(\dot{\gamma})$ must be specified. Several models can be used for this relationship: one of the most used ones for blood simulations is the Carreau model. In this model the equation

for the viscosity is given by:

$$\mu(\dot{\gamma}) = \mu_\infty + (\mu_0 - \mu_\infty)\left[1 + (\lambda\dot{\gamma})^2\right]^{\frac{n-1}{2}} \tag{2.17}$$

where $\mu_\infty = 3.45 \times 10^{-3} Pa \cdot s, \mu_0 = 5.6 \times 10^{-2} Pa \cdot s, n = 0.3568$ and $\lambda = 3.313$ s. Those values for the parameters are well established in literature and they are obtained in ([6]) by numerical fitting of experimental data. It should be noticed that μ_∞ correspond to the constant viscosity usually imposed in the Newtonian case: $\mu_\infty = \mu_{newtonian}$, so the model switches off as the shear stress goes to $+\infty$. This model was found to fit well experimental data (see for example [6], [15], [21]). The introduction of this constitutive equation leads to an additional non-linear term in the Navier-Stokes equation (since now the viscosity depend on the solution $\boldsymbol{u}$).

2.2.2. Blood viscosity and the shear rate

Blood is a non-Newtonian, shear thinning fluid with many properties. Usually cardiovascular handbooks consider blood viscosity values between 3.5×10^{-3} and $5.5 \times 10^{-3} Pa \cdot s$ to be normal ([7]). However, blood viscosity cannot be summarized by a single value. Due to the shear thinning property of blood, the viscosity of this fluid changes depending on the hemodynamic conditions. For example, in correspondence of a very low shear rate, the viscosity can reach even a physiological value of $6 \times 10^{-2} Pa \cdot s$ ([11]).
According to equation (2.17) the value of the viscosity is driven by the shear rate, which is a function of the symmetric velocity gradient. In particular, looking to Figure 2.1 we can roughly identify 3 regions:

- a high viscosity plateau, corresponding to a value of the shear rate $< \sim 10^{-1} s^{-1}$, for which the value of the viscosity is approximately constant and equal to $\mu_0 = 5.6 \times 10^{-2} Pa \cdot s$,
- a transition region for a value of the shear rate between $\sim 10^{-1} s^{-1}$ and $\sim 5 \times 10^2 s^{-1}$. In this region the viscosity is very sensitive to the shear rate.
- a low viscosity plateau, for a shear rate $> \sim 5 \times 10^2 s^{-1}$. Also in this region the viscosity can be considered constant and equal to $\mu_\infty = 3.45 \times 10^{-3} Pa \cdot s$, i.e. the value employed in Newtonian models.

Since the shear rate grows increasing the magnitude of the symmetric velocity gradient, we can draw some hypothesis on the expected distribution of the viscosity in the cylindrical given the blood flow.
For example, in case of parabolic velocity profile over a circular section the velocity gradient is null in the center and it increases moving towards the boundary, producing a high value for the viscosity in the region near the center of the section.
We can replicate the argument considering a generic Womersley profile (see Figure 2.2, its definition is provided rigorously in the next section): we expect a high value of the viscosity in the

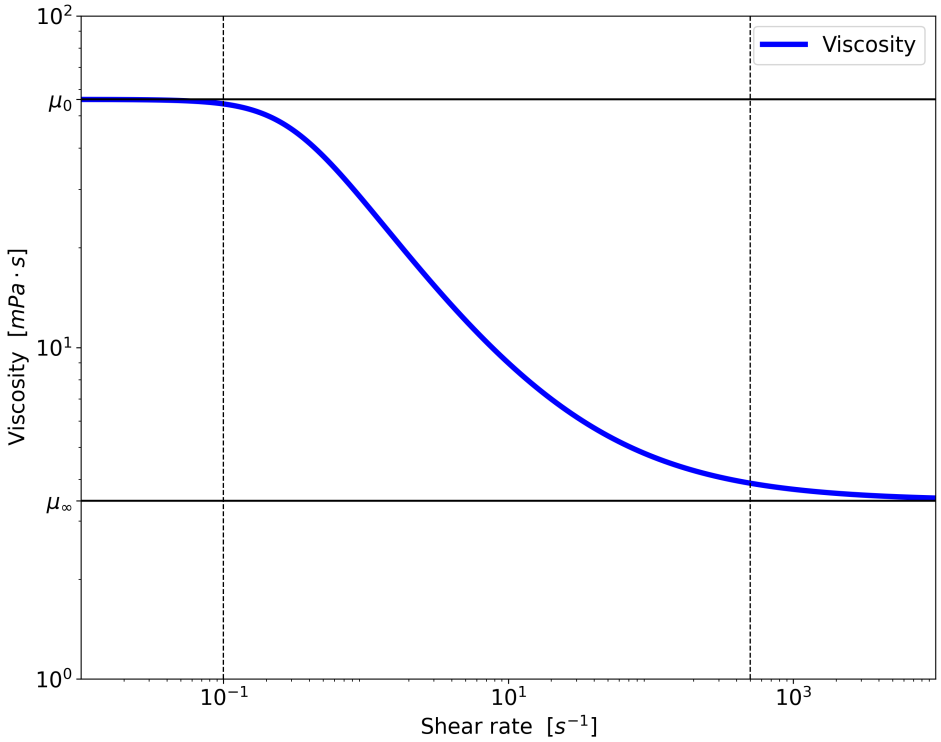


Figure 2.1: Relation between shear rate and dynamic viscosity according to Carreau model (2.17)

regions near each minimum/maximum point since they correspond to null velocity gradient and a lower value in the other regions.

2.3. Inverse Womersley problem

We stated in the previous sections that a parabolic velocity profile is a good approximation for the blood flow in very small vessels as much as a uniform velocity profile is a good representative for the blood flow in big vessels. Indeed these two profiles are limit cases of the well known Womersley profile, i.e. the solution to the incompressible Navier-Stokes equations in a cylinder when a oscillating pressure gradient is imposed. As a consequence, if the diamater of the vessel is neither sufficiently big or small (we will specify later this condition in a more rigorous way) the former approximations are no more suitable and the exact Womersley profile must be imposed.
The second main problem addressed in this project is the so-called **inverse Womersley problem**: given a time evolution of the flow rate through a circular surface, we aim to reconstruct the time evolution of the Womersley profile so that we can use it as a Dirichlet boundary condition. The solution of this problem represents a valuable tool for a variety of research branches, also including biological fluid dynamics and biomedical engineering, since flow rate is the main physical quantity which can be actually measured in many practical situations.

2.3.1. Womersley velocity profile

Let us consider a cylindrical pipe of length l, radius R, filled with a viscous liquid of density ρ and dynamic viscosity μ. We recall the definition of kinematic viscosity $\nu = \frac{\mu}{\rho}$. Let (r, θ, z) be the reference system where z is the longitudinal direction and let $\mathbf{u} = (u_r, u_\theta, u_z)$.
In steady flow the solution is the well-known Poiseouille velocity profile: we immediately find $u_r = u_\theta = 0$ and $\frac{\partial p}{\partial r} = \frac{\partial p}{\partial \theta} = 0$. Moreover, if p_1 and p_2 are the pressures respectively at the beginning and at the end of the pipe, the pressure gradient in the longitudinal direction is $\frac{\partial p}{\partial z} = \frac{p_2 - p_1}{l} \leqslant 0$.
The equation of the motion of the fluid in the z direction reduces to the following form:

$$\frac{\mathrm{d}^2 u_z}{\mathrm{d}r^2} + \frac{1}{r}\frac{\mathrm{d}u_z}{\mathrm{d}r} - \frac{1}{\mu}\frac{\partial p}{\partial z} = 0 \tag{2.18}$$

and its solution is

$$u_z = -\frac{1}{4\mu}\frac{\partial p}{\partial z}\left(R^2 - r^2\right) \tag{2.19}$$

Now we suppose that the pressure gradient is oscillating with frequency $f = \frac{n}{2\pi}$. For a more complete theoretical discussion we work in the field of complex numbers following the procedure described in the original work by Womersley ([24]), in particular we suppose that $\frac{\partial p}{\partial z} = Ae^{jnt}$ where $A \in \mathbb{R}$ and j is the imaginary unit. Then equation (2.18) presents an additional time-derivative term:

$$\frac{\partial^2 w}{\partial r^2} + \frac{1}{r}\frac{\partial w}{\partial r} - \frac{1}{\nu}\frac{\partial w}{\partial t} = \frac{A}{\mu}\mathrm{e}^{jnt} \tag{2.20}$$

we now make the substitution $w = u(r)\mathrm{e}^{jnt}$ in (2.20) and we obtain:

$$\frac{\mathrm{d}^2 u}{\mathrm{d}r^2} + \frac{1}{r}\frac{\mathrm{d}u}{\mathrm{d}r} - \frac{jn}{\nu}u = \frac{A}{\mu}. \tag{2.21}$$

If we write (2.21) in the following form:

$$r^2\frac{\mathrm{d}^2u}{\mathrm{d}r^2}+r\frac{\mathrm{d}u}{\mathrm{d}r}+\frac{j^3n}{\nu}u=\frac{A}{\mu} \tag{2.22}$$

we can recognize a non-homogeneous modified Bessel differential equation (see [5] for more details) and its solution is:

$$u=-\frac{A}{\rho}\frac{1}{jn}\left\{1-\frac{J_0\left((-1)^{\frac{3}{4}}r\sqrt{\frac{n}{\nu}}\right)}{J_0\left((-1)^{\frac{3}{4}}R\sqrt{\frac{n}{\nu}}\right)}\right\} \tag{2.23}$$

where

$$J_k(w):=\sum_{m=0}^{\infty}\frac{(-1)^m(w/2)^{2m+k}}{m!\Gamma(m+k+1)}\quad k\in\mathbb{N},w\in\mathbb{C}$$

denotes the Bessel function of the first kind of order k, while Γ is the Euler gamma function.
It is well known that this kind of functions arise in problems connected with the distribution of current in conductors: this it is not surprising since it is well established in literature that, considering the mathematical aspect, there is a strong analogy between the velocity and pressure gradient of blood flow and the current and potential difference in a conductor.
Note that in the (complex) argument of the bessel functions, the dimensionless quantity $\mathrm{Wo}:=r\sqrt{\frac{n}{\nu}}$ express the ratio between inertial oscillatory forces and viscous forces and this is the so-called Womersley number (Wo), widely used in fluid dynamics to describe the unsteady nature of fluid flow in response to an unsteady pressure gradient.
Finally if we take as pressure gradient the real part of Ae^{int}, the corresponding flow would be the real part of (2.23).
Figure 2.2 show some velocity profiles corresponding to different values of Wo and we can notice that the parabolic and uniform profiles are the limit cases of the Womersley profile respectively for $\mathrm{Wo}\longrightarrow 0$ and $\mathrm{Wo}\longrightarrow+\infty$. Moreover experiments show a phase difference between the flowrate and the pressure gradient, just like a phase difference between current and potential difference is observed in conductors and this phase difference varies with the value of *Wo*.
We can summarize as follows:

- When Wo is small ($\leqslant 1$) it means the frequency of pulsations is sufficiently low that a parabolic velocity profile has time to develop during each cycle and the flow will be very nearly in phase with the pressure gradient. In this case a to a good approximation will be given by the Poiseuille's law, using the instantaneous pressure gradient.

- When Wo is large ($\geqslant 10$) it means the frequency of pulsations is sufficiently large that the velocity profile is relatively flat and the mean flow is about 90 degrees ahead the pressure gradient in terms of phase.

Now that we have an insight about what is the Womersley velocity profile and how is it derived, we can proceed with the description of the second part of this this project: the so-called **Inverse Womersley problem**.

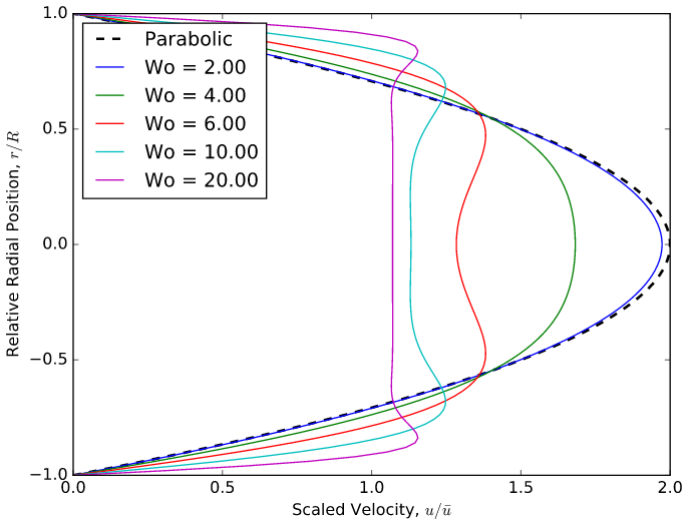


Figure 2.2: Velocity profiles for different values of Wo

Let us consider the following situation: we want to perform a numerical simulation of blood flow in a vessel (modeled as a cylinder) and we know the value of the blood flow rate for a time interval equal to the period of the pulsation of the flow itself (recall that the flow rate and the pressure gradient oscillate with the same period). It would be interesting to set in the inlet surface a Dirichlet boundary condition imposing the Womersley velocity profile $v(r,t)$ corresponding to the given flow rate $q(t)$: this can be done solving the **Inverse Womersley problem** and in the following we show how to solve this problem step by step.

2.3.2. Assumptions and definition of parameters

Let us suppose that the the flow rate oscillates with period $T \in \mathbb{R}^+$. We define $\sigma(t) := -\frac{1}{\rho}\frac{\partial p}{\partial z}$ and we assume that $q(t), \sigma(t)$ and $v(r,t)$ are regular enough to admit a Fourier expansion (this assumption is quite straightforward according to (2.6)) and we truncate the series up to a sufficiently large number of components to ensure the desired accuracy in terms of approximation, namely:

$$\begin{pmatrix} q(t) \\ \sigma(t) \\ v(r,t) \end{pmatrix} = \sum_{n=-N}^{N} \begin{pmatrix} q_n \\ \sigma_n \\ v_n(r) \end{pmatrix} e^{j\omega_n t} \tag{2.24}$$

where $q_n, \sigma_n, v_n(r) \in \mathbb{C}$, $N \in \mathbb{N}$ and $\omega_n := \frac{2\pi n}{T}$ $\forall n \in \{-N, -N+1, \dots, N-1, N\}$.
The idea is to obtain the map $q_n \mapsto v_n$ as the composition of two maps:

1. $q_n \mapsto \sigma_n$
2. $\sigma_n \mapsto v_n$

It is worth mentioning that we can study the relation amongst q_n, σ_n and v_n by assuming $n \geqslant 0$. Indeed, since we are interested in real valued solutions, we directly impose $(q_{-n}, \sigma_{-n}, v_{-n}) = (q_n^*, \sigma_n^*, v_n^*)$, where the star superscript denotes complex conjugation.

Moreover the case $n = 0$ corresponds to a constant flow, therefore the values of σ_0 and v_0 can be derived from the Poiseuille flow.
Finally, given the Fourier expansion (2.24), it is useful to define here the following generalized Womersley number:

$$\mathrm{Wo}_{r,n} := r\sqrt{\frac{\omega_n}{\nu}} \tag{2.25}$$

2.3.3. Map $q_n \mapsto \sigma_n$

As we mentioned before, σ_0 is obtained using the definition of q_0 and (2.19). The result is the following relation:

$$\sigma_0 = \frac{8\nu q_0}{\pi R^4} \tag{2.26}$$

For $n > 0$, the map $q_n \mapsto \sigma_n$ is formally studied in [8] for an arbitrary domain by considering a Fourier expansion with sines and cosines (conversion into complex exponential form is straightforward). The problem consists in a 4-th order equation, which can be also decomposed in two coupled Poisson problems.
This problem is then solved in [4] for a circular section which, thanks to its smoothness, provide the following explicit relation:

$$\frac{q_n}{\pi R^2} = \left[1 - \frac{{}_0\tilde{F}_1\left(;2;j\mathrm{Wo}_{R,n}^2/4\right)}{{}_0\tilde{F}_1\left(;1;j\mathrm{Wo}_{R,n}^2/4\right)}\right]\frac{\sigma_n}{j\omega_n}, \quad \forall n > 0 \tag{2.27}$$

where

$${}_0\tilde{F}_1(;b;w) := \sum_{k=0}^{\infty}\frac{w^k}{k!\Gamma(b+k)}, \quad b, w \in \mathbb{C} \tag{2.28}$$

denotes the regularized confluent hyper-geometric limit function and Γ is, once again, the Euler gamma function.

2.3.4. Map $\sigma_n \mapsto v_n$

In order to obtain v_0 we exploit once again the Poiseuille flow. In particular, formula (2.19) directly leads to:

$$v_0(r) := \frac{\sigma_0 R^2}{4\nu}\left[1 - \left(\frac{r}{R}\right)^2\right] \tag{2.29}$$

About v_n for $n > 0$, we can use the linearity of the problem with respect to the pressure gradient and solve the equation considering only the n-th term of the Fourier decomposition (and the term corresponding to -n) of σ. In this case it's enough to exploit (2.23) and the Fourier expansions of both $v(r,t)$ and $\sigma(t)$.
The result is the following relation:

$$v_n = \left\{ 1 - \frac{J_0\left[(-1)^{3/4}\mathrm{Wo}_{r,n}\right]}{J_0\left[(-1)^{3/4}\mathrm{Wo}_{R,n}\right]} \right\} \frac{\sigma_n}{j\omega_n} \quad \forall n > 0 \tag{2.30}$$

Summing up, we can solve the **Inverse Womersley problem** by the following steps:

1. Given the flow rate $q(t)$ we approximate it through a truncated Fourier expansion $q(t) = \sum_{n=-N}^{N} q_n e^{jnt}$.
2. We compute σ_0 and $v_0(r)$ from q_0 exploiting the theory of the Poiseuille flow (equation (2.19)). In particular we obtain (2.26) and (2.29) .
3. For $n \in \{1, 2, \ldots, N\}$, σ_n is obtained inverting equation (2.27) while $v_n(r)$ is derived from (2.30).
4. σ_{-n} and $v_{-n}(r)$ are obtained respectively as the complex conjugate of σ_n and $v_n(r)$ $\forall n \in \{1, 2, \ldots, N\}$.
5. Finally $\sigma(t)$ and $v(r,t)$ are obtained by summation as in (2.24).

3. Implementation

This project was developed in the framework of the life^x library ([2]). life^x is a high performance library for the solution of multi-physics and multi-scale problem for cardiac applications ([3]). To deal with the demanding computational costs, it is written in C++ using the most modern programming techniques available in the C++17 standard and is based on the deal.II finite element core ([1]). All the code is natively parallel, highly scalable and designed to run on different architectures.

The life^x library consists in different classes: each one is designed to solve a specific problem. They are eventually coupled and often further specialized for the solution of practical problems (the so called applications). In particular there are several classes implemented for the numerical solution of some macro physical problem such as electrophysiology, electromechanics, fluid dynamics and many others. Every class has several dependencies: the main building block is the deal.II finite element library.

In this project we will mainly focus on the *FluidDynamics* class: in the next sections we are going to briefly describe this class: this will help in the understanding of the implementation of the specific subjects of this project.

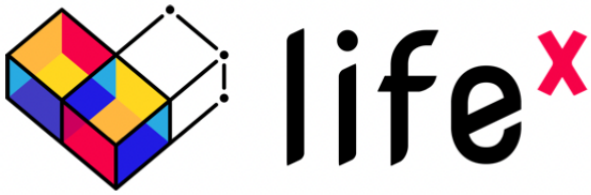


Figure 1: life^x logo.

3.1. *FluidDynamics* class

The *FluidDynamics* class implements the solver for the incompressible Navier-Stokes equations by means of Galerkin Finite Elements Methods as we discussed in Chapter 2.

This class, as the majority of the physical models implemented in life^x, inherits from the *Core* class, as can be seen from Figure 3.1. The *Core* class is implemented to provide a common interface for handling the parameters and to some utilities such as MPI communicator, rank and size, a parallel standard output object, a parallel error output object and a timer output. Additionally, the *CoreModel* class provides a common interface for parameter handling. Thanks to the inheritance from *CoreModel*, the *FluidDynamics* class offers high flexibility to the user about many parameters such as the type of formulation, stabilization techniques, type of preconditioner to solve the linear system and many of the features related to the numerical solution of the Navier Stokes problem that are described in Chapter 2. In addition the typical parameters that are required for a simulation, such as the dimension of the domain and the computational mesh, are included. This is done

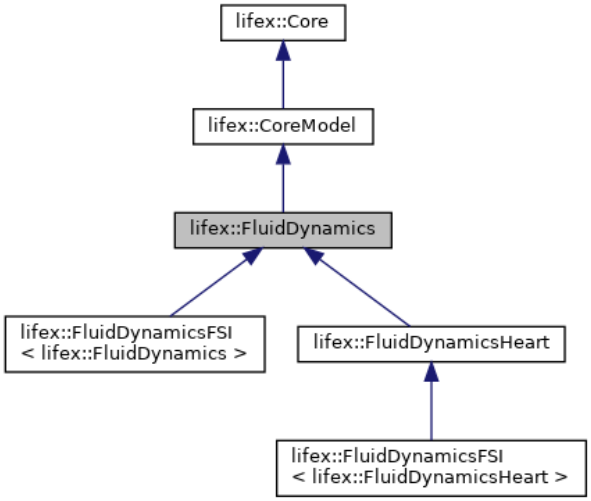


Figure 3.1: Inheritance diagram for the *FluidDynamics* class

through two methods inherited from *CoreModel*: `declare_parameters` and `parse_parameters`. Thanks to these methods the user can put all the parameters required in a simulation in a *.prm* file (generated with an arbitrarily chosen verbosity by `declare_parameters`, when the code is compiled with *-g* option): those are then directly read in the code by `parse_parameters`.
As we said, the *FluidDynamics* class serves only as a generic Navier-Stokes solver. So, to use this class for the solution of any specific problem, a further step needs to be done in order to decide the domain, how to apply boundary conditions and so on. This can be done through a further specialization (this is done for example in *FluidDynamicsHeart* or *FluidDynamicsFSI*, see bottom of Figure 3.1), or using other classes in which *FluidDynamics* is used as a member. This is the case, for example, of *TestFluidDynamicsCylinder*: a class that is specialized in the solution of a fluid dynamics problem in a cylindrical domain. This is the "specialized" class that we used the most in our project to test our implementation: for this reason and for simplicity we will use this particular test as an example to explain how the code works in practice.

```
int
main(int argc, char **argv)
{
  lifex::lifex_init lifex_initializer(argc, argv, 1);
  try
    {
      lifex::utils::CSVTest<lifex::tests::TestFluidDynamicsCylinder> test(
        "Test fluid dynamics cylinder", "Fluid dynamics");
      test.main_run_generate([&test]() {

        ... //series of parameters that we do not specify for simplicity

      });
    }
  LIFEX_CATCH_EXC();
  return EXIT_SUCCESS;
}
```

Listing 3.1: `main`

In the main, after first a member of a class called *lifex_init* is constructed in order to initialize life^x

Core functionalities, the constructor of *TestFluidDynamicsCylinder* is called (actually through an object of a template class of type *CSVTest* <*Model* >, as can be seen in the code above). Since, as we said, *TestFluidDynamicsCylinder* has a member of type *FluidDynamics*, the constructor of *FluidDynamics* is called. After this, `main_run_generate` (a member function of *CoreModel*) is called. With this function all the parameters are properly set through the `parse_parameters` function that we mentioned before. Then, the function `run`, originally defined as a member of *CoreModel* but overridden in every child (that in this case represents the different physical models) is called. The function `run` is in charge to run the model simulation and so, in this specific case, to solve the Navier Stokes equations.
The function `TestFluidDynamicsCylinder.run` first set up the right boundary conditions in the cylindrical domain (this part will be addressed and described better in **??** where the main focus will the imposition of the boundary conditions) and then `FluidDynamics.run` is finally called: this is the function that performs the solution of the Navier Stokes equations by means of Finite Elements method. The most important lines of this function are shown below:

```
void
FluidDynamics::run()
{
  setup_system();
    ...
  // Time loop.
  while (time < prm_time_final)
    {
      time_advance();
        ...
      apply_BCs();
        try
          {
            solve_time_step(true);
          }
        ...
      output_results();
      print_info();
    }
}
```

Listing 3.2: `FluidDynamics::run()`

`setup_system` deals with the initialization of the mesh, the finite element spaces and allocates the matrices and vectors that we are going to use for the assembly of the algebraic system (with size depending on the number of quadrature nodes).
The assembly of system (2.14) is then performed at every time step through the function `solve_time_step`: this is done in a classic way looping on the elements of the grid and integrating with a quadrature rule the basis function on each quadrature node. The linearized system based on Newton is then solved using a preconditioner (chosen by the user) and the quantities are then updated through the method `time_advance`.

3.2. Non-Newtonian model: numerical scheme and implementation

For the solution of the Navier-Stokes equation in case of non-Newtonian fluid, the time scheme discretization can be obtained in analogous way as in Subsection 2.1.2 considering the backward Euler method with a semi-implicit treatment of the convective term. The non-linear term arising from the Carreau model is treated semi-implicitly (as in [10]). The resulting formulation is analogous to Equation 2.10 where the diffusive term, this time, depends on the previous solutions interpolated by the BDF formula:

$$\mathcal{D}(\mathbf{u}_h^{n+1}, \mathbf{v}_h; \mu) = \mathcal{D}(\mathbf{u}_h^{n+1}, \mathbf{v}_h, \mu\left(\mathbf{u}_{hBDF}^n\right)) \tag{3.1}$$

where $\mu\left(\mathbf{u}_{hBDF}^n\right)$ is calculated using the formula derived by the Carreau model showed in Equation 2.17.

To implement this new feature in the *FluidDynamics* class some fundamental steps had to be performed:

- The function `parse_parameters` and `declare_parameters` had to be modified in order to add a new section containing the parameters for the non-Newtonian model:

```
1   params.enter_subsection("Non-Newtonian model");
2   {
3     params.declare_entry_selection("Model",
4                                    "None",
5                                    "None | Carreau",
6                                    "Non-Newtonian model to be used.");
7
8     params.declare_entry("Viscosity zero",
9                          "5.6e-2",
10                         Patterns::Double(0),
11                         "Viscosity zero [Pa * s].");
12
13    params.declare_entry("Viscosity infinity",
14                         "3.45e-3",
15                         Patterns::Double(0),
16                         "Viscosity infinity [Pa * s].");
17
18    params.declare_entry("Exponent power law",
19                         "0.3568e0",
20                         Patterns::Double(0),
21                         "Exponent power law [1].");
22
23    params.declare_entry("Lambda",
24                         "3.313e0",
25                         Patterns::Double(0),
26                         "Lambda [s].");
27
28    params.declare_entry(
29      "Enable viscosity output",
30      "true",
31      Patterns::Bool(),
32      "If a non-newtonian model is enabled, toggle output of viscosity↩
```

```
            .");
33    }
34    params.leave_subsection();
```

Listing 3.3: `declare_parameters`

```
1     params.enter_subsection("Non-Newtonian model");
2     {
3       const std::string &non_newtonian_model_string = params.get("Model"↩
          );
4
5       // NonNewtonianModel.
6       if (non_newtonian_model_string == "None")
7         {
8           prm_flag_non_newtonian_model = NonNewtonianModel::None;
9         }
10      else if (non_newtonian_model_string == "Carreau")
11        {
12          prm_flag_non_newtonian_model = NonNewtonianModel::Carreau;
13        }
14
15      if (prm_flag_non_newtonian_model == NonNewtonianModel::Carreau)
16        {
17          prm_carreau_viscosity_zero = params.get_double("Viscosity zero↩
              ");
18          prm_carreau_viscosity_infinity =
19            params.get_double("Viscosity infinity");
20          prm_carreau_exponent_power_law =
21            params.get_double("Exponent power law");
22          prm_carreau_lambda = params.get_double("Lambda");
23        }
24
25      prm_flag_output_viscosity = params.get_bool("Enable viscosity ↩
          output");
26    }
27    params.leave_subsection();
```

Listing 3.4: `parse_parameters`

Thanks to these additions the user is allowed to work with all the parameters needed for the non-Newtonian model directly in the .prm file.

- The quantities that are used in the non-Newtonian model had to be computed in every quadrature node.

```
1     if (prm_flag_non_newtonian_model != NonNewtonianModel::None)
2       {
3         for (unsigned int c1 = 0; c1 < dim; ++c1)
4           for (unsigned int c2 = 0; c2 < dim; ++c2)
5             symgrad_u_ext_loc[q][c1][c2] +=
6               sol_ext_i * fe_values[velocities]
7                           .symmetric_gradient(i, q)[c1][c2];
8       }
```

Listing 3.5: Assembly of `symgrad_u_ext_loc`

In particular the quantity `symgrad_u_ext_loc`, corresponding to $\varepsilon(\boldsymbol{u}^n_{hBDF})$, is computed in every quadrature node, is computed. The same procedure follows for all the quanti-

ties needed in order to assembly the algebraic system and this is done in the loop on the quadrature nodes through the function `solve_time_step` mentioned in Section 3.1.

- A function that computes the viscosity based on the non-Newtonian model (Equation 2.17) had to be implemented.

```
double
compute_viscosity_carreau(const unsigned int q) const
{{{
  // For incompressible fluid we should have trace(symgrad_u_ext_loc)=0
  double trace = 0.0;
  for (unsigned int d1 = 0; d1 < dim; ++d1)
    for (unsigned int d2 = 0; d2 < dim; ++d2)
      trace += symgrad_u_ext_loc[q][d1][d2] * symgrad_u_ext_loc[q][d2][↩
          d1];

  const double gamma_dot = sqrt(2.0 * trace);

  return prm_carreau_viscosity_infinity +
         (prm_carreau_viscosity_zero - prm_carreau_viscosity_infinity) *
           std::pow(1.0 + (prm_carreau_lambda * gamma_dot) *
                          (prm_carreau_lambda * gamma_dot),
                    (prm_carreau_exponent_power_law - 1.0) / 2.0);
}
```

Listing 3.6: `compute_viscosity_carreau`

- The viscosity in every quadrature point had to be computed through the function `compute_viscosity_carreau` and stored in a std::vector.

```
    if (prm_flag_non_newtonian_model != NonNewtonianModel::None)
      {
        for (unsigned int q = 0; q < n_q_points; ++q)
          viscosity_loc[q] = compute_viscosity_carreau(q);
        if (prm_flag_output_viscosity)
          viscosity_loc_all_cells[c] = viscosity_loc;
      }
```

Listing 3.7: Computation of viscosity

This happens in the same loop on the quadrature nodes as `symgrad_u_ext_loc` (and every other local quantities). The variable `viscosity_loc_all_cells` is used only for post processing manners as we will explain at the end of this section.

- Every time the viscosity had to be used for the calculation of some quantities (for example in the assembly of the diffusive term Equation 3.1 and, possibly, the preconditioner), if the non-Newtonian flag is on, the viscosity computed by the non-Newtonian model is used instead of the constant Newtonian viscosity.

```
    double viscosity =
    (prm_flag_non_newtonian_model != NonNewtonianModel::None) ?
      viscosity_loc[q] :
      prm_viscosity;
```

Listing 3.8: Assignment of viscosity

Note that even this operation had to be performed in the loop on the quadrature nodes since the viscosity could be different in every node.

In order to better postprocess the output of the simulation we added the possibility to store the viscosity associated to each cell (and at every time instant) in a hdmf-h5 file that can be read by Paraview (the same file where the values of the pressure and velocity are stored).

```
1     // Project non-Newtonian viscosity at DoFs for output purposes.
2     if (prm_flag_output_viscosity == true &&
3         prm_flag_non_newtonian_model != NonNewtonianModel::None)
4       {
5         project_l2_scalar->project<std::vector<std::vector<double>>>(
6           viscosity_loc_all_cells, viscosity_fem_owned);
7
8         viscosity_fem = viscosity_fem_owned;
9       }
```

Listing 3.9: Projection of `viscosity_loc`

In every cell, this output quantity is computed as the L^2 projection on the finite element space. Every plot of the viscosity that we will show in Chapter 4 is obtained thanks to this functionality.

3.3. Womersley profile as boundary condition

As we stated in the previous sections, the second goal of this project is to implement a solver for the **Inverse Womersley Problem** in order to impose a suitable and more realistic velocity profile on the inlet surface as Dirichlet boundary condition. More precisely, starting from a given flowrate $q(t)$ along a certain time interval, let's say $[0, T]$ (T is considered as the period of the flow), the problem consists in finding a velocity profile $v_z(r, t)$ such that:

1. $\int_A v_z(r, t)\, dA = q(t) \;\; \forall t \in [0, T]$ where A is a circular section of the cylinder (in other words, the velocity profile $v_z(r, t)$ produces the given flowrate $q(t)$)

2. the velocity profile $v_z(r, t)$ generates a oscillating pressure gradient in the cylinder.

In this case $v_z(r, t)$, called "Womersley velocity profile", is the solution of Equation 2.20.
Because of the assumption of cylindrical domain required for the validity of the procedure described in Subsection 2.3.1, and, more in general, for the development of the Womersley velocity profile, we implemented this feature in order to work specifically in *TestFluidDynamicsCylinder* (the application in a cylindrical domain of the *FluidDynamics* class that we already mentioned in Section 3.1).
The difficulties in the implementation of this feature arise from the fact that the function that we need to impose as boundary condition is not separable in space and time, differently from all other Dirichlet condition that were already implemented for *TestFluidDynamicsCylinder*. Indeed, all previous Dirichlet boundary condition where handled by a class called *FlowBC*, that for a given boundary Γ_D, represent the Dirichlet boundary condition $\mathbf{u}(t, \sigma) = g(t)\mathbf{f}(\sigma)$, where $\sigma = (x, y)$

is an element of the inlet surface, $\mathbf{f}$ is the space distribution of the velocity profile (a so called SpaceFunction) and g its time evolution (a so called TimeFunction). Unfortunately, in the case of Womersley profile, as can be seen from Equation 2.24, this assumption is not satisfied (however, trough Foruer expansion, we can write the flowrate as sum of function with separable variables): for this reason it was impossible to adapt the interface of *FlowBC* to this specific case.

In order to solve this issue we decided to implement a new class able to handle this specific problem, while keeping the whole interface in the handling of boundary conditions, and how they are imposed in *TestFluidDynamicsCylinder*, as close as possible to *FlowBC*. In details, we want to impose a boundary condition of the type $\mathbf{u}(t, \sigma) = v_z(r, t)\,\mathbf{n}_z$ where $r = \sqrt{x^2 + y^2}$, $v_z(r, t)$ is the solution to the inverse Womerlsey problem and $\mathbf{n}_z$ is the entering normal direction to the inlet surface.

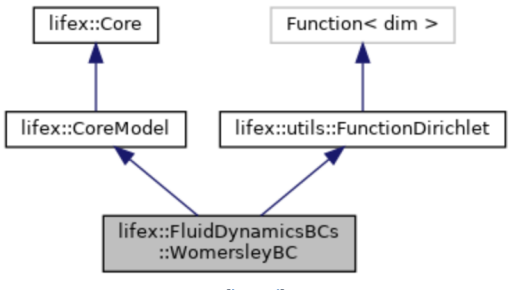


Figure 3.2: Inheritance diagram for the *WomersleyBC* class

The new class that we implemented is called *WomersleyBC* and, as we can see from Figure 3.2, is inherited from *CoreModel* and *FunctionDirichlet*. The inheritance from *CoreModel* is performed in order to use the interface for parameter handling, as we already explained in Section 3.1. *FunctionDirichlet*, instead, is a class representing an analytical function for Dirichlet boundary conditions. This is essentially a deal.II *Function* ([1]), but with a different type to prevent wrong conversions between a *FunctionDirichlet* and a *FunctionNeumann*.

```
1   class WomersleyBC : public CoreModel, public utils::FunctionDirichlet
2   {
3   public:
4   /// Constructor.
5   WomersleyBC(const std::string &  subsection,
6               const FluidDynamics &fluid_dynamics_);
7
8   /// Override of vector_value method.
9   virtual void
10  vector_value(const Point<dim> &p, Vector<double> &values) const override;
11
12  //... some utilities functions ...//
13
14  protected:
15  /// Function for csv evolution.
16  std::shared_ptr<CSVFunction> csv_function;
17
18  /// Space distribution label.
19  std::string prm_space_distribution;
```

```
20
21      /// Function for Womersley distributions.
22      std::shared_ptr<Womersley> womersley_function;
23
24      /// Scaling factor; can be used to adjust units of measure.
25      double scaling_factor;
26
27      /// Fraction of total flow assigned to this boundary.
28      double flow_repartition;
29
30      /// Chamber volumes.
31      std::vector<double> volumes;
32
33      /// Variation of chamber volumes between subsequent timesteps.
34      std::vector<double> volume_variations;
35
36      /// Reference to the fluid dynamics instance.
37      const FluidDynamics &fluid_dynamics;
38
39      /// Period of the flow rate input function
40      double period;
41    };
```

Listing 3.10: Interface of `WomersleyBC`

The interface of the class is very similar to that of *FlowBC*: the two most important members are `csv_function`, a shared pointer to *CSVfunction* that represents the flowrate $q(t)$ read from the given .csv file, and `womersley_function`, that represents the function in space we need to impose. The most relevant method is `vector_value`: a method inherited (and overridden) from the dealII function that is called wherever the boundary condition needs to be applied.

```
1  void
2    WomersleyBC::vector_value(const Point<dim> &p, Vector<double> &values) ↩
       const
3    {
4      const Point<dim> &barycenter = this->get_barycenter();
5      const double      r          = std::sqrt(p.distance_square(barycenter));
6      const double boundary_radius = std::sqrt(this->get_surface_area() / M_PI)↩
         ;
7      const std::vector<double> &re_ck =
8        csv_function->get_time_interpolation().get_re_ck();
9      const std::vector<double> &im_ck =
10       csv_function->get_time_interpolation().get_im_ck();
11     const std::complex<double> i(0, 1);
12     const unsigned int         mu = (re_ck.size() + 1) % 2;
13     const size_t               M  = (re_ck.size() - mu - 1) / 2;
14     std::complex<double>       v_n(0.0, 0.0);
15     std::complex<double>       v_n_conj(0.0, 0.0);
16     std::complex<double>       velocity(0.0, 0.0);
17     const double               t_mod = this->get_time() / period * 2 *
18                           M_PI; // t \in [0,T] mapped into t_mod \in [0,2*pi]
19
20     const double sigma0 =
21       (8.0 * fluid_dynamics.get_viscosity() * re_ck[M]) /
22       (fluid_dynamics.get_density() * M_PI * std::pow(boundary_radius, 4));
23     const double v0 =
24       (sigma0 * boundary_radius * boundary_radius) /
25       (4.0 * fluid_dynamics.get_viscosity() / fluid_dynamics.get_density()) *
26       (1 - (r / boundary_radius) * (r / boundary_radius));
```

```
27
28
29      velocity += v0;
30
31
32      for (std::size_t n = 1; n <= M; n++)
33        {
34          womersley_function->vector_value(
35            p, v_n, re_ck[M + n], im_ck[M + n], n); // compute v_n
36
37          v_n_conj = std::conj(v_n); // compute v_{-n}
38          velocity += v_n * std::exp(i * static_cast<double>(n) * t_mod) +
39                      v_n_conj * std::exp(-i * static_cast<double>(n) * t_mod);
40        }
41
42      if (mu == 1)
43        {
44          womersley_function->vector_value(
45            p, v_n, re_ck[2 * M + 1], im_ck[2 * M + 1], 2 * M + 1);
46          v_n *= 2.0 * std::cos((M + 1) * t_mod);
47          velocity += v_n;
48        }
49
50      for (unsigned int j = 0; j < dim; ++j)
51        if (utils::is_positive(boundary_radius - r))
52          values[j] = -1.0 * velocity.real() * scaling_factor *
53                      this->get_normal_vector()[j];
54        else
55          values[j] = 0.0;
56      // Pressure component.
57      values[dim] = 0.0;
58    }
```

Listing 3.11: `WomersleyBC::vector_value`

This function aims to impose the proper Dirichlet boundary condition on a single quadrature node (received as first input). The vector `values` passed as second input represents the vector (v_x, v_y, v_z, p).

Before entering into the details, we recall that the basic assumption on which this method relies is the Fourier (truncated) decomposition of the flow-rate $q(t)$, the pressure gradient $\sigma(t)$ and the velocity in the axial direction $v_z(r,t)$ (see 2.24). The function aim to solve the **Inverse Womersley problem** through the following steps:

1. Given the flow rate $q(t)$ we approximate it through a truncated Fourier expansion $q(t) = \sum_{n=-N}^{N} q_n e^{jnt}$ (this is handled by the class *CSV Function* at the beginning of each simulation).

2. First σ_0 and $v_0(r)$ are computed starting from q_0 exploiting equations (2.26) and (2.29).

3. For $n \in \{1, 2, \dots, N\}$, σ_n is obtained inverting equation (2.27) and, consequently, $v_n(r)$ is derived from (2.30).

4. $v_{-n}(r)$ is obtained as the complex conjugate of $v_n(r)$ $\forall n \in \{1, 2, \dots, N\}$.

5. Finally $v(r,t)$ is computed by summation as in (2.24).

Listing 3.11 shows the implementation of *WomersleyBC::vector_value*. The first part of the function is devoted to the initialization of the required variables, like the distance `r` of the quadrature node from the center of the inlet section, two vectors containing the real part (`re_ck`) and the imaginary part (`im_ck`) of the coefficients $\{q_n\}_{n=-N}^{N}$ associated to the Fourier expansion of the flow-rate $q(t)$ and three `std::complex`, initialized to $(0.0, 0.0)$: fixed an integer $n > 0$, `v_n` and `v_n_conj` represents respectively the Fourier coefficients v_n and v_{-n} of the velocity v_z, while `velocity` stores the value of v_z.
Lines 20-29 aim to solve step 2. computing σ_0 and $v_0(r)$ through the Poiseuille flow theory.
At lines 32-40 a for cycles loops over the possible values of the index $n \in \{1, 2, \dots, N\}$ and v_n is computed through the function *Womersley::vector_value*. In particular, this function first computes σ_n through the map $q_n \mapsto \sigma_n$ (see Equation 2.27) and subsequently computes v_n through the map $\sigma_n \mapsto v_n$ (see Equation 2.30).
After that, the coefficient v_{-n} is computed as the complex conjugate of v_n and, finally, the value of the velocity v_z is updated.
Lines 42-48 handle the case in which the number of data provided in the .csv file for the flow-rate $q(t)$ is even, let's say equal to $2M$. In this case, following the standard procedure of the Discrete Fourier Transform algorithm, the Fourier expansion of the flow-rate reads as:

$$q(t) = \sum_{n=-M+1}^{M-1} q_n e^{jnt} + 2q_M \cos\left(M\frac{2\pi}{T}t\right)$$

and, consequently, the velocity v_z is:

$$v_z(r,t) = \sum_{n=-M+1}^{M-1} v_n(r)\, e^{jnt} + 2v_M \cos\left(M\frac{2\pi}{T}t\right)$$

.
Given the value of the velocity v_z, the last part of the function is devoted to the actual imposition of the Dirichlet boundary condition $\mathbf{u} = v_z\mathbf{n}_z$ where $\mathbf{n}_z$ is the entering normal direction to the inlet surface. We stress the fact that in this case $\mathbf{n}_z$ is aligned with the z axis, consequently the boundary condition corresponds to $\mathbf{u} = v_z\mathbf{n}_z = (0, 0, v_z)$. However, the function is implemented in a more general way so that it works for any orientation of the inlet surface.

4. Tests and validation

4.1. How to run a simulation

In this section we show what the commands required to run a simulation and the main settings available in the parameter file.
The name of executable we need to run is ***lifex_test_fluid_dynamics_cylinder*** and the first step before executing it is to generate the parameter file containing all the default parameter values. This is done through the following command:

```
./executable_name VERBOSITY -g -f custom_param_file.ext
```

The `-g` flag is used to generate the parameter file with a certain verbosity that can be decreased or increased by passing the optional flag `VERBOSITY` that can be either `minimal` or `full`. Also the `-f` flag is optional and it allows the user to customize the parameter file basename. If `-f` is not used, a default name is assigned.
Let us inspect the most important features that the user can set through the parameter file:

```
3  subsection Test fluid dynamics cylinder
4    subsection Boundary conditions
5      # Length of the cylinder [m], to impose an analytical ALE displacement.
6      set Length of the cylinder                        = .0450
7
8      # Radius of the inlet boundary [m].
9      set Radius of the inlet boundary                  = 0.0045
10
11     # Displacement factor [m].
12     set Radial displacement factor                    = 100
13
14     # Dirichlet conditions impose a parabolic or flat velocity profile;
15     # Neumann conditions a uniform stress.
16     # Available options are: Dirichlet | Neumann.
17     set Type of inlet condition                       = Dirichlet
18
19     # If set to true, the velocity tangential to the inlet boundary is set
20     # to zero for Neumann boundary conditions, resulting in practice in a
21     # mixed Dirichlet-Neumann condition.
22     set Constrain tangential flux for Neumann inlet = true
```

Listing 4.1: parameter file

In the first part the user can set the length and radius of the cylinder, together with the type of boundary condition. Note that, in case of Neumann boundary condition, if `set Constrain tangential flux for Neumann inlet` is `true` a homogeneous Dirichlet boundary condition is adopted in the direction normal to the inlet surface, while the Neumann boundary condition is preserved in the tangential direction. This option can be used to set a boundary condition different from the natural one associated to that a certain formulation (see table 2.1).

```
subsection Dirichlet inlet
      # A multiplicative scaling factor.
      set Scaling factor     = 1.0

      # The type of evolution in time the flow will have.
      # Available options are: CSV function | Constant | Pulsatile | Ramp.
      set Time evolution     = Constant

      # The type of space distribution the flow velocity will have.
      # Available options are: Parabolic | Uniform | Womersley.
      set Space distribution = Parabolic
```

Listing 4.2: parameter file

In case of Dirichlet boundary condition the user can select a time evolution for the inflow velocity field (which non-zero only in the direction parallel to the cylinder axis). This can be:

- `Constant`: the velocity field prescribed on the inlet surface is time independent. The user has to specify the value of the velocity (in case of uniform flow) or the flow rate.
- `Pulsatile`: the velocity oscillates with a period that must be specified by the user, together with the maximum and the minimum value of the velocity (or flow rate).
- `CSV function`: the velocity (or flow rate) is prescribed by a .csv file and it is supposed to be periodic with period equal to the time interval reported in it. In this case the user must specify the name of the .csv file (which must be placed in the same folder of the executable file) and the interpolation method that must be used to extract the velocity (or flow rate) function.
- `Ramp`: the velocity (or flow rate) starts from a certain value v_1 and then moves linearly to a final value v_2 in a certain time interval δt. The user must specify the initial time of the ramp, together with v_1, v_2 and δ_t.

Moreover a space evolution must be specified. The available options are:

- `Uniform`: a flat velocity profile is imposed at every time step, suitable for big vessels.
- `Parabolic`: a parabolic velocity profile is prescribed at every time step, suitable for vessels with small radius.
- `Womersley`: the Womersley velocity profile is imposed at every time step. This option is a new feature introduced with this project (see Subsection 2.3.1).

In case of Neumann boundary condition a time evolution for the pressure can be selected. The options are analogous to those described above.

```
subsection Non-Newtonian model
    # Non-Newtonian model to be used.
    # Available options are: None | Carreau.
    set Non-Newtonian model     = None

```

```
326     # Viscosity zero [Pa * s].
327     set Viscosity zero          = 5.6e-2
328
329     # Viscosity infinity [Pa * s].
330     set Viscosity infinity      = 3.45e-3
331
332     # Exponent power law [1].
333     set Exponent power law      = 0.3568e0
334
335     # Lambda [s].
336     set Lambda                  = 3.313e0
```

Listing 4.3: parameter file

There is also a section that contains all the parameters related to the Carreau model and the fluid under study. Moreover it can be specified which model should be used (Newtonian or non-Newtonian) and the type of formulation for the diffusion term (see (2.7)).
Many other features can be set through the parameter file. In particular, in this project we made the following choices:

- A cylindrical mesh with hexagonal finite elements is used.
- For both the velocity and pressure Finite Elements spaces we set $\mathbf{V_h} = [\mathbb{P}^1(\Omega_h)]^3$ and $Q_h = \mathbb{P}^1(\Omega_h)$. Since this couple of spaces are not Inf-Sup stable, we need a stabilization term and for all the experiments we used the SUPG method (rigorously SUPG-PSPG).
- The non-linear term is always treated as explicit (see the end of section 2.3).
- Regarding the time discretization, a BDF scheme of order 1 is employed.

After the parameters are set up, the user can run a simulation with the following command:

```
./executable_name -f custom_param_file.ext -o ./results/
```

The `-f` flag is now used to select the desired parameter file, while the `-o` flag is used to specify in which folder the solution files should be stored.
A simulation produces as output the following files:

- One .h5 file and one .xdmf file for each time step which can be analyzed through a specific software (for instance we used **Paraview**) to see the evolution in space and time of velocity and pressure.
- One .prm file which is simply a copy of the parameter file selected.
- One .csv file that contains the values of the velocity flow rate at each time step.

Many information are displayed on the terminal (or in a separate file if the user prefers to) during the execution, such as the Finite Elements spaces and stabilization employed and number of iterations needed by the iterative solver in each time-step.

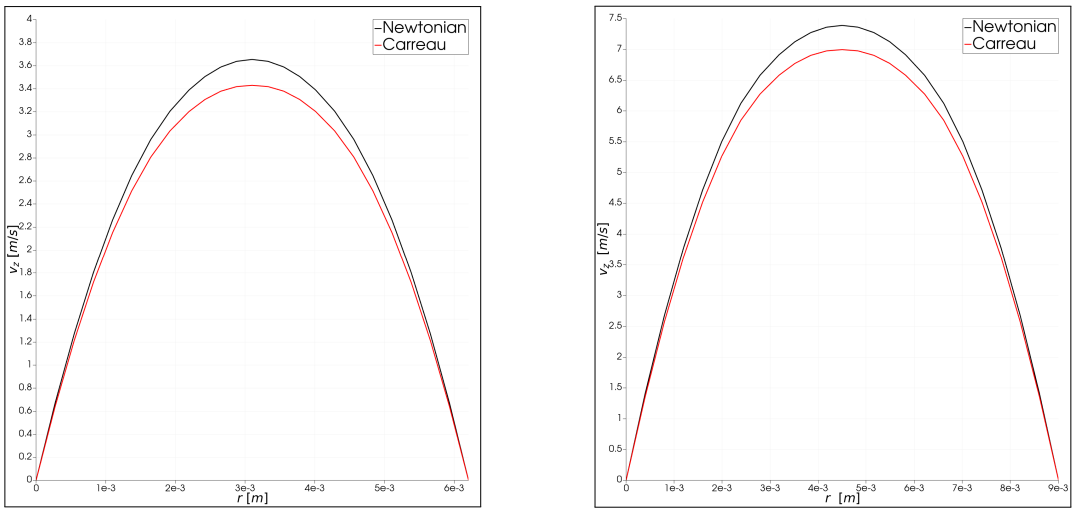
Finally we mention that, in order to execute multiple simulations sequentially without changing the parameter file every time, we implemented a .bash script, reported in details in Appendix A. Here the user can select multiple options (like a set of radii and lengths of the cylinder, types of boundary conditions etc...), then the script will run an instance for all the possible combinations of the parameters imposed. After the end of each simulation all the output files are automatically moved and renamed in a suitable way, simplifying the analysis of the results.

4.2. Tests and validation

In this section we compare the results obtained solving problem (2.11) first assuming a Newtonian flow and then integrating the Carreau model for the modeling of the Non-Newtonian viscosity. In order to validate the model implementation, some reference results are taken by [21], where the same problem is solved in a cylindrical domain but using a different method.
A grid independent study highlighted that the results shown in this section are still sensitive to further refinements, which on the other hand would affect dramatically the execution time, making a simulation campaign unfeasible. However the results reported here are still reliable and in accordance with the ones reported in [21].

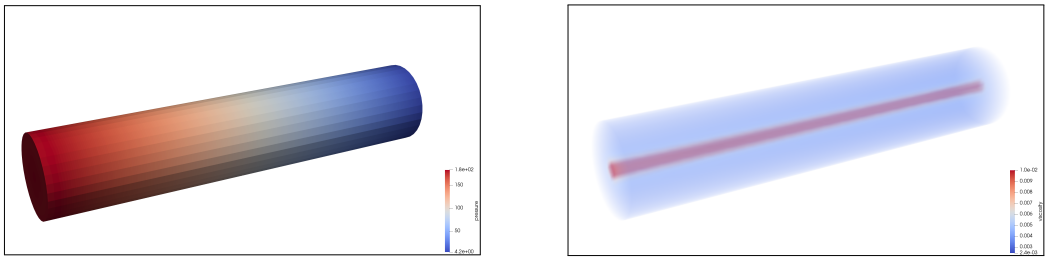
4.2.1. Steady-state blood flow

To assess the effectiveness of the Carreau model we first compare the results obtained imposing a steady-state flow. In particular, following the procedure illustrated in [21], we considered a cylindrical domain and two different radii equal to $R_1 = 0.0031m$ and $R_2 = 0.0045m$ respectively. We imposed a mixed boundary condition corresponding to a constant pressure drop equal to $6000Pa/m$ (Figure 4.2a).



(a) $r = 0.0031m$ (b) $r = 0.0045m$

Figure 4.1: Velocity profiles for the steady case.



(a) Pressure (b) Viscosity

Figure 4.2: Pressure and viscosity distribution in the domain for the steady case.

In Figure 4.1 we can see a comparison between the velocity profiles associated to the Newtonian and non-Newtonian models in both cylinders. The 2D representation of the velocity profiles exploits the radial symmetry.
As we could expect the resulting flow is parabolic (Poiseuille flow) and we observe that the non-Newtonian flow, which in general leads to a greater value of the viscosity, is characterized by a lower flow rate.
The details of the experiments such as flux and wall shear stress (WSS= $\mu \frac{\partial u_r}{\partial r}|_{r=R}$) are reported in tables 4.1-4.2 (here "Ratio flux" is computed as (Newtonian flux - Carreau flux) / (Newtonian flux)). The differences in the absolute values of the fluxes with respect to the reference results can be caused by many reasons: a different projection method, a different discretization both in space and time, a different solver etc., however it is important to notice that the diversity between Newtonian and non-Newtonian flows, together with the values of the WSS, are in complete accordance with the reference results.
Finally, Figure 4.2b shows the value of the viscosity in the volume of the domain: the maximum is reached in the center, where the velocity gradient is at its minimum value, and the minimum is reached near the boundary, where the velocity gradient is at its maximum value. Thus, the result is coherent with 2.17.

4.2.2. Pulsatile blood flow

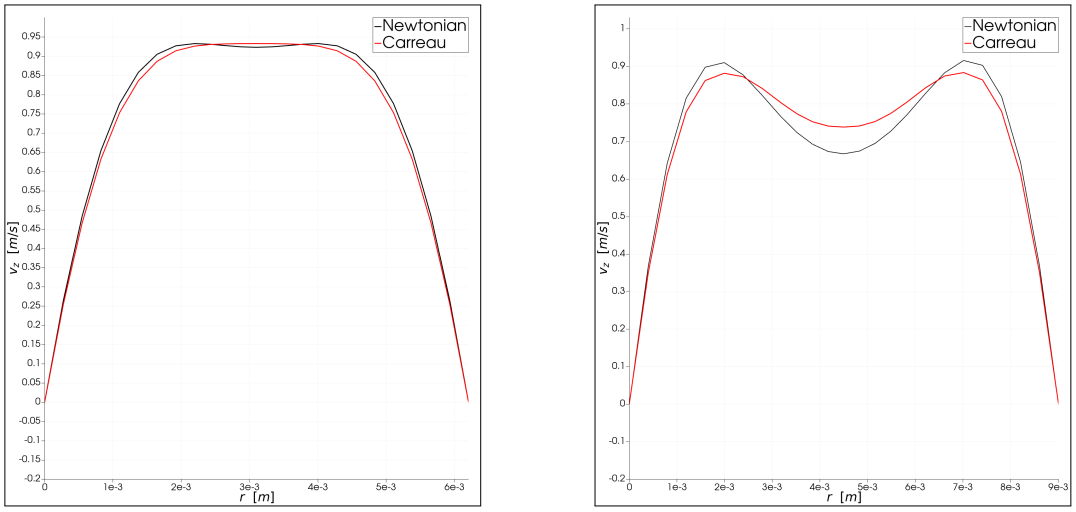
The second validation test we have performed consists in solving the same problem of the previous section, this time imposing a pulsatile pressure gradient. In details the momentum equation can be written as Equation 2.20 where the amplitude of the oscillatory pressure gradient is taken to be equal to $A = 6000 Pa/m$ (corresponding to $45mm$ mercury height per meter tube length) while the angular frequency is taken as $n = 2\pi f$ with $f = 1.2Hz$ (corresponding to the normal pulse frequency of 72 beats per minute).

	Reference	life^x
Newtonian flux (m^3/s)	$6.30e-05$	$5.72e-05$
Carreau flux (m^3/s)	$5.98e-05$	$5.43e-05$
Ratio flux	5.08%	5.03%
Newtonian WSS (Pa)	9.3	9.31
Careau WSS (Pa)	9.3	9.26

Table 4.1: Comparison with the reference results [21]. $r = 0.0031cm$

	Reference	life^x
Newtonian flux (m^3/s)	$2.80e-04$	$2.49e-04$
Carreau flux (m^3/s)	$2.69e-04$	$2.38e-04$
Ratio flux	4.04%	4.30%
Newtonian WSS (Pa)	13.5	13.91
Careau WSS (Pa)	13.5	13.87

Table 4.2: Comparison with the reference results [21]. $r = 0.0045cm$

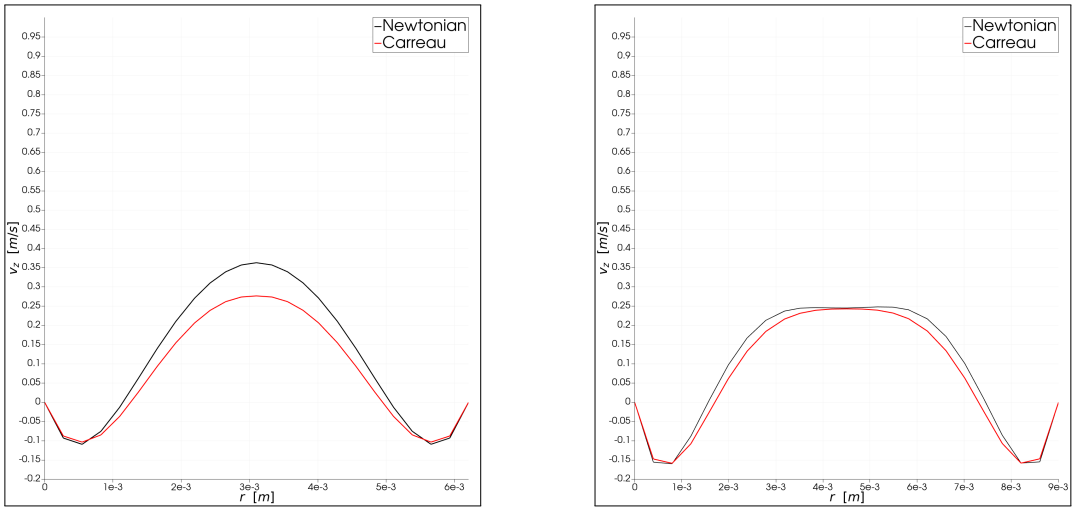


(a) $r = 0.0031m$ (b) $r = 0.0045m$

Figure 4.3: Velocity profiles corresponding to the maximum flow rate in the pulsatile case.

Figure 4.3 shows the velocity profiles obtained as solutions of both the Newtonian and non-Newtonian models corresponding to the maximum value of the flow rate in the two cylinders. We can observe the typical shape of the Womersley profile in the cylinder with radius $r = 0.0045m$, while for the smaller radius $r = 0.0031m$ we can see a sort of transition to a parabolic profile. The two models differ mainly in the central region of the cylinder and the difference becomes more evident as the radius decreases.

It is interesting to compare also the velocity profiles associated to the minimum values of the flow rate (in the positive time cycle, i.e. the lowest positive value of the flow rate). These are reported in Figure 4.4, in which we can notice that the non-Newtonian velocity profiles are slightly flattened with respect to the Newtonian profiles. Similar velocity profiles have been obtained in [12] for the 2D blood flow in the carotid artery using the Carreau-Yasuda model.



(a) $r = 0.0031m$ (b) $r = 0.0045m$

Figure 4.4: Velocity profiles corresponding to the minimum flow rate in the pulsatile case.

	Reference	life^x
Newtonian max flux (m^3/s)	$1.77e-5$	$1.86e-05$
Carreau max flux (m^3/s)	$1.73e-5$	$1.81e-05$

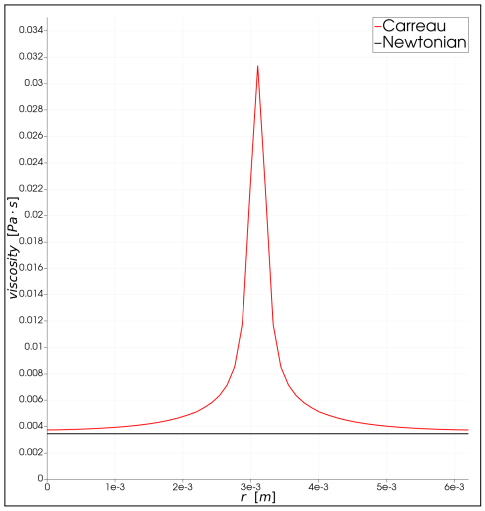
Table 4.3: Comparison of the max flow rate with the reference results [21]. $r = 0.0031cm$

	Reference	life^x
Newtonian max flux (m^3/s)	$4.10e-5$	$4.23e-05$
Carreau max flux (m^3/s)	$4.06e-5$	$4.14e-05$

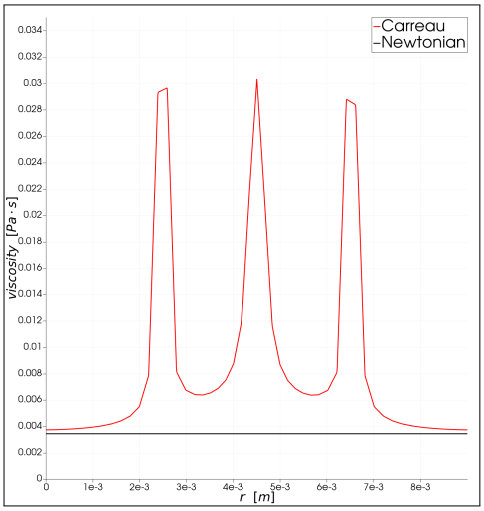
Table 4.4: Comparison of the max flow rate with the reference results [21]. $r = 0.0045cm$

As for the steady-state case tables 4.3-4.4 summarize the comparison of the results obtained using the life^x solver against the ones reported in [21].
The viscosity distribution of both cylinders for the maximum and minimum flowrates are shown in Figure 4.5 and Figure 4.6 respectively. The peaks in the central region correspond to the lower velocity gradients, while the satellite peaks in Figure 4.5b and Figure 4.6a are connected with the local minima of the velocity gradient presented in Figure 4.3b and Figure 4.4a.

Its is worth noticing that, if an oscillating pressure gradient is imposed, the Womerley velocity profiles naturally develop in a cylindrical domain even if a Neumann or mixed boundary condition is imposed. This fact confirms the importance, for real applications, of introducing the possibility to impose a Womersley velocity profile as Dirichlet boundary condition.

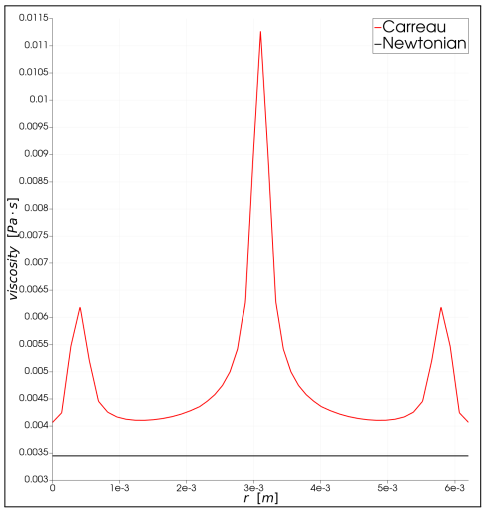


(a) $r = 0.0031m$

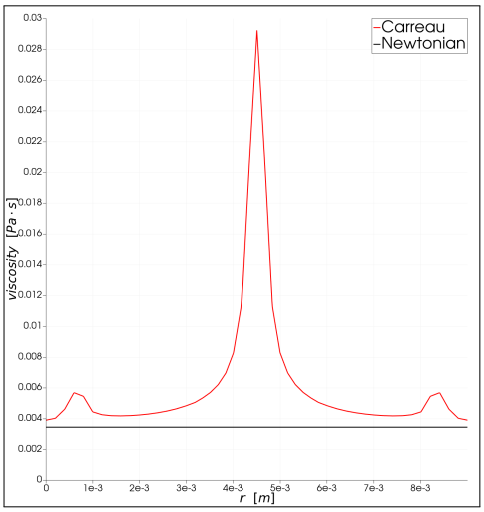


(b) $r = 0.0045m$

Figure 4.5: Comparison between the Newtonian and the non-Newtonian viscosity at the maximum flow rate.



(a) $r = 0.0031m$



(b) $r = 0.0045m$

Figure 4.6: Comparison between the Newtonian and the non-Newtonian viscosity at the minimum flow rate.

4.3. Experimental results Womersley

In this section we show some results to assess the effectiveness of the second new feature we introduced in life^x: a solver for the inverse Womersley problem in order to impose a more proper and realistic velocity profile in the inlet surface as Dirichlet boundary condition.
The validation consists in verifying the two following aspects:

1. the flowrate computed from the velocity profiles obtained by the solver corresponds to the original one given as input,

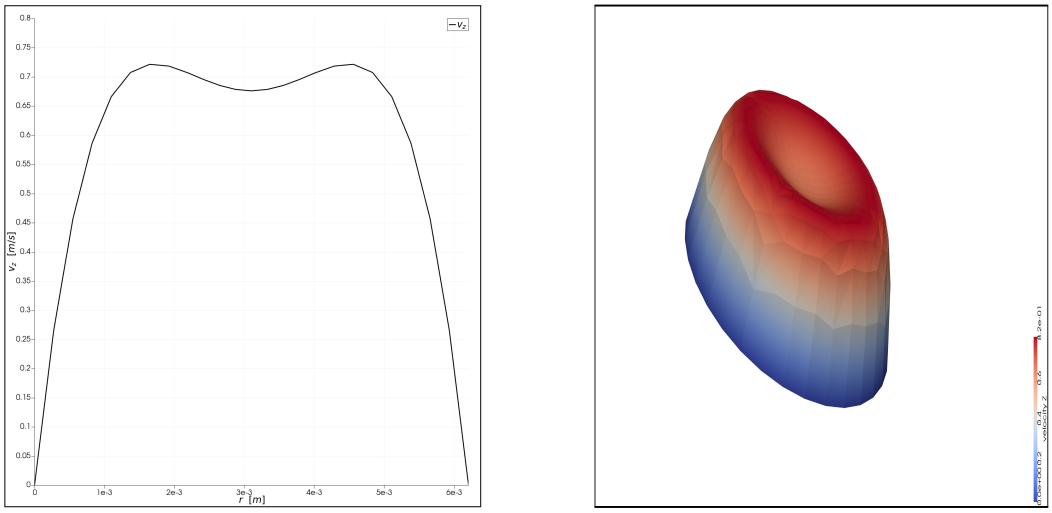
2. the shape of the velocity profiles at the inlet (obtained here through an analytical procedure) are in accordance with the ones obtained numerically in the previous section where we

imposed an oscillating pressure gradient (Neumann boundary condition).

All the results in this section have been obtained on a cylinder with radius $r = 0.0031m$ employing a Newtonian model.
We stress that, since we are interested in the velocity profiles at the inlet, there are no differences between Newtonian and non-Newtonian models. Thus, a Newtonian model is preferable since it requires a lower computational effort.

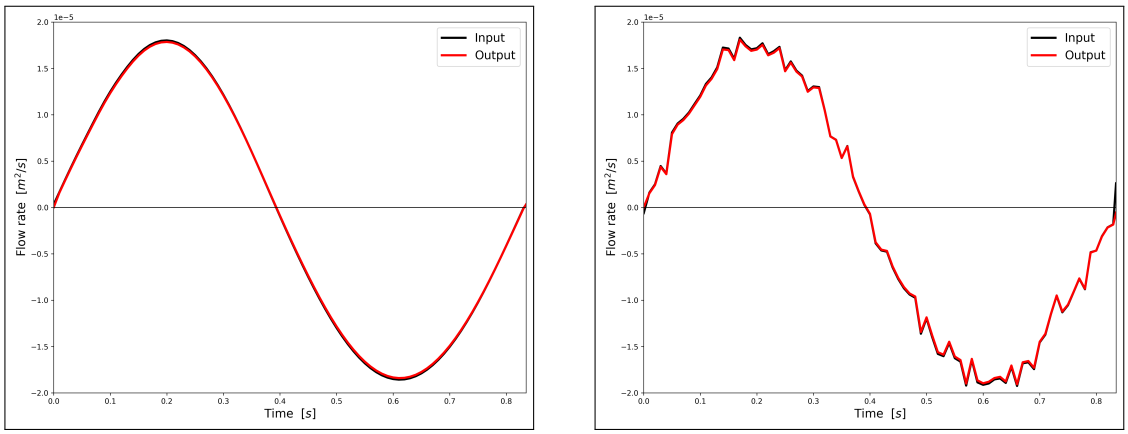
First, in order to generate a .csv file containing a flowrate adequate for our purpose, we run an instance imposing, as a Neumann boundary condition, an oscillating pressure gradient with amplitude $A = 6000Pa/m$ (coherently with the previous section). This simulation produces a .csv file with the resulting flow-rate over time $q(t)$, which is used as a Dirichlet boundary condition for the second simulation, this time selecting the options `Womersley` as space function and `CSV function` as time function for the inlet. We recall that, as we explain in Subsection 2.3.1, the first step for solving the inverse Womersley problem is to expand the flowrate by means of Fourier series. Thus, among all the possible interpolation methods available in life^x, the Fourier series mode must be selected in order to solve the inverse Womersley problem.
Finally we compared the .csv file obtained as output with the one given as input and, using *Paraview*, we obtained some plots of the velocity profiles in the inlet section to verify the consistency of their shape.
In Figure 4.8a we report a comparison between the input flow-rate (passed as a parameter through a .csv file) and the output flow-rate, i.e. the flow-rate generated by the solution (obtained integrating the velocity field according to a given quadrature rule) and we can see that the flow-rate is successfully recovered.
Moreover, in Figure 4.7 we report the velocity profiles corresponding to the time instant $t = 0.14s$. The shape is conforming to those predicted by the theory (see Subsection 2.3.1).
There is one more aspect that we have to take into account: for our simulation we employed a sinusoidal flow-rate with period $T = 1/f$ where $f = 1.2Hz$ (corresponding to 72 heart beats per minute) but, in real application, the blood flow-rate is not sinusoidal and it can also present some irregularities (due to measurement errors but they can also be associated to physiological or pathological reasons). In light of this, we introduced a random noise in the input flow rate and we run a new instance passing, again as .csv input file, a new flowrate $\widetilde{q}(t)$ defined at each time-step t_k as $\widetilde{q}(t_k) = q(t_k) + \delta_k$ where δ_k is sampled from a Gaussian distribution with null mean and standard deviation equal to 10^{-6}. From Figure 4.8b we can see that the perturbation does not affect the precision in reconstructing the input flowrate.
Finally, in Figure 4.9 we can see the two velocity profiles obtained at time $t = 0.14s$ in both cases with and without noise. The velocity profile associated to the perturbed flowrate is overall greater than other one and this is coherent with the fact that $\widetilde{q}(0.14) > q(0.14)$. It is very important to notice that, even after introducing a random noise, the resulting velocity profiles are still smooth and the shapes are in accordance with the expectations. This result is a confirm that the procedure we have implemented in life^x is not only able to solve the inverse Womersley problem but it is also robust to small perturbations which is a key feature for medical applications, where

the measurement of the blood flow-rate is often an easier and preferable procedure compared to the measurement of velocity profiles.



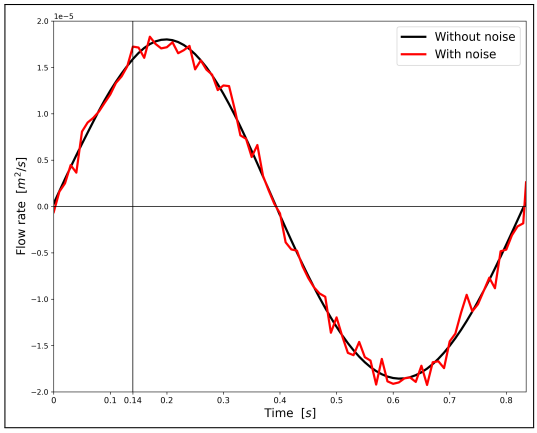
(a) 2D representation (b) 3D representation on a circular section

Figure 4.7: Reconstructed velocity profile at time $t = 0.14s$.

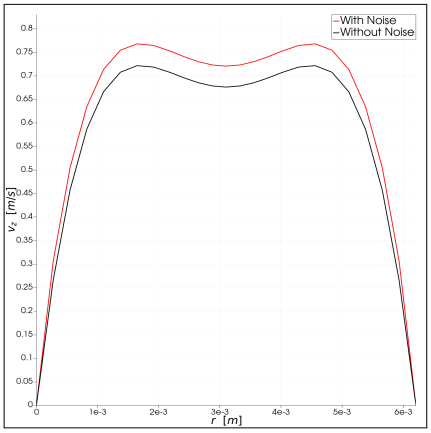


(a) Without noise (b) With noise

Figure 4.8: Comparison between input and reconstructed flowrates.



(a) Flowrates with and without noise



(b) Velocity profiles at $t = 0.14s$

Figure 4.9: Comparison between velocity profiles with and without noise.

A. A bash script for sequential executions

```
#!/bin/bash

declare -a pressure
declare -a radius
declare -a model_flag
declare -a boundary_condition
declare -a velocity_flux
declare -a final_times
declare -a time_steps
declare -a refinement
declare -a debug_viscosity
declare -a radius_to_length_scale_factor
declare -a density
declare -a vel_degree
declare -a p_degree
declare -a stabilization
declare -a inlet_type_time
declare -a inlet_type_space
declare -a interpolation

# Here you can put multiple options: nested loops will run all the possible
# combinations
pressure=('0.00')                           # Inlet pressure imposed for
                                            # Neumann boundary condition
radius=('0.0031' '0.0045')                  # Radius of the cylinder
model_flag=('Newtonian' 'Non_Newtonian')    # Newtonian | Non_Newtonian
boundary_condition=('Dirichlet')            # Dirichlet | Neumann | mixed
velocity_flux=('1.0e-10')                   # Velocity flux for Dirichlet
                                            # boundary condition
final_times=('3.0')                         # Final time simulated
time_steps=('0.01')                         # Timestep of the simulation
refinement=('3')                            # Mesh refinement

# Here you can set only one value for each parameter
debug_viscosity='true'                      # true | false (if true viscosity
                                            # is available in paraview)
radius_to_length_scale_factor=10            # Ratio between length and radius
density=1000                                # Density (kg/m^3)
vel_degree=1                                # Dimension of the fe space for
                                            # velocity
p_degree=1                                  # Dimension of the fe space for
                                            # pressure
stabilization='SUPG'                        # None | SUPG | VMS-LES
                                            # | Sigma LES
inlet_type_time='CSV function'              # Constant | Pulsatile | Ramp
                                            # | CSV function
inlet_type_space='Womersley'                # Parabolic | Uniform | Womersley
interpolation='Fourier series'              # Linear | Cubic spline
                                            # | Fourier series

# Path to the parameters file
file_path= path/to/parameters/file/parameters_file.prm

# Path to folders "Newtonian", "Non_Newtonian" and "Comparison"
dest_path=/path/where/to/store/solutions

```

```
# Path to folders "Newtonian", "Non_Newtonian" and "Comparison" inside
# "Womersley"
dest_path_Womersley=/path/where/to/store/solutions

# Path to file "lifex_test_fluid_dynamics_cylinder"
run_path=/path/to/executable/file

# Additional title for the folders where the solutions will be stored, it
# will appear at the end after a "-"
mark_path= to_characterize_solutions

# Set preliminary parameters (no multiple choice on these)
sed -i "s/    set Debug flag                  =.*/    set Debug flag ↩
                    = $debug_viscosity/" $file_path
sed -i "s/    set Density                     =.*/    set Density ↩
                        = $density/" $file_path
sed -i "s/    set Velocity FE space degree =.*/    set Velocity FE space ↩
    degree = $vel_degree/" $file_path
sed -i "s/    set Pressure FE space degree =.*/    set Pressure FE space ↩
    degree = $p_degree/" $file_path
sed -i "s/    set Model                                 =.*/    set Model ↩
                                  = $stabilization/" $file_path
sed -i "s/      set Time evolution     =.*  # Inlet time evolution/      set ↩
    Time evolution     = $inlet_type_time    # Inlet time evolution/" ↩
    $file_path
sed -i "s/      set Space distribution = .* # Inlet space function/      set ↩
    Space distribution = $inlet_type_space   # Inlet space function/" ↩
    $file_path
sed -i "s/        set Interpolation               =.*  #Dirichlet inlet ↩
    interpolation/        set Interpolation               = $interpolation  #↩
    Dirichlet inlet interpolation/" $file_path

# Set the refinement level
for ref in ${refinement[@]}
do

sed -i "s/    set Number of refinements    =.*/    set Number of refinements ↩
      = $ref       # default: 0/" $file_path

   for model in ${model_flag[@]}
   do

   # Check the validity of the flag debug_viscosity
   if [[ $debug_viscosity != 'true' && $debug_viscosity != 'false' ]]
   then
     echo 'ERROR: non valid viscosity flag'
     break
   fi

   # Set the model_flag for the model (Newtonian or Carreau)
     if [ $model == 'Newtonian' ]
     then
       sed -i "s/    set Non-Newtonian model flag.*/    set Non-Newtonian ↩
          model flag   = false/" $file_path
     elif [ $model == 'Non_Newtonian' ]
     then
       sed -i "s/    set Non-Newtonian model flag.*/    set Non-Newtonian ↩
          model flag   = true/" $file_path
       sed -i "s/    set Debug flag                  =.*/    set Debug flag ↩
                           = $debug_viscosity/" $file_path
     else
```

```
103       echo 'ERROR: non valid model flag'
104       break
105     fi
106
107   # Set the radius and length
108     for r in ${radius[@]}
109     do
110       length_cyl=$(expr $radius_to_length_scale_factor*$r | bc)
111       sed -i "s/    set Length of the cylinder                    =.*/    set Length of the cylinder                    = $length_cyl/" $file_path
112       sed -i "s/      set Length          =.*/      set Length          = $length_cyl/" $file_path
113       sed -i "s/    set Radius of the inlet boundary              =.*/    set Radius of the inlet boundary              = $r/" $file_path
114       sed -i "s/      set Radius          =.*/      set Radius          = $r/" $file_path
115
116   # Set the boundary conditions
117       for my_bc in ${boundary_condition[@]}
118       do
119         if [ $my_bc == 'Dirichlet' ]
120         then
121           sed -i "s/    set Type of inlet condition                   =.*/    set Type of inlet condition                   = Dirichlet/" $file_path
122
123         elif [ $my_bc == 'Neumann' ]
124         then
125           sed -i "s/    set Type of inlet condition                   =.*/    set Type of inlet condition                   = Neumann/" $file_path
126           sed -i "s/    set Constrain tangential flux for Neumann inlet =.*/    set Constrain tangential flux for Neumann inlet = false/" $file_path
127
128         elif [ $my_bc == 'mixed' ]
129           then
130           sed -i "s/    set Type of inlet condition                   =.*/    set Type of inlet condition                   = Neumann/" $file_path
131           sed -i "s/    set Constrain tangential flux for Neumann inlet =.*/    set Constrain tangential flux for Neumann inlet = true/" $file_path
132         else
133           echo 'ERROR: non valid boundary condition'
134           break
135         fi
136
137         # Set the final time
138         for final_t in ${final_times[@]}
139           do
140             sed -i "s/    set Final time              =.*/    set Final time              = $final_t/" $file_path
141
142           # Set the timestep
143           for t_step in ${time_steps[@]}
144           do
145             sed -i "s/    set Time step               =.*/    set Time step               = $t_step/" $file_path
146
```

```
147             # Set velocity flux if Dirichlet BC
148             if [ $my_bc == 'Dirichlet' ]
149             then
150 
151                  if [[ $inlet_type_space == 'Womersley' ]]
152                    then
153                       (cd $run_path; ./lifex_test_fluid_dynamics_cylinder -f↩
                              parameters_file.prm -o $dest_path_Womersley/↩
                          $model/r$r-$my_bc-s$final_t-vel$flux-$mark_path)
154                       cp $dest_path_Womersley/$model/r$r-$my_bc-s$final_t-↩
                          vel$flux-$mark_path/fluid_dynamics.csv ↩
                          $dest_path_Womersley/Comparison/Csv
155                       mv $dest_path_Womersley/Comparison/Csv/fluid_dynamics.↩
                          csv $dest_path_Womersley/Comparison/Csv/$model-r$r↩
                          -$my_bc-s$final_t-vel$flux-$mark_path.csv
156                    else
157                 for flux in ${velocity_flux[@]}
158                 do
159                   sed -i "s/        set Value =.* flow/        set Value = ↩
                         $flux # flow/" $file_path
160 
161                   (cd $run_path; ./lifex_test_fluid_dynamics_cylinder -f ↩
                         parameters_file.prm -o $dest_path/$model/r$r-$my_bc-↩
                         s$final_t-vel$flux-$mark_path)
162                   cp $dest_path/$model/r$r-$my_bc-s$final_t-vel$flux-↩
                         $mark_path/fluid_dynamics.csv $dest_path/Comparison/↩
                         Csv
163                   mv $dest_path/Comparison/Csv/fluid_dynamics.csv $dest_path↩
                         /Comparison/Csv/$model-r$r-$my_bc-s$final_t-vel$flux-↩
                         $mark_path.csv
164                 done
165                    fi
166 
167             # Set pressure otherwise
168             else
169               for p in ${pressure[@]}
170               do
171                 sed -i "s/        set Value =.* Inlet pressure/        set ↩
                       Value = $p   # Inlet pressure/" $file_path
172 
173                  (cd $run_path; ./lifex_test_fluid_dynamics_cylinder -f ↩
                       parameters_file.prm -o $dest_path/$model/r$r-$my_bc-↩
                       s$final_t-p$p-$mark_path)
174                  cp $dest_path/$model/r$r-$my_bc-s$final_t-p$p-$mark_path/↩
                       fluid_dynamics.csv $dest_path/Comparison/Csv
175                  mv $dest_path/Comparison/Csv/fluid_dynamics.csv $dest_path/↩
                       Comparison/Csv/$model-r$r-$my_bc-s$final_t-p$p-↩
                       $mark_path.csv
176               done
177             fi
178           done
179         done
180       done
181     done
182   done
183 done
```

Listing A.1: Script to run multiple instances sequentially

Bibliography

[1] The deal.ii finite element library. https://www.dealii.org

[2] lifex. https://lifex.gitlab.io

[3] Africa P.C., Piersanti R., Fedele M., Dede' L., , Quarteroni A.: lifex – heart module: a high-performance simulator for the cardiac function (2022)

[4] Berselli L. C., Miloro P., Menciassi A. and Sinibaldi E: Exact solution to the inverse womersley problem for pulsatile flows in cylindrical vessels, with application to magnetic particle targeting. Applied Mathematics and Computations **219**, 5717–5729 (2013)

[5] Bowman, F.: Introduction to bessel functions (1958)

[6] Cho Y.I., Kensey K.R. : Effects of the non-newtonian viscosity of blood on flows in a diseased arterial vessel. part 1: Steady flows. Biorheology pp. 241–262 (1991)

[7] E. Nader, S. Skinner, M. Romana et al.: Blood rheology: Key parameters, impact on blood flow, role in sickle cell disease and effects of exercise. Frontiers in Physiology (2019)

[8] G. P. Galdi and A. M. Robertson: The relation between flow rate and axial pressure gradient for time-periodic poiseuille flow in a pipe. Journal of Mathematics and Fluid Mechanic **7**, 215–223 (2005)

[9] Gill R.W.: Measurement of blood flow by ultrasound: accuracy and sources of error. Ultrasound in Medicine and Biology pp. 625–641 (1985)

[10] Guerciotti B., Vergara C.: Computational comparison between newtonian and non-newtonian blood rheologies in stenotic vessels (2016)

[11] J. Biasetti, T. C. Gasser, M. Auer, U. Hedin, F. Labruto: Hemodynamics of the normal aorta compared to fusiform and saccular abdominal aortic aneurysms with emphasis on a potential thrombus formation mechanism. Annals of Biomedical Engineering (2009)

[12] J. Boyd, J. M. Buick, S. Green: Analysis of the casson and carreau-yasuda non-newtonian blood models in steady and oscillatory flow using the lattice boltzmann method. Physics of Fluids (2007)

[13] J. Chen, X. Y. Lu, W. Wang: Non-newtonian effects of blood flow on hemodynamics in distal vascular graft anastomoses. Journal of Biomechanics **39**, 1983–1995 (2006)

[14] Kefayati S., Holdsworth D. W., Poepping T. L.: Turbulence intensity measurements using particle image velocimetry in diseased carotid artery models: effect of stenosis severity, plaque eccentricity, and ulceration. Journal of Biomechanics **47**, 253–364 (2014)

[15] Mendieta J.B, Fontanarosa, Wang J., Paritala P.K., McGahan T., Lloyd T,Li1 Z.: The importance of blood rheology in patient-specific computational fluid dynamics simulation of stenotic carotid arteries (2020)

[16] N. Bessonov, A. Sequeira, S. Simakov, Y. Vassilevskii, V. Volpert: Non-newtonian effects of blood flow on hemodynamics in distal vascular graft anastomoses. JMath. Model. Nat. Phenom. **11**, 1–25 (2016)

[17] Parolini N.: Course notes of computational fluid dynamics, politecnico di milano (2020)

[18] Quarteroni A., Grandperrin G., Deparis S.: Parallel preconditioners for the unsteady navier–stokes equations and applications to hemodynamics simulations. Computers & Fluids (2013)

[19] Quarteroni A., Manzoni A., Vergara C.: The cardiovascular system: Mathematical modelling, numerical algorithms and clinical applications. Acta Numerica pp. 365–590 (2017)

[20] Salsa S.: Partial differential equations in action - from modelling to theory (2008)

[21] Tabakova S., Nikolova E., Radev S.: Carreau model for oscillatory blood flow in a tube. AIP Conference Proceedings pp. 336–343 (2014)

[22] Tezduyar T. and Sathe S.: Stabilization parameters in supg and pspg formulations. Journal of Computational and Applied Mechanics **4**, 71–88 (2003)

[23] van de Vosse F.N. : Cardiovascular fluid mechanics (1991)

[24] Womerlsey J.R.: Method for the calculation of velocity, rate of flow and viscous drag in arteries when the pressure gradient is known. The journal of physiology pp. 553–563 (1955)