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

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 increasing interest in the understanding of cardiovascular diseases.

Blood is mainly composed of plasma, which consists mostly of water, and blood cells whose diameter is approximately 10^{-3} cm. Most arteries and veins have a diameter of 10^{-1} cm, thus way larger than the blood cells size. Therefore, 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 [27, 18] and becomes of fundamental importance in small vessels like the capillaries [15, 24]. When treated as a non-Newtonian fluid, the viscosity of blood varies with the shear rate and this increases the complexity of the numerical model.

In addition to the fluid rheology, the mathematical model describing blood flow has to be completed with suitable boundary conditions. Clinical data are often used to provide boundary conditions, for example Doppler ultrasound may be used to estimate the flow rates at different times [14]. However the flow rate cannot be employed directly as a Dirichlet boundary condition: we need to impose a value for the velocity in every point of the inlet and, for this reason, we need to model a velocity profile evolution. For vessels with a large diameter (e.g. the ascending aorta) a uniform profile velocity imposition is considered a good approximation, while for vessels with a very small diameter (e.g. capillaries) a parabolic profile is often used [27]. However for a pulsatile flow in a cylindrical domain, the velocity profile in a section is given by the "Womersley profile", described by J.R. Womersley in his work [33]. The uniform and parabolic profiles that we have just mentioned are limit cases respectively for big and small values of the Womersley number: a dimensionless parameter whose square represents the ratio between transient inertial force and viscous force. However, for intermediate values of the Womersley number (that can be found, for example, in small arteries or veins) these approximations are not appropriate and 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**$^{\mathbf{x}}$: a high-performance Finite Element C++ library mainly focused on mathematical models and numerical methods for cardiac applications. Our aim was to deepen the study of the 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 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 section the Navier-Stokes equations, together with the associated weak formulation and Galerkin projection.

2.1. The incompressible Navier-Stokes equations

Navier-Stokes equations constitute the main mathematical model to describe the motion of viscous fluids. 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. 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. The Navier-Stokes equations for incompressible fluid in convective form read as follows:

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

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 $T \in \mathbb{R}^+$ and the boundary $\partial\Omega$ is partitioned in a Dirichlet boundary Γ_D and a Neumann boundary Γ_N such that $\Gamma_D \cup \Gamma_N = \partial\Omega$ and $\Gamma_D \cap \Gamma_N = \varnothing$. The 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 the stress tensor and it is a function of $p, \mathbf{u}$ and the dynamic viscosity μ. We define 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:

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

Since we are dealing with an evolutive problem, we need an initial condition and this is provided by the last equation of (2.1). 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 [25]. In Table 2.1 we report the 4 main formulations, together with the natural Neumann boundary conditions they give rise to.

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

2.1.1. Weak formulation

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

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

.

Under these assumptions, a weak formulation of (2.2) can be obtained by proceeding formally. Let us multiply the momentum conservation equation by a test function $\mathbf{v}\in\mathbf{V} = [H^1_{\Gamma_D}(\Omega)]^d$, 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 $q \in Q = \begin{cases} L^2(\Omega) & \text{if } \Gamma_N \neq \varnothing \\ L^2_0(\Omega) & \text{if } \Gamma_N = \varnothing \end{cases}$ 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 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 [29].

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 $v^n(x) \coloneqq v(x, t_n) \ \forall n \in \{0, 1, \ldots, N\}$. 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 Ω_h and let $\mathcal{T}_h = \{K_i\}_{i=1}^{N_{el}}$ be the associated mesh. We now select two finite-element spaces $\mathbf{V}_h \subset \mathbf{V}$, $Q_h \subset Q$ such 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, we 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 discrete problem 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:

$$\begin{cases} \int_{\Omega_h} \frac{1}{\Delta t}\alpha_{BDF}\mathbf{u}^{n+1}_h \cdot \mathbf{v}_h d\Omega_h + \mathcal{D}(\mathbf{u}^{n+1}_h, \mathbf{v}_h; \mu) + \int_{\Omega_h}[(\mathbf{u_*}^{n+1} \cdot \nabla)\mathbf{u}^{n+1}_h] \cdot \mathbf{v}_h d\Omega_h - \int_{\Omega_h} p^{n+1}_h \nabla \cdot \mathbf{v}_h d\Omega_h \\ = \int_{\Omega_h} \frac{1}{\Delta t}\mathbf{u}^n_{hBDF} \cdot \mathbf{v}_h d\Omega_h + \int_{\Omega_h} \mathbf{f}^n_h \cdot \mathbf{v}_h d\Omega_h + \int_{\Gamma_N} \mathbf{h}^{n+1}_h \cdot \mathbf{v}_h d\gamma \qquad \forall \mathbf{v}_h \in \mathbf{V}_h \\ \int_{\Omega_h} q_h \nabla \cdot \mathbf{u}^{n+1}_h d\Omega_h = 0 \qquad \forall q_h \in Q_h \end{cases} \tag{2.10}$$

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 linear.

The fluid dynamics module includes the possibility to apply a stabilization procedure to problem (2.10). One of the most common is SUPG-PSGS, a combination of two different stabilization methods: Streamline-Upwind-Petrov-Galerkin (SUPG) and Pressure-Stabilizing-Petrov-Galerkin (PSPG). The utility of such method 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-domination 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, regardless of the degree of the finite element spaces employed in the simulation.

For more details about the SUPG-PSPG stabilization see [31].
Other stabilization techniques are available in **life**$^{\mathbf{x}}$ in order to be able to solve problem (2.6) in 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 large, the pulsatile nature of blood flow does not allow a full transition to turbulence ([27]). 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. [20]).

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 $\{\boldsymbol{\phi}_j\}_{j=1}^{\mathcal{N}_\mathbf{u}}$ 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:

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

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 $\{\boldsymbol{\phi}_j\}_{j=1}^{\mathcal{N}_\mathbf{u}}$ 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:

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

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**$^\mathbf{x}$, 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) 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, this 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 Schur complement. More details about the SIMPLE preconditioner can be found in [26].

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 rate is usually high ([15]).
Blood, however, has more complex mechanical properties: those become especially significant where the particles size is comparable with the lumen size. Indeed, at low shear rates, the erythrocytes tend to aggregate and, consequently, the viscosity of the blood increases. On the contrary, if the shear rate increases, it will eventually be high enough to deform the erythrocytes, thus decreasing the viscosity. We refer to [32] for more details about this process. For this reason, blood has, in general, larger viscosity than plasma and, when the volume percentage of red blood cells (hematocrit) rises, the viscosity of the suspension increases and the non-Newtonian behavior 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 $\text{Pa} \cdot \text{s}$ for shear rates larger than 200 s^{-1} ([10]).

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 relation 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

"generalized Newtonian fluids", for which a relation between the viscosity and the scalar strain rate $\dot{\gamma}$ 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 generalized 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 $\operatorname{tr}\left(\varepsilon(\boldsymbol{u})\right) = \nabla \cdot \boldsymbol{u} = 0$, we have $II_\varepsilon = -\frac{1}{2}\left(\operatorname{tr}\left(\varepsilon(\boldsymbol{u})^2\right)\right)$, so if we define the shear rate as $\dot{\gamma} = \sqrt{2\operatorname{tr}\left(\varepsilon(\boldsymbol{u})^2\right)}$ 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 function $\mu(\dot{\gamma})$ must be specified. Several models can be used for this relation: one of the most used ones for blood simulations is the Carreau model, in which 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, for instance [10], where they obtained the values reported here by numerical fitting of experimental data. It should be noticed that μ_∞ corresponds 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 proven to well fit experimental data (see for example [10], [23], [30]). The introduction of this constitutive equation leads to an additional non-linear term in the Navier-Stokes equation, since now the viscosity depends on the solution $\boldsymbol{u}$.

2.2.2. Blood viscosity and 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 [12]. 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 even reach a physiological value of $6 \times 10^{-2} Pa \cdot s$ [16].
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 $< \ 10^{-1} s^{-1}$, 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 $10^{-1} s^{-1}$ and $5 \times 10^2 s^{-1}$. In this region the viscosity is very sensitive to the shear rate.
- a low viscosity plateau, for a shear rate $> \ 5 \times 10^2 s^{-1}$. In this region the viscosity can

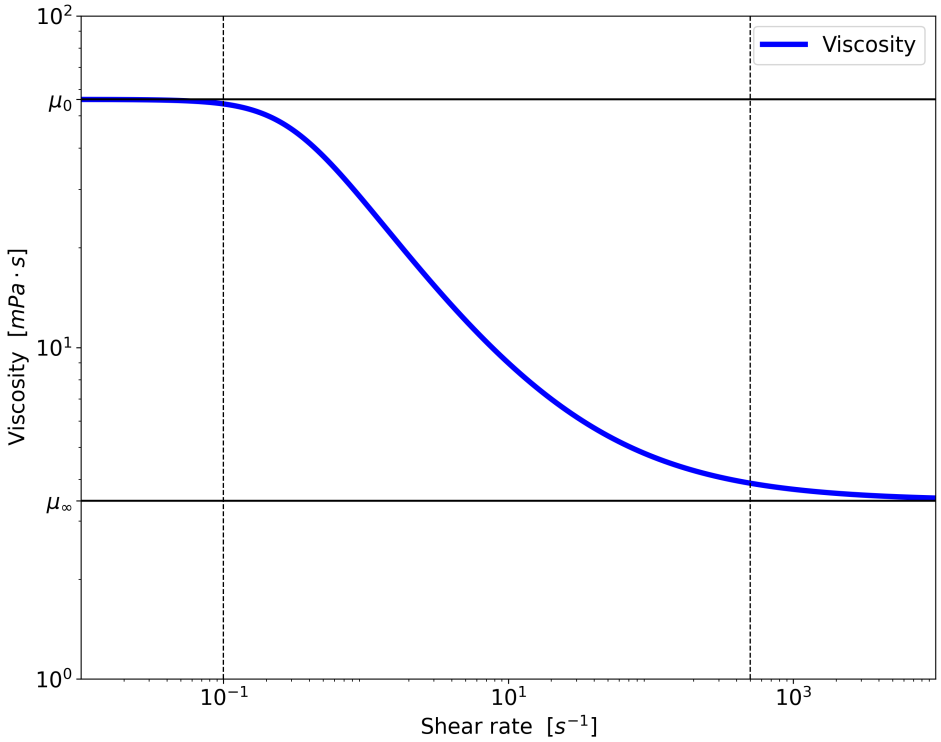


Figure 2.1: Relation between shear rate and dynamic viscosity according to Carreau model (2.17)

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 with the magnitude of the symmetric velocity gradient, we can draw some hypothesis on the expected distribution of the viscosity corresponding to a blood flow in a cylindrical domain.
For example, in the case of a 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 in the next section): we expect a high value of the viscosity in the regions near each minimum/maximum point since they correspond to null velocity gradient.

2.3. Inverse Womersley problem

In the previous sections, we stated 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 large vessels. Indeed, these two profiles are limit cases of the Womersley profile, that is 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 large or small (we will better specify this condition) the former approximations are no more suitable and the 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 flowrate through a circular surface, we aim at reconstructing 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 outside of the fields of biological fluid dynamics and biomedical engineering, since flowrate is the main physical quantity that can be 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 oriented as the longitudinal axis of the cylinder and let $\mathbf{u} = (u_r, u_\theta, u_z)$. In steady flow the solution is the well-known Poiseouille velocity profile: $u_r = u_\theta = \frac{\partial p}{\partial r} = \frac{\partial p}{\partial \theta} = 0$; $\frac{\partial p}{\partial z} = \frac{p_2 - p_1}{l} \leqslant 0$ where p_1 and p_2 are the pressures respectively at the beginning and at the end of the pipe.
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 [33], 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:

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

we can recognize a non-homogeneous modified Bessel differential equation (see [9] 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 these kinds 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 $\text{Wo} := r\sqrt{\frac{n}{\nu}}$ 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 for $\text{Wo} \longrightarrow 0$ and $\text{Wo} \longrightarrow +\infty$ respectively. 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 for a parabolic velocity profile to develop during each cycle and the flow will have low phase difference with respect to the pressure gradient. In this case a good approximation will be given by Poiseuille's law, using the instantaneous pressure gradient.

- When Wo is large ($\geqslant 10$) it means the frequency of pulsations is sufficiently large for the velocity profile to be relatively flat and the mean flow is about 90 degrees ahead of 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 project: the **inverse Womersley problem**.
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 flowrate for a period of the pulsation of the flow itself (recall that the flowrate and the pressure gradient oscillate with the same period). We aim at setting a Dirichlet boundary condition on the inlet, imposing the Womersley velocity profile $v(r,t)$ corresponding to the given flowrate $q(t)$: this is the **inverse Womersley problem**, described and solved in the following section.

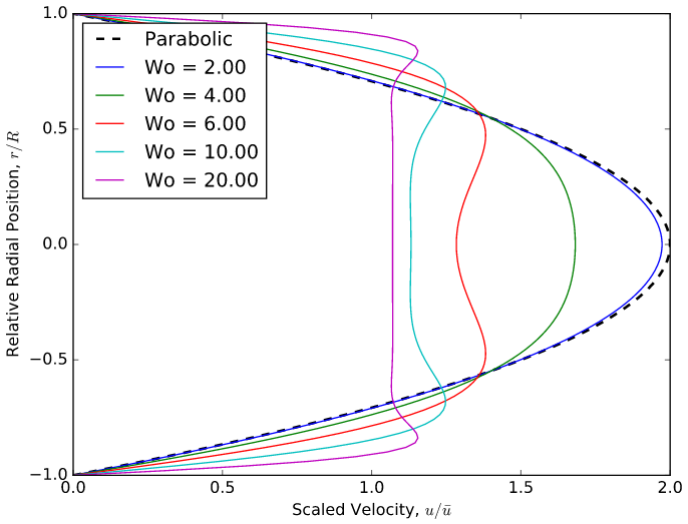


Figure 2.2: Velocity profiles for different values of Wo

2.3.2. Assumptions and definition of parameters

Let us suppose that the the flowrate 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 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 $\omega_n := \frac{2\pi n}{T}$ $\forall n \in \{-N, -N+1, \ldots, N-1, N\}$.
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. $\sigma_n \mapsto v_n$

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), we define 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 [13] 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 decomposed into two coupled Poisson problems.
The problem is then solved in [8] for a circular section, obtaining 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.

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 flowrate $q(t)$ 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**$^{\mathbf{x}}$ library [4]. **life**$^{\mathbf{x}}$ is a high performance library for the solution of multi-physics and multi-scale problem, mainly for cardiac applications [7]. It is written in C++ using the most modern programming techniques available in the C++17 standard and it is based on the deal.II finite element core [3]. All the code is natively parallel, highly scalable and designed to run on different architectures.
The **life**$^{\mathbf{x}}$ library consists in different classes, each one designed to solve a specific physics, and then coupled and further specialized for the solution of practical problems. In particular, there are several classes implemented for the numerical solution of some macro physical problems such as electrophysiology, electromechanics, fluid dynamics and many others.
In this project we focused mainly on the *FluidDynamics* class and in the next sections we are going to briefly describe it: this will help in the understanding of the implementation of the specific subjects of this project.

Figure 1: **life**$^{\mathbf{x}}$ 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**$^{\mathbf{x}}$, 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 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 regarding 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 equations 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 through two methods inherited from *CoreModel*: `declare_parameters` and `parse_parameters`. Thanks to these methods the user can collect all the parameters required in a simulation in a

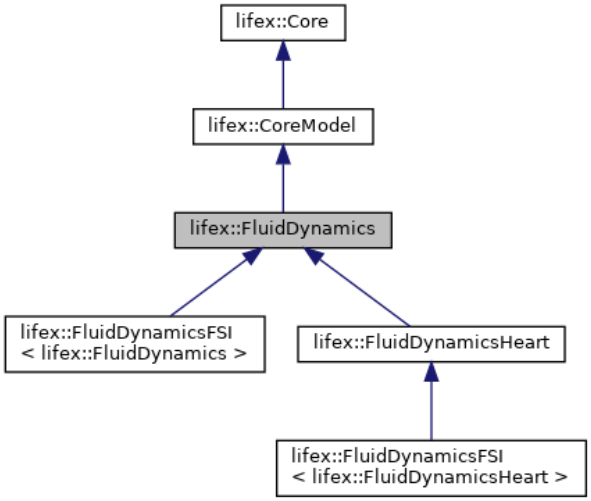


Figure 3.1: Inheritance diagram for the *FluidDynamics* class

prm file (generated with an arbitrarily chosen verbosity by `declare_parameters`, when the code is compiled with the `-g` flag): those are then directly read in the code by `parse_parameters`.
As we said, the *FluidDynamics* class only serves as a generic Navier-Stokes solver. So, in order to use this class for the solution of any specific problem, an additional step is necessary to define the domain, how to apply boundary conditions and so on. This can be done through further inheriting (this is done for example in *FluidDynamicsHeart* or *FluidDynamicsFSI*, see bottom of Figure 3.1), or by using other classes containing *FluidDynamics* as a member. This is the case, for example, of *TestFluidDynamicsCylinder*: a class that implements 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.cpp`

In the main, after a member of a class called *lifex_init* is constructed in order to initialize **life**$^{\text{x}}$ *Core* functionalities, the constructor of *TestFluidDynamicsCylinder* is called (actually through an object of a template class of type *CSVTest* that adds some testing functionalities to it, as it can be

seen in the code above). Since *TestFluidDynamicsCylinder* has a member of type *FluidDynamics*, the constructor of *FluidDynamics* is called. After that, thanks to `main_run_generate` (a member function of *CoreModel*) 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 represent the different physical models) is called. The function `run` is in charge of running 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 Section 3.3 where the main focus is on the imposition of the boundary conditions) and then `FluidDynamics::run` is finally called: this is the function that computes the solution of the Navier Stokes equations by means of Finite Elements method. The most relevant lines of this function are shown below:

```
1  void
2  FluidDynamics::run()
3  {
4    setup_system();
5      ...
6    // Time loop.
7    while (time < prm_time_final)
8      {
9        time_advance();
10         ...
11       apply_BCs();
12         try
13           {
14             solve_time_step(true);
15           }
16         ...
17       output_results();
18       print_info();
19     }
20 }
```

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 are going to be used for the assembly of the algebraic system (with size depending on the number of quadrature nodes).

The assembly and resolution 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 over the elements of the grid and integrating according to a numerical quadrature rule. The linearized system is then solved using a preconditioner (chosen by the user) and the time-dependent quantities are 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 space and time scheme discretization can be obtained in an analogous way as in Subsection 2.1.2, with a semi-implicit treatment of the convective term. The non-linear term arising from the Carreau model is treated semi-implicitly as well (as in [15]). The resulting formulation is analogous to 2.10 where the viscosity depends on the solutions to the previous time steps in the following way:

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

where $\mu\left(\mathbf{u}_{hBDF}^{n}\right)$ is calculated according to 2.17.
In order to implement this new feature in the *FluidDynamics* class we proceeded in the following way:

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

```
params.enter_subsection("Non-Newtonian model");
{
  params.declare_entry_selection("Model",
                                 "None",
                                 "None | Carreau",
                                 "Non-Newtonian model to be used.");

  params.declare_entry("Viscosity zero",
                       "5.6e-2",
                       Patterns::Double(0),
                       "Viscosity zero [Pa * s].");

  params.declare_entry("Viscosity infinity",
                       "3.45e-3",
                       Patterns::Double(0),
                       "Viscosity infinity [Pa * s].");

  params.declare_entry("Exponent power law",
                       "0.3568e0",
                       Patterns::Double(0),
                       "Exponent power law [1].");

  params.declare_entry("Lambda",
                       "3.313e0",
                       Patterns::Double(0),
                       "Lambda [s].");

  params.declare_entry(
    "Enable viscosity output",
    "true",
    Patterns::Bool(),
    "If a non-newtonian model is enabled, toggle output of viscosity.");
}
params.leave_subsection();
```

Listing 3.3: `declare_parameters`

```
    params.enter_subsection("Non-Newtonian model");
    {
      const std::string &non_newtonian_model_string = params.get("Model");

      // NonNewtonianModel.
      if (non_newtonian_model_string == "None")
        {
          prm_flag_non_newtonian_model = NonNewtonianModel::None;
        }
      else if (non_newtonian_model_string == "Carreau")
        {
          prm_flag_non_newtonian_model = NonNewtonianModel::Carreau;
        }

      if (prm_flag_non_newtonian_model == NonNewtonianModel::Carreau)
        {
          prm_carreau_viscosity_zero = params.get_double("Viscosity zero");
          prm_carreau_viscosity_infinity =
            params.get_double("Viscosity infinity");
          prm_carreau_exponent_power_law =
            params.get_double("Exponent power law");
          prm_carreau_lambda = params.get_double("Lambda");
        }

      prm_flag_output_viscosity = params.get_bool("Enable viscosity output");
    }
    params.leave_subsection();
```

Listing 3.4: `parse_parameters`

Thanks to these additions the user is allowed to tune 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 at each quadrature node. We report here a representative code snippet for the assembly of $\varepsilon(\mathbf{u})$.

```
    if (prm_flag_non_newtonian_model == NonNewtonianModel::Carreau)
      {
        for (unsigned int c1 = 0; c1 < dim; ++c1)
          for (unsigned int c2 = 0; c2 < dim; ++c2)
            symgrad_u_ext_loc[q][c1][c2] +=
              sol_ext_i * fe_values[velocities]
                            .symmetric_gradient(i, q)[c1][c2];
      }
```

Listing 3.5: Assembly of `symgrad_u_ext_loc`

The same procedure is followed for all the quantities needed to assembly the algebraic system, and this is done in the loop on the quadrature nodes inside the function `solve_time_step` mentioned in Section 3.1.

- A function that computes the viscosity based on the non-Newtonian model (2.17) was implemented.

```
double
compute_viscosity_carreau(const unsigned int q) const
{{{
  // Incompressibility implies trace(symgrad_u_ext_loc)=0, thus the
  // second invariant is just trace([symgrad_u_ext_loc]^2)
  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 at every quadrature point was 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

  The variable `viscosity_loc_all_cells` is used only for post processing purposes, as we will explain at the end of this section.

- Wherever the viscosity has to be used for the calculation of some quantities (for example in the assembly of the diffusive term 3.1 and, possibly, the preconditioners), if a non-Newtonian model is selected, 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

In the output of the simulation we added the possibility to store the viscosity associated to each cell (and at every time instant) in xdmf and h5 files (the same file where the values of pressure

and velocity are stored), that can be read by a suitable application (e.g. **Paraview**).

```
    // Project non-Newtonian viscosity at DoFs for output purposes.
    if (prm_flag_output_viscosity == true &&
        prm_flag_non_newtonian_model != NonNewtonianModel::None)
      {
        project_l2_scalar->project<std::vector<std::vector<double>>>(
          viscosity_loc_all_cells, viscosity_fem_owned);

        viscosity_fem = viscosity_fem_owned;
      }
```

Listing 3.9: Projection of `viscosity_loc`

In every cell, this output quantity is computed as the L^2 projection onto 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 was to implement a solver for the **inverse Womersley Problem** in order to impose a 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 $[0, T]$ (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 associated pressure gradient oscillates like the given flowrate, possibly with a different phase.

In this case $v_z(r, t)$, is the Womersley velocity profile, that is the solution of 2.20.
The procedure described in Subsection 2.3.1 relies on the assumption of circular section. Thus, even if we worked in *TestFluidDynamicsCylinder* (the application in a cylindrical domain of the *FluidDynamics* class that we already mentioned in Section 3.1), the implementation works for every type of domain that has a circular inlet section.
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 the other Dirichlet boundary conditions that were already implemented for *TestFluidDynamicsCylinder*. Indeed these are handled by *FlowBC*, a class that represents the boundary condition $\mathbf{u}(t, \mathbf{s}) = g(t)\mathbf{f}(\mathbf{s})$ on Γ_D, where $\mathbf{s} = (x, y)$ is a point of the inlet surface (or, possibly, other parts of the boundary), $\mathbf{f}$ is the space distribution of the velocity profile (a *SpaceFunction*) and g its time evolution (a *TimeFunction*). In the case of Womersley velocity profile, as it can be observed from 2.24, this assumption is not satisfied. However, by means of Fourier expansion, we can write the input flowrate as a sum of functions with separable variables. For this reason we needed to devise a new class for the Womersley profile.

In order to solve this issue we decided to implement a new class able to handle this specific problem, keeping the interface for 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.

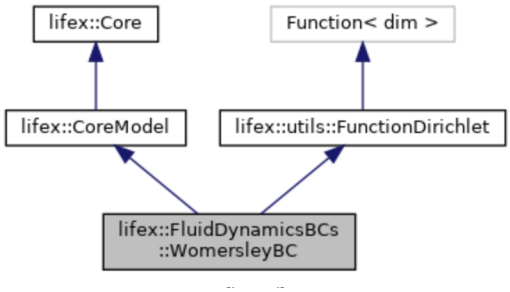


Figure 3.2: Inheritance diagram for the *WomersleyBC* class

The new class that we implemented is called *WomersleyBC* and, as we can see from Figure 3.2, inherits from *CoreModel* and *FunctionDirichlet*. The inheritance from *CoreModel* is necessary in order to receive the parameters, 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* [3, 11], but with a different type to prevent undesired 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   // Single Womersley mode defined as a subclass.
9   class WomersleyMode : public Function<dim, std::complex<double>>
10  {
11  public:
12  /// Constructor
13    WomersleyMode(const double &    an_,
14                  const double &    bn_,
15                  const size_t &    index_,
16                  const FluidDynamics &fluid_dynamics_)
17
18    // The value is a complex scalar: return it via value method.
19    std::complex<double>
20    value(const Point<dim> & p,
21          const unsigned int component = 0) const override;
22
23    //... some utilities functions .../
24
25  protected:
26    /// Real part of Fourier coefficient
27    const double an;
28
29    /// Imaginary part of Fourier coefficient
30    const double bn;
```

```
31
32      /// Index of Fourier coefficient
33      const size_t index;
34
35      /// Inlet surface area
36      double surface_area;
37
38      /// Inlet surface barycenter
39      Point<dim> barycenter;
40
41      /// Reference to the fluid dynamics instance.
42      const FluidDynamics &fluid_dynamics;
43    };
44
45
46
47
48    /// Override of vector_value method.
49    virtual void
50    vector_value(const Point<dim> &p, Vector<double> &values) const override;
51
52    //... some utilities functions ...//
53
54    protected:
55    // Modes.
56    mutable std::vector<WomersleyMode> modes;
57
58    /// Function for csv evolution.
59    std::shared_ptr<CSVFunction> flowrate_over_time;
60
61    /// Reference to the fluid dynamics instance.
62    const FluidDynamics &fluid_dynamics;
63
64    /// Period of the flow rate input function
65    double period;
66
67    /// bool variable used to initialize modes at the first entering in
68    /// vector_value
69    mutable bool initialized;
70
71    //... other members ...//
72  };
```

Listing 3.10: Interface of *WomersleyBC*

Listing 3.10 shows the interface of the class *WomersleyBC*. Among its protected members we find `flowrate_over_time` and `modes`: the first is a shared pointer to *CSVfunction* that represents the flowrate $q(t)$ read from the given csv file; the second is a vector of *WomersleyMode*: a inner class of *WomersleyBC* representing a single mode in the Fourier expansion. In particular, given a flowrate in the form $q(t) = \sum_{n=-N}^{N} q_n e^{jw_n t}$ where j is the imaginary unit and w_n is the angular frequency, the k-th element of `modes` stores the real and the imaginary part of the coefficient q_k in the variables `an` and `bn` respectively, together with the index k. We stress that there is no need to create a *WomersleyMode* object for the modes corresponding to a negative index since q_{-k} can be computed as the complex conjugate of q_k. The most relevant method of *WomersleyMode* is `value`, which, as we will describe in details later, takes as input the coordinates of an element of the inlet surface and returns the Fourier mode $v_k(r)$ of the Womersley velocity profile, where r

is the distance of the input element from the inlet surface barycenter and k is the value stored in `index`.

The method responsible for the imposition of the boundary condition is `WomersleyBC::vector_value`: a method inherited (and overridden) from the dealII function.

```
void
  WomersleyBC::vector_value(const Point<dim> &p, Vector<double> &values) const
  {
    const Point<dim> &barycenter = this->get_barycenter();
    using namespace std::complex_literals;
    const double r               = std::sqrt(p.distance_square(barycenter));
    const double boundary_radius = std::sqrt(this->get_surface_area() / M_PI);
    const std::vector<double> &re_ck =
      flowrate_over_time->get_time_interpolation().get_re_ck();
    const std::vector<double> &im_ck =
      flowrate_over_time->get_time_interpolation().get_im_ck();
    const unsigned int         mu = (re_ck.size() + 1) % 2;
    const size_t               M  = (re_ck.size() - mu - 1) / 2;
    std::complex<double>       v_n(0.0, 0.0);
    std::complex<double>       v_n_conj(0.0, 0.0);
    std::complex<double>       velocity(0.0, 0.0);
    const double               t_mod = this->get_time() / period * 2 *
                         M_PI; // t \in [0,T] mapped into t_mod \in [0,2*pi]

    const double sigma0 =
      (8.0 * fluid_dynamics.get_viscosity() * re_ck[M]) /
      (fluid_dynamics.get_density() * M_PI * std::pow(boundary_radius, 4));
    const double v0 =
      (sigma0 * boundary_radius * boundary_radius) /
      (4.0 * fluid_dynamics.get_viscosity() / fluid_dynamics.get_density()) *
      (1 - (r / boundary_radius) * (r / boundary_radius));

    velocity += v0;


    if (!initialized)
    {
    for (size_t n = 1; n <= M; ++n)
    {
      modes.push_back(WomersleyMode(re_ck[M + n], im_ck[M + n], n, fluid_dynamics));
      modes[n-1].set_barycenter(get_barycenter());
      modes[n-1].set_surface_area(get_surface_area());
    }
    if (mu == 1)
    {
      modes.push_back(WomersleyMode(re_ck[2 * M + 1], im_ck[2 * M + 1], 2 * M + 1, fluid_dynamics));
      modes[M].set_barycenter(get_barycenter());
      modes[M].set_surface_area(get_surface_area());
    }
    initialized = true;
    }

    for (size_t n = 1; n <= M; ++n)
      {
        v_n      = modes[n - 1].value(p); // compute v_n
```

```
51          v_n_conj = std::conj(v_n);          // compute v_{-n}
52          velocity += v_n * std::exp(1i * static_cast<double>(n) * t_mod) +
53                      v_n_conj * std::exp(-1i * static_cast<double>(n) * t_mod)↩
                        ;
54        }
55
56      if (mu == 1)
57        {
58          v_n = modes[M].value(p); // compute v_n
59          v_n *= 2.0 * std::cos((M + 1) * t_mod);
60          velocity += v_n;
61        }
62
63      for (unsigned int j = 0; j < dim; ++j)
64        {
65          if (utils::is_positive(boundary_radius - r))
66            values[j] = -1.0 * velocity.real() * scaling_factor *
67                        this->get_normal_vector()[j];
68
69          else
70            values[j] = 0.0;
71        }
72
73      // Pressure component.
74      values[dim] = 0.0;
75    }
```

Listing 3.11: `WomersleyBC::vector_value`

This function aims to set the desired 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). In this function the **Inverse Womersley problem** is solved through the following steps:

1. Given the flowrate $q(t)$ 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 flowrate $q(t)$ 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 20-28 solve step 2. computing σ_0 and $v_0(r)$ through the Poiseuille flow theory.

During the first call to this function, at lines 31-46 an object of type *WomersleyMode* is created for each flowrate Fourier mode (with positive index) and stored in the vector `modes`.
At lines 48-54 a `for` cycle loops over the possible values of the index $n \in \{1, 2, \dots, N\}$ and the Fourier mode v_n is computed through the function `WomersleyMode::value`.
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 56-61 handles the case in which the number of data provided in the csv file for the flowrate $q(t)$ 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 flowrate 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 the function is implemented in a general way so that it works for any orientation of the inlet surface.
Finally, the pressure component is set equal to 0.

```
1  std::complex<double>
2    WomersleyBC::WomersleyMode::value(const Point<dim> & p,
3                                      const unsigned int component) const
4    {
5      const Point<dim> &barycenter = get_barycenter();
6      using namespace std::complex_literals;
7      const double r = std::sqrt(p.distance_square(barycenter));
8      const double boundary_radius =
9        std::sqrt(get_surface_area() / M_PI);
10     const std::complex<double> q_n = std::complex<double>(an, bn);
11     const double                omega_n =
12       2 * M_PI * index / fluid_dynamics.get_period();
13     const double wo_r =
14       std::sqrt(r * r * omega_n * fluid_dynamics.get_density() /
15                 fluid_dynamics.get_viscosity());
16     const double wo_R = std::sqrt(boundary_radius * boundary_radius * omega_n↩
           *
17                                   fluid_dynamics.get_density() /
18                                   fluid_dynamics.get_viscosity());
19     const std::complex<double> arg_bessel =
20       std::complex<double>(0.0, (wo_R * wo_R) / 4.0);
21
22     // The following lines of code compute the value of the Fourier ↩
          coefficients
23     // associated to the velocity profile.
24
25
```

```
26      const std::complex<double> f_11 =
27        sp_bessel::besselJ(0, 2.0 * std::sqrt(-arg_bessel));
28      const std::complex<double> f_12 =
29        (std::pow(-arg_bessel, -1 / 2.0)) *
30        (sp_bessel::besselJ(1, 2.0 * std::sqrt(-arg_bessel)));
31
32      const std::complex<double> sigma_n =
33        (1i * omega_n * q_n) /
34        ((M_PI * boundary_radius * boundary_radius) * (1 - f_12 / f_11));
35
36      const std::complex<double> j_0R = sp_bessel::besselJ(
37        0, 0.5 * (std::complex<double>(-std::sqrt(2), std::sqrt(2))) * wo_R);
38      const std::complex<double> j_0r = sp_bessel::besselJ(
39        0, 0.5 * (std::complex<double>(-std::sqrt(2), std::sqrt(2))) * wo_r);
40
41      return sigma_n / (1i * omega_n) * (1 - j_0r / j_0R);
42    }
```

Listing 3.12: `WomersleyMode::value`

To conclude, in Listing 3.12 we reported the implementation of the function `WomersleyMode::value`. As we stated previously, it receives as input an object of type *Point* with the coordinates of an element of the inlet surface (the second input, which is set by default equal to 0, is not relevant since it is not used inside the function).
We can split this function in three main parts:

- At lines 5-20 the required variables are initialized. The Fourier mode q_n is initialized at line 10 as a complex double with real part equal to `an` and imaginary part equal to `bn`.
- Lines 26-34 are devoted to the computation of σ_n. This is done according to 2.27. In particular, for evaluating the regularized confluent hyper-geometric limit function ${}_0\tilde{F}_1$ we exploited the following identity:

$$\frac{{}_0F_1(a,z)}{\Gamma(a)} = (-z)^{(1-a)/2} J_{a-1}(2\sqrt{-z})$$

- Finally, at lines 36-41, the Fourier mode $v_n(r)$ is computed exploiting 2.30 and returned by the function.

We want to point out the fact that this function involves the computation of a Bessel function with complex argument. However, in the Boost library [1], there is only a method to evaluate Bessel functions on real arguments. Thus, we had to use an external library, called `complex_bessel` [2], which allows complex arguments. In order to perform this operation we had to compile and link this as a dynamic library and suitably modify the Cmake files in order to use this function.

```
1
2   /// Constructor.
3       TestFluidDynamicsCylinder(const std::string &subsection,
4                                 const std::string &subsection_fluid)
5         : CoreModel(subsection)
6         , fluid_dynamics(subsection_fluid)
7         , inlet_dirichlet_bc(std::make_shared<FluidDynamicsBCs::FlowBC>(
8             subsection + " / Boundary conditions / Dirichlet inlet",
```

```
          std::initializer_list<std::string>(
            {FluidDynamicsBCs::Constant::label,
             FluidDynamicsBCs::Ramp::label,
             FluidDynamicsBCs::Pulsatile::label,
             FluidDynamicsBCs::CSVFunction::label}),
          FluidDynamicsBCs::Pulsatile::label,
          std::initializer_list<std::string>(
            {FluidDynamicsBCs::Parabolic::label,
             FluidDynamicsBCs::UniformVelocity::label}),
          FluidDynamicsBCs::Parabolic::label,
          fluid_dynamics))
      , inlet_womersley_dirichlet_bc(
          std::make_shared<
            FluidDynamicsBCs::WomersleyBC>( // Instantiation of new
                                            // object to take care of ↩
                                               Womersley
                                            // profile if requested
            subsection + " / Boundary conditions / Dirichlet inlet",
            fluid_dynamics))
      , inlet_neumann_bc(std::make_shared<FluidDynamicsBCs::PressureBC>(
        // ... //
        )
      , outlet_neumann_bc(std::make_shared<FluidDynamicsBCs::PressureBC>(
        // ... //
        )
    {}
```

Listing 3.13: *TestFluidDynamicsCylinder* constructor

Now we see how these two new classes can be used in practice inside the *TestFluidDynamicsCylinder* class. In this class we can find a member, called `inlet_dirichlet_bc`, which is a shared pointer to a *FlowBC* object. This object, as we stated previously, is responsible for the imposition of a Dirichlet boundary condition in the case where the boundary condition can be represented by a function that can be separated in space and time. As we already said, since the Womersley velocity profile does not match this condition, we had to implement an alternative class, *WomersleyBC*, with the same inheritances of *FlowBC*. Consequently, we introduced a new object, called `inlet_womersley_dirichlet_bc`, which is a shared pointer to *WomersleyBC* and it is initialized in the constructor of *TestFluidDynamicsCylinder* (see Listing 3.13).

```

      // Build BCs.
      std::vector<utils::BC<utils::FunctionDirichlet>> bcs_dir(
        1,
        utils::BC<utils::FunctionDirichlet>(
          prm_tag_wall,
          std::make_shared<utils::ZeroBCFunction>(dim + 1),
          ComponentMask({true, true, true, false})));
      std::vector<utils::BC<utils::FunctionNeumann>>   bcs_neu            = ↩
         {};
      std::vector<utils::BC<utils::FunctionDirichlet>> bcs_dir_tangential = ↩
         {};


      if (prm_inlet_type == "Dirichlet")
        {
          if (prm_womersley)

```

```
17            bcs_dir.emplace_back(prm_tag_inlet,
18                                 inlet_womersley_dirichlet_bc,
19                                 ComponentMask({true, true, true, false}));
20
21          else
22            bcs_dir.emplace_back(prm_tag_inlet,
23                                 inlet_dirichlet_bc,
24                                 ComponentMask({true, true, true, false}));
25        }
26      else if (prm_inlet_type == "Neumann")
27        bcs_neu.emplace_back(prm_tag_inlet,
28                             inlet_neumann_bc,
29                             ComponentMask({true, true, true, false}));
```

Listing 3.14: Boundary condition initialization.

Both classes *FlowBC* and *WomersleyBC* inherit from *CoreModel* and *FunctionDirichlet*. The *CoreModel* part is mainly used to generate and read the parameter file. Then, once we know the details about the boundary condition that has to be imposed, the code needs only the *FunctionDirichlet* part of the class, which can be either *FlowBC* or *WomersleyBC*.
In Listing 3.14 we reported the initialization of the object `bcs_dir`: a vector containing objects of type *utils::BC<utils::FunctionDirichlet>*. The template *BC* is a helper class used to represent an arbitrary boundary condition. Passing as template parameter *FunctionDirichlet* we are saying that the boundary condition is of Dirichlet type. The second argument in the constructor of *BC* is a shared pointer to the same type of the template parameter, which in our case is *FunctionDirichlet*. Thus, at lines 13-25 of Listing 3.14, a new *BC<FunctionDirichlet>* object is created and inserted in the vector `bcs_dir`. The new object takes as second argument of the constructor `inlet_womersley_dirichlet_bc` if the user chose to impose a Womersley boundary condition or `inlet_dirichlet_bc` in case of any other Dirichlet boundary condition.
In other words, we are passing a shared pointer to the *FunctionDirichlet* part of either *WomersleyBC* or *FlowBC*. So when the boundary condition had to be imposed in the solution of the Navier Stokes equation (in `Fluiddynamics::run()`), thanks to C++ polymorphism, the same functions are called in the two cases but they behave differently, depending on which type of Dirichlet boundary condition has been selected by the user.
In particular in `apply_BCs()` (called in `Fluiddynamics::run()`, see Listing 3.2) the function `FunctionDirichlet::vector_value` is called and, depending on which type of object we have in the vector `bcs_dir`, either `FlowBC::vector_value` or `WomersleyBC::vector_value` is called, imposing the desired velocity profile for the Dirichlet inlet.

3.4. How to run a simulation

In this section we show 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 modified passing the optional flag `VERBOSITY`, which can be either `minimal` or `full`. Also the `-f` flag is optional and it allows the user to customize the parameter file name. If `-f` is not used, a default name is assigned.
Now we inspect the most important features that the user can set from the prm file:

```
3  subsection Test fluid dynamics cylinder
4  subsection Boundary conditions
5    # Tag of the inlet boundary.
6    set Inlet tag                                  = 1
7
8    # Tag of the outlet boundary.
9    set Outlet tag                                 = 2
10
11   # Tag of the wall boundary.
12   set Wall tag                                   = 0
13
14   # Length of the cylinder [m], to impose an analytical ALE displacement.
15   set Length of the cylinder                     = .0310
16
17   # Radius of the inlet boundary [m].
18   set Radius of the inlet boundary               = 0.0031
19
20   # Displacement factor [m].
21   set Radial displacement factor                 = 100
22
23   # Dirichlet conditions impose a parabolic or flat velocity profile;
24   # Neumann conditions a uniform stress.
25   # Available options are: Dirichlet | Neumann.
26   set Type of inlet condition                    = Dirichlet
27
28   # If set to true, a Womersley velocity profile is imposed for Dirichlet
29   # boundary conditions.
30   set Womersley boundary condition               = true
31
32   # If set to true, the velocity tangential to the inlet boundary is set to
33   # zero for Neumann boundary conditions, resulting in practice in a mixed
34   # Dirichlet-Neumann condition.
35   set Constrain tangential flux for Neumann inlet = false
```

Listing 3.15: parameter file

In the first part the user can set the tags for different kind of boundaries, the length and radius of the cylinder and 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 tangent to the inlet surface, while the Neumann boundary condition is preserved in the normal direction. This option can be used to set a boundary condition different from the natural one associated to a certain formulation (see Table 2.1).
The flag `Womersley boundary condition` is a new feature introduced with this project. If this flag is set equal to `true` and `Type of inlet condition` is set to `Dirichlet` then a Womersley

velocity profile is imposed on the inlet surface at every time step.

```
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 .
      set Space distribution = Parabolic
```

Listing 3.16: parameter file

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

- `Constant`: the velocity field prescribed on the inlet surface is constant in time. The user has to specify the value of the velocity (in case of uniform flow) or the flowrate.
- `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 flowrate).
- `CSV function`: the velocity (or flowrate) 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 and the interpolation method that must be used to reconstruct the velocity (or flowrate) function.
- `Ramp`: the velocity (or flowrate) starts from a certain value v_1 and, at a given time t_1, it starts moving monotonously as a sinusoidal function to a final value v_2, reached at a second given time $t_2 = t_1 + \delta_t > t_1$. The value of the velocity is kept constant and equal to v_2 for every time instant $t \geqslant t_2$. 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.

We point out that, in case of Womersley boundary condition, the space function and time function selected are ignored and a csv file containing the input flowrate must be provided.
In case of Neumann boundary conditions 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

    # Viscosity zero [Pa * s].
    set Viscosity zero          = 5.6e-2

    # Viscosity infinity [Pa * s].
    set Viscosity infinity      = 3.45e-3

    # Exponent power law [1].
    set Exponent power law      = 0.3568e0

    # Lambda [s].
    set Lambda                  = 3.313e0
```

Listing 3.17: parameter file

There is also a section that contains all the parameters related to the adopted model and the fluid under study. In particular, the user must specify which model should be used (at the moment there are only two options: `None`, corresponding to the Newtonian model and `Carreau`, corresponding to the Carreau model) and the type of formulation for the diffusion term (see (2.7)).
Many other features can be set from the prm file. In particular, for all the simulation performed during this project, we made the following choices:

- A cylindrical mesh with hexahedral finite elements is employed.
- For the velocity and the pressure Finite Elements spaces we chose $\mathbf{V_h} = [\mathbb{P}^1(\Omega_h)]^3$ and $Q_h = \mathbb{P}^1(\Omega_h)$. Since this pair of spaces are not inf-sup stable, for all the experiments we used the SUPG-PSPG method.
- The non-linear term is always treated as semi-implicit (see the end of Subsection 2.1.2).
- Regarding time discretization, a BDF scheme of order 1 is employed (semi-implicit Euler method).

When all the parameters are suitably 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 the path in which 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 **Paraview**) to see the evolution in space and time of velocity, pressure and viscosity.

- One prm file containing all the values of the parameters employed.
- One csv file which contains the values of the velocity flowrate through portions of the boundary domain at each time step.

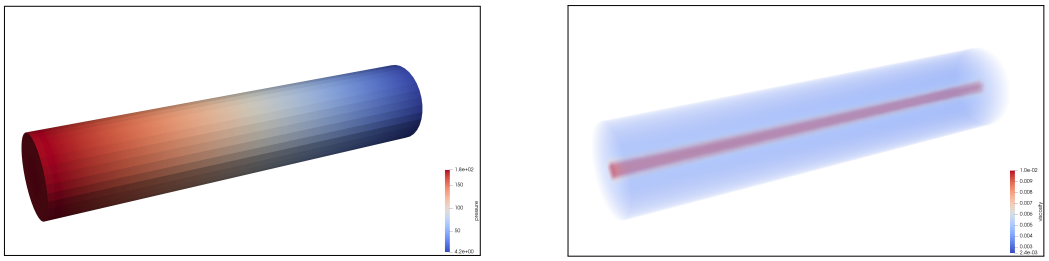
Additional information is displayed on terminal (or in a separate file if the user prefers to) during the execution, such as the Finite Elements spaces and stabilization employed and number of iterations needed by the iterative solver at each time-step.
Finally we implemented a bash script to execute multiple simulations sequentially without changing the parameter file every time, as reported in detail in Appendix A. 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 a simulation 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, thus simplifying the analysis of the results.

4. Numerical results

In this section we compare the results obtained solving problem (2.11) with a Newtonian rheology with the ones obtained by using the Carreau model for the non-Newtonian viscosity. In order to validate the model implementation, some reference results are taken by [30], where the same problem is solved in a cylindrical domain but using a different method.
A grid independence study highlighted that the results shown in this section are still sensitive to further refinements (a simulation with a higher refinement provided a flowrate on average 10-15% higher than the one obtained with the refinement employed to collect the other results), which on the other hand would affect dramatically the execution time, making a simulation campaign unfeasible. However, we are more interested in the differences between Newtonian and non-Newtonian rheologies and the results reported here are still reliable and in accordance with the ones reported in [30].

4.1. Steady-state blood flow

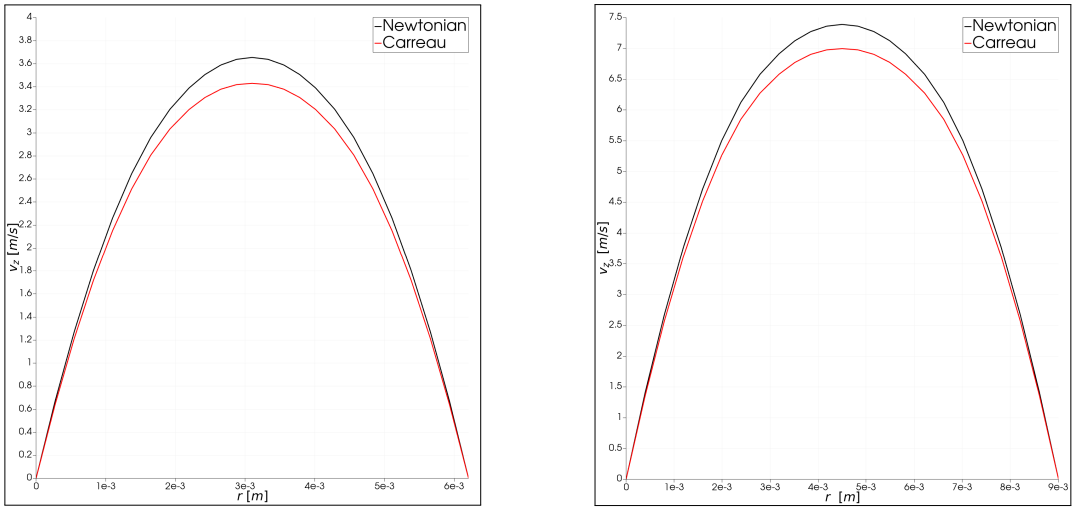
To assess the effectiveness of the Carreau model we first compared the results obtained imposing a steady-state flow. In particular, following the procedure illustrated in [30], we considered a cylindrical domain and two different radii equal to $R_1 = 0.0031m$ and $R_2 = 0.0045m$ respectively, with length equal to 10 times the radius. We set the density equal to $\rho = 1060kg/m^3$ and the viscosity for the Newtonian model equal to $\mu_{newtonian} = 0.00345Pa \cdot s$. The space discretization is performed by means of 1-degree hexagonal finite elements with SUPG-PSPG stabilization, while the time derivatives are approximated through a BDF1 scheme. We imposed a boundary condition on the inlet section and a Neumann homogeneous boundary condition on the outlet section, resulting in a constant pressure drop equal to $6000Pa/m$ (Figure 4.1a).



(a) Pressure (b) Viscosity

Figure 4.1: Pressure and viscosity distribution in the domain for the steady case.

In Figure 4.2 we can see a comparison between the velocity profiles associated to the Newtonian



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

Figure 4.2: Velocity profiles for the steady case.

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 yields to a greater value of the viscosity, is characterized by a lower flowrate.

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, where *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 difference between Newtonian and non-Newtonian flows, together with the values of the WSS, are in complete accordance with the reference results.

Finally, Figure 4.1b 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).

	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
Carreau WSS (Pa)	9.3	9.26

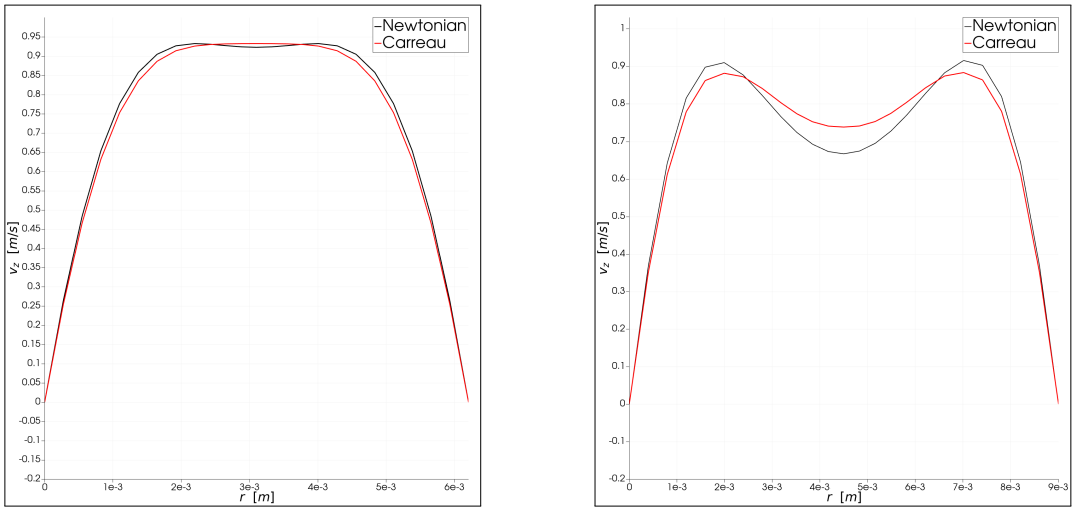
Table 4.1: Comparison with the reference results [30]. $r = 0.0031cm$

	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
Carreau WSS (Pa)	13.5	13.87

Table 4.2: Comparison with the reference results [30]. $r = 0.0045cm$

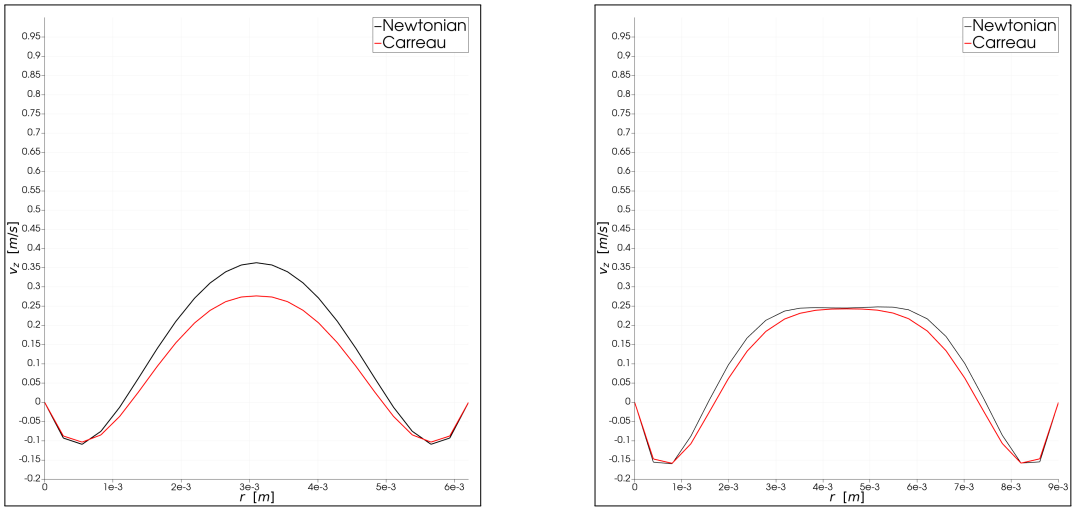
4.2. Pulsatile blood flow

The second test we have performed consists in solving a similar problem as in the previous section, but imposing a pulsatile pressure gradient. The problem solved by the code is formulated as it is reported in 2.10. However, for analysis purposes, the momentum equation can be written as 2.20 where the amplitude of the oscillatory pressure gradient is taken to be equal to $A = 6000Pa/m$ (corresponding to $45mmHg$) 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).



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

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



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

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

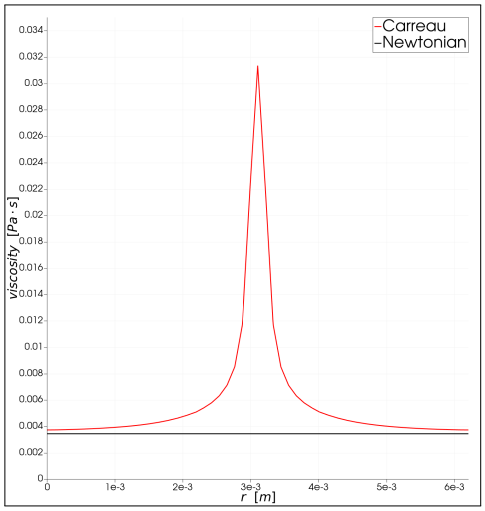
	Reference	life^x
Newtonian max flux (m^3/s)	$1.77e-5$	$1.86e-05$
Carreau max flux (m^3/s)	$1.73e-5$	$1.81e-05$

Table 4.3: Comparison of the max flow rate with the reference results [30]. $r = 0.0031cm$

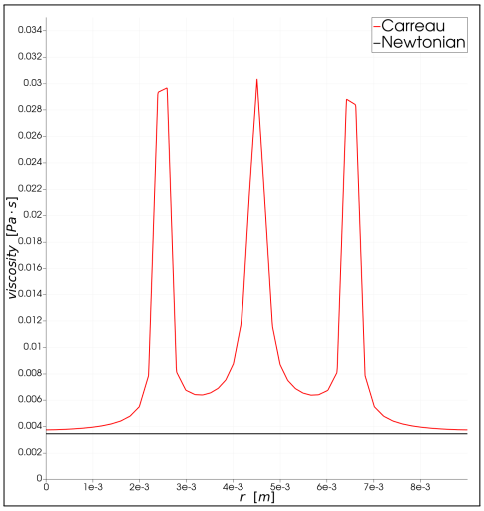
	Reference	life^x
Newtonian max flux (m^3/s)	$4.10e-5$	$4.23e-05$
Carreau max flux (m^3/s)	$4.06e-5$	$4.14e-05$

Table 4.4: Comparison of the max flow rate with the reference results [30]. $r = 0.0045cm$

Figure 4.3 shows the velocity profiles obtained as solutions of the Newtonian and non-Newtonian models, at the time of the maximum value of the flowrate. 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 mainly differ 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 flowrate (in the positive time cycle, i.e. the lowest positive value of the flowrate). 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 [17] for the 2D blood flow in the carotid artery using the Carreau-Yasuda model.
Tables 4.3-4.4 summarize the comparison of the results obtained using the **life^x** solver against the ones reported in [30].
The viscosity distribution in 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.

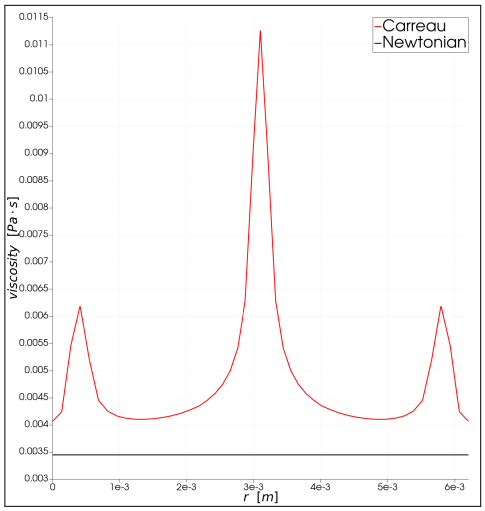


(a) $r = 0.0031m$

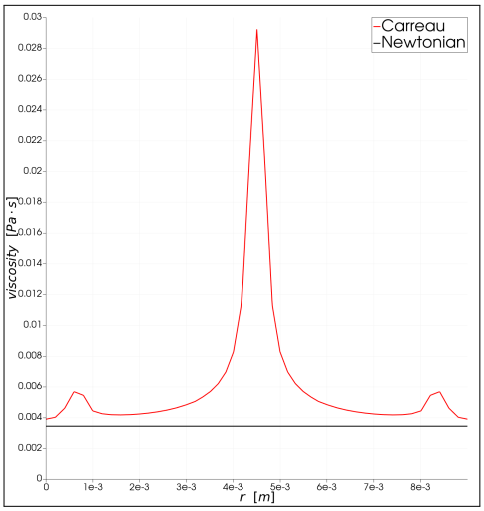


(b) $r = 0.0045m$

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



(a) $r = 0.0031m$



(b) $r = 0.0045m$

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

Its is worth noticing that, if an oscillating pressure gradient is imposed, the Womersley velocity profiles naturally develop in a cylindrical domain even if a Neumann or mixed (i.e. Neumann in the normal direction and homogeneous Dirichlet in the tangential direction) 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.

4.3. Womersley profile

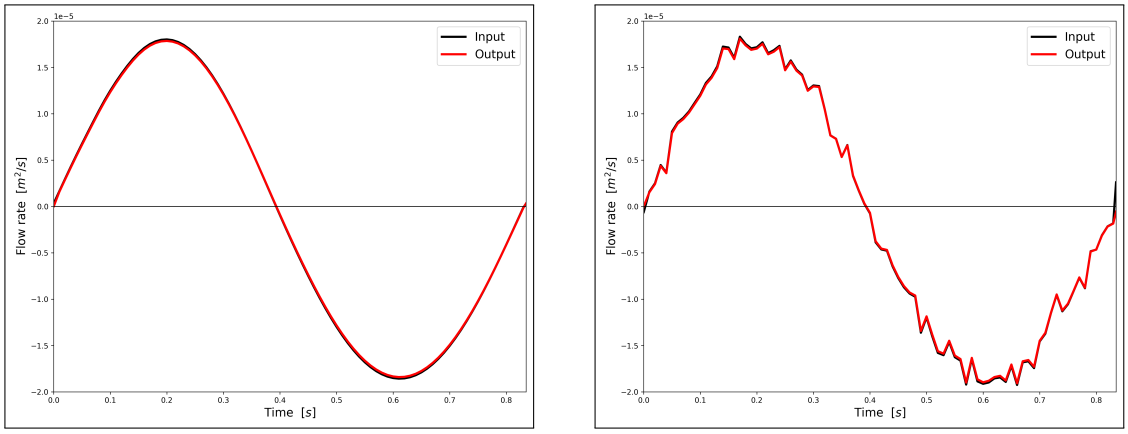
In this section we assess the effectiveness of our solver for the inverse Womersley problem, in order to impose a proper and realistic velocity profile in the inlet surface as Dirichlet boundary condition.
To this aim:

1. we perform a consistency check on the obtained flowrate
2. we compare the velocity profiles at the inlet (obtained here through an analytical procedure) with the ones obtained numerically in the previous section imposing 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: since we are interested in imposing the velocity profiles at the inlet, there are no differences between Newtonian and non-Newtonian models, and a Newtonian model requires a lower computational effort.

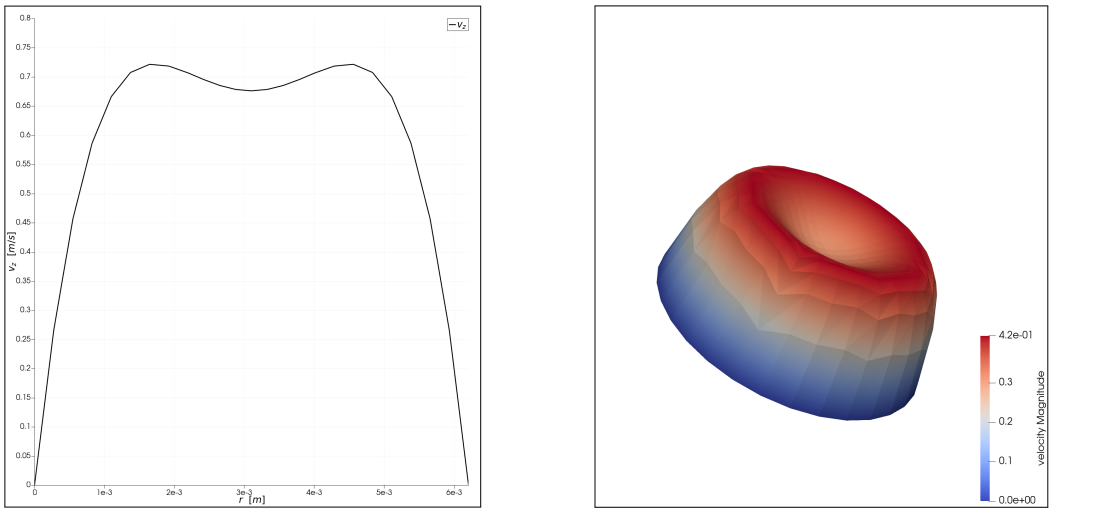
In order to generate a csv file containing a time-varying flowrate adequate for our purpose, we ran a simulation imposing an oscillating pressure jump berween inlet and outlet with amplitude $A = 6000Pa/m$ (as in the previous section). This simulation produced a csv file with the resulting flowrate over time $q(t)$, which we used as a Dirichlet boundary condition for the second simulation, in which we set the parameter `Womersley boundary condition` to `true` 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 their consistency.
In Figure 4.7a we report a comparison between the input flowrate (passed as a parameter through a csv file) and the output flowrate at the inlet, i.e. the flowrate generated by the solution (obtained integrating the velocity field according to a given quadrature rule) and we can see that the flowrate is successfully recovered.
Moreover, in Figure 4.8 we report the velocity profiles corresponding to the time instant $t = 0.14s$. The shape is conforming to the one predicted by the theory (see Subsection 2.3.1).
For our simulation we employed a sinusoidal flowrate with period $T = 1/f$ where $f = 1.2Hz$ (corresponding to 72 heart beats per minute). However, in real application, the blood flowrate is not sinusoidal and it can also present some irregularities due to measurement errors or pathological reasons. In light of this, we introduced a random noise in the input flowrate and we run a new simulation 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}.

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 the other one and this is coherent with the fact that $\tilde{q}(0.14) > q(0.14)$. 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 confirmation on the fact that the procedure that we have implemented in **life**$^{\mathbf{x}}$ 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 flowrate is often an easier and preferable procedure compared to the measurement of velocity profiles.



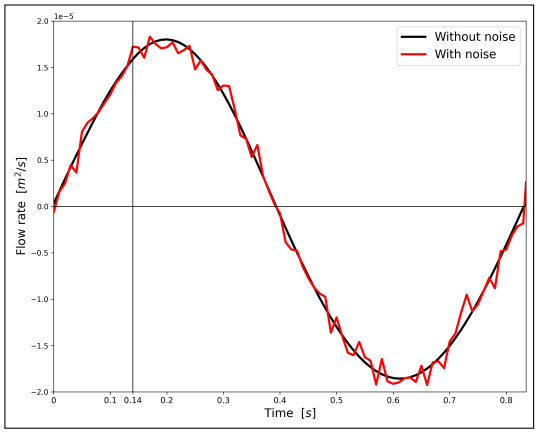
(a) Without noise (b) With noise

Figure 4.7: Comparison between input and reconstructed flowrates.

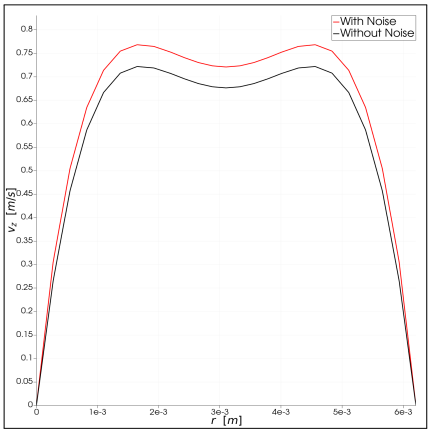


(a) 2D representation (b) 3D representation on a circular section

Figure 4.8: Reconstructed velocity profile at time $t = 0.14s$.



(a) Flowrates with and without noise



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

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

4.4. Simulation on a bifurcated carotid

As a final experimental result, we had the possibility to test the new features we have introduced on a realistic domain. Indeed, as we already stated, the implementation of the Womersley profile works with any type of circular inlet section so it can be applied, for example, in case of a circular vessel (for a non-Newtonian model there are no problems in terms of generalization). In particular we used a mesh representing a bifurcated carotid, employed also in [5, 28].

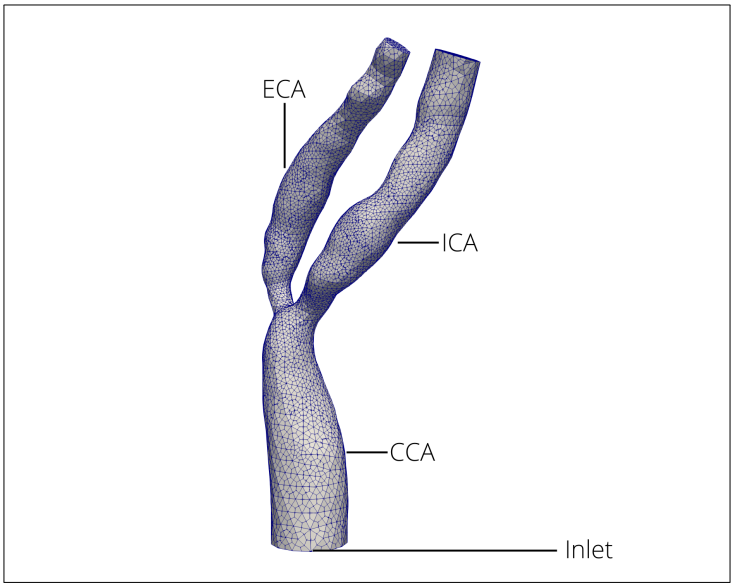


Figure 4.10: Geometry and hexahedral mesh for a bifurcated carotid.

Figure 4.10 shows the details of the mesh employed in the simulation. It is composed by about 54000 hexahedrons, and as can be noticed from the figure, size of the elements becomes smaller near the bifurcation in order to capture the complex behaviour of the flow in that area. More details about the number of elements and sizes of the elements of the mesh are reported in table 4.5. The space discretization is performed by means of Q1-Q1 finite elements with SUPG-PSPG stabilization, while the time derivatives are approximated through a BDF1 scheme.

Number of cells	54316
$h_{\max}$	$1.29 \cdot 10^{-3}$ m
$h_{\min}$	$2.22 \cdot 10^{-4}$ m
h_{mean}	$1.29 \cdot 10^{-3}$ m

Table 4.5: Number of elements and width of the mesh.

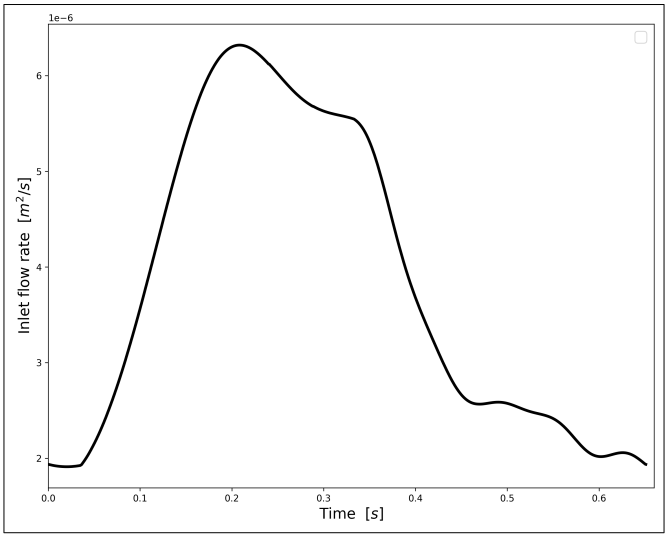
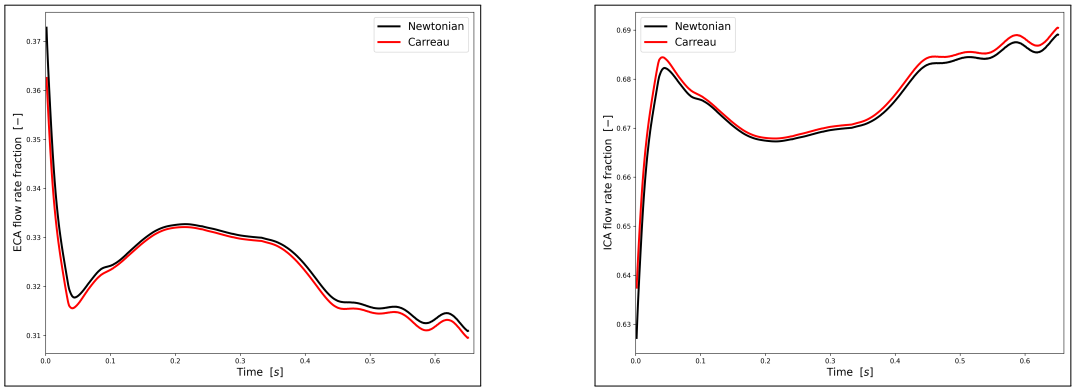


Figure 4.11: Flowrate imposed at the inlet surface as Dirichlet boundary condition.

In real applications, the blood enters from the aorta in the carotid through the common carotid artery (CCA) and it splits between the two bifurcations, called internal carotid artery (ICA) and external carotid artery (ECA). Thus, in the code, we consider as inlet surface the beginning of CCA. There, the Womersley velocity profile is imposed at each time step so that the resulting flow rate matches the one showed in Figure 4.11. In particular, the period of the flow rate is $0.6475s$, corresponding to a cardiac cycle of about 92 beats per minute. We adopted a time step equal to $\Delta t = 5.0 \cdot 10^{-4}$ s. A homogeneous Neumann boundary conditions is imposed at the outlet surfaces of both ICA and ECA.



(a) ECA flowrate fraction.

(b) ICA flowrate fraction.

Figure 4.12: Fraction of flow passed to ECA and ICA. Comparison between Newtonian and non-Newtonian models.

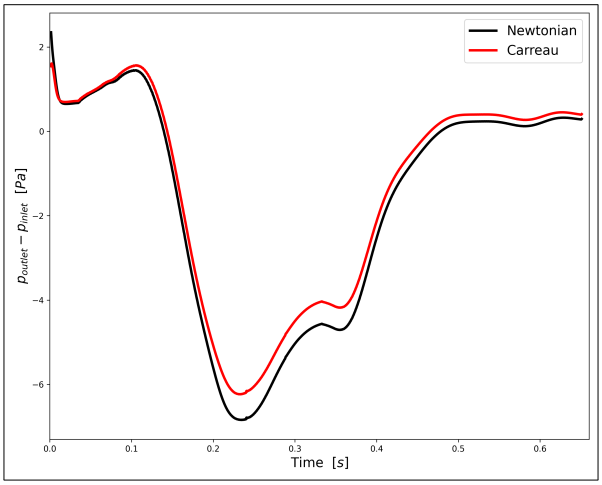


Figure 4.13: Pressure jump between inlet and outlet surfaces. Comparison between Newtonian and non-Newtonian models.

We performed two different simulations: in the first one we used the Newtonian model with dynamic viscosity $\mu_{newtonian}$=0.00345 Pa·s, while in the second one we employed the Carreau model (see Subsection 2.2.1 for details about the parameters). For both simulations we set the density to the value $\rho = 1060 kg/m^3$. Finally, we analyzed some quantities of interest in order to highlight the differences between the two models. In figure Figure 4.12 we can see how the model for the viscosity influences the flow partition at the bifurcation, while in Figure 4.13 we can see the difference between the two models in terms of pressure jump between the outlet surfaces and the inlet surface. For both quantities the differences provided by the two models seem to be significant. In particular, with the Newtonian model more blood is sent to ECA compared to the Carreau model and a higher pressure drop between inlet and outlet is observed, especially for intermediate time instants.

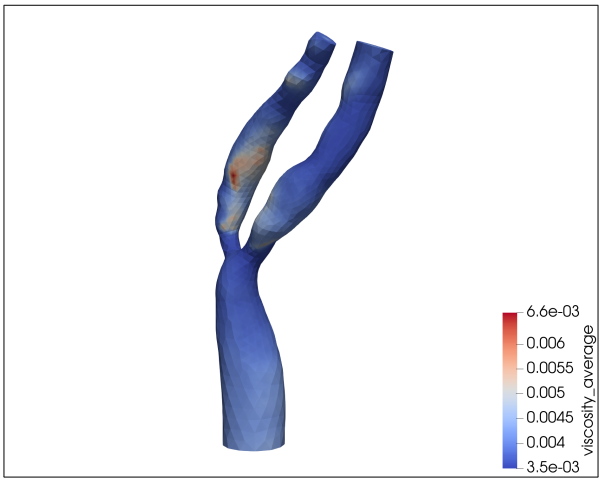
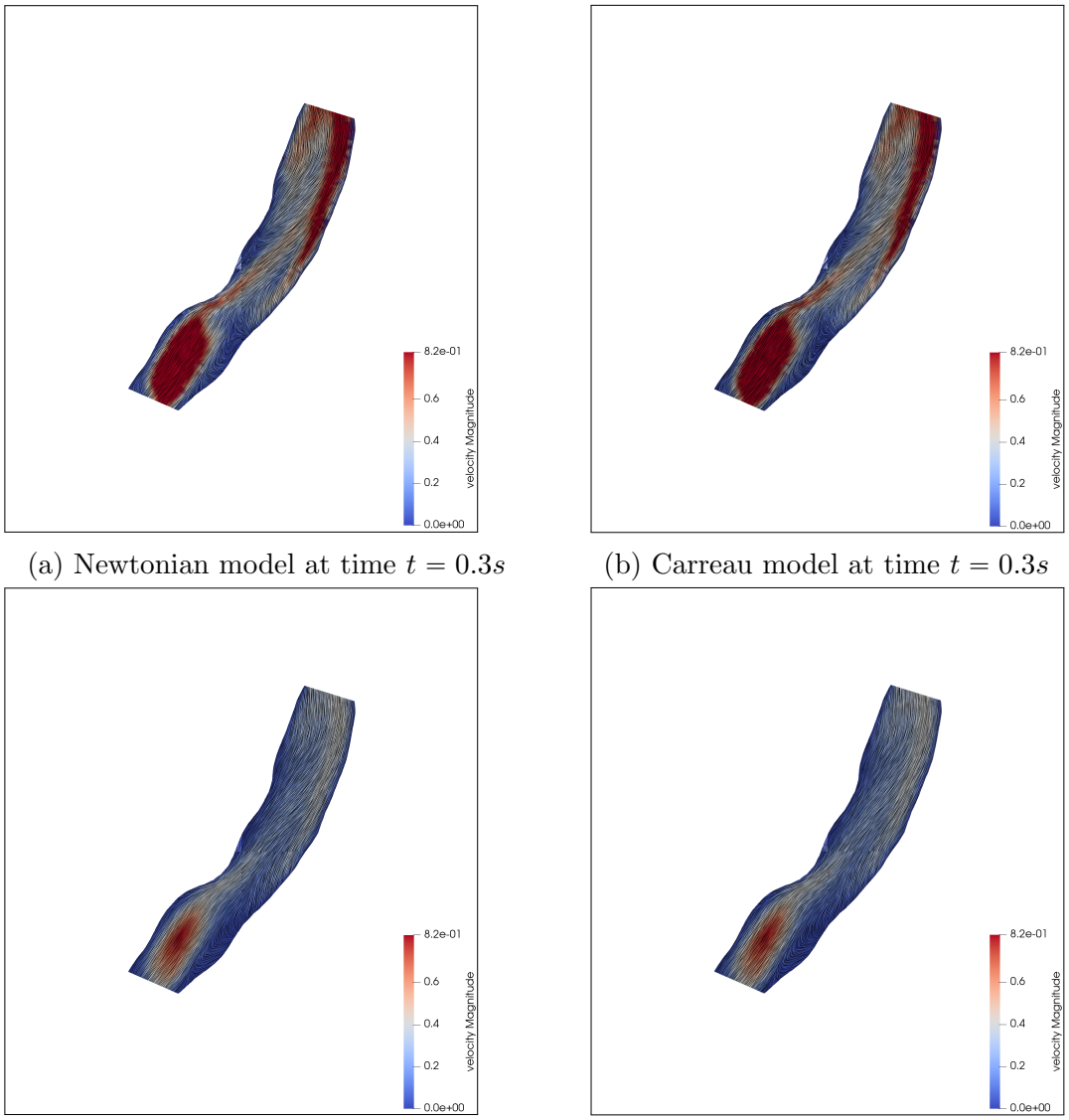


Figure 4.14: Time average distribution of viscosity in the case of non-Newtonian model.

Those difference are justified by the different distribution of the viscosity in the case of the non-Newtonian model. In Figure 4.14 we reported the distribution of the time averaged viscosity for the Carreau model and we can observe that the viscosity is almost everywhere greater than

the constant value employed in the Newtonian case and much higher in some specific areas. In particular, the higher viscosity in ECA can justify the lower amount of flow rate sent to that region compared to the Newtonian model: when the blood reaches the bifurcation, a greater resistance will come from the vessel characterized by a higher viscosity, thus it is easier for the blood to flow in the other vessel.



(a) Newtonian model at time $t = 0.3s$

(b) Carreau model at time $t = 0.3s$

(c) Newtonian model at time $t = 0.5s$

(d) Carreau model at time $t = 0.5s$

Figure 4.15: Velocity field on a slice of ICA. Comparison between Newtonian (left) and non-Newtonian (right) models at different time instants.

Finally, in Figure 4.15 we reported the velocity field at two different time instants for both models. In this specific case the model seems not to influence significantly the velocity field (at least at macroscopic level), but this could be expected since the resulting shear rate is relatively high.

5. Conclusions and further developments

In this work we introduced two new features related to the fluid dynamics of blood flow in the high-performance Finite Element library **life^x**:

1. the implementation of a non-Newtonian model in order to capture the variability of the blood viscosity,

2. the implementation of a solver for the inverse Womersley problem, which allows to impose a more appropriate Dirichlet boundary condition when a pulsatile blood flow is under study.

Developing the project in the framework of **life^x** required adhering to the coding guidelines of the library and participate in a developer community. During the implementation of the new features we took into accounts those coding guidelines while trying to keep the code as flexible as possible for future developments. We were careful in making all those changes understandable for future developers while giving to the users a flexible and clear interface.
This project gave us the opportunity to confront many times with other developers and constantly learn and work with git version control and automated testing pipelines. We are glad to mention that the branch developed for the implementation of the non-Newtonian model is already merged in the master branch of the official library and it is already used for different practical applications. Moreover, the merge of the branch for the inverse Womersley problem is in progress and should be completed in the near future. This merge is more complex since, as we explained in Section 3.3, it needs an external library in order to evaluate Bessel functions with complex argument. However, this gave us the opportunity to better understand the usage of dynamic libraries in a very extended and versatile library such as **life^x**, and to work on Cmakefiles.
Regarding the numerical results, in Chapter 4, after giving the details about how to set the parameters for a simulation, the validation of both models was performed. In particular, we compared the results obtained with both Newtonian and non-Newtonian models with the one reported in [30], finding a satisfactory correspondence for both the steady and pulsatile flow cases. About the second new feature, i.e. the imposition of the Womersley velocity profile as boundary condition on the inlet surface, we first validated the implementation making sure that the resulting flowrate matches the one given in the csv input file by the user and, secondly, we performed a robustness analysis introducing a random noise on the input flowrate, which was still properly reconstructed. Finally, both features were applied simultaneously to a real geometry: a bifurcated carotid, this was done in order to show that both models can be applied to a real case scenario. In particular, we would like to thank Prof. Christian Vergara for making available and providing the geometry of the stenotic carotid. The simulation of this last testcase was performed on a high performance computing cluster provided by Modelling and Scientific Computing (MOX) laboratory. This gave us the possibility to learn how to work with such advanced resources. Moreover, we highlight that, in order to run multiple simulations sequentially, we implemented a bash script that can be found in Appendix A. This was particularly useful when we had to study

different scenario in the cylindrical case. The script is also in charge of moving and renaming suitably the output files, simplifying the postprocessing.

Further developments on the subjects of this project can be done in multiple directions. First of all, the differences between Newtonian and non-Newtonian models could be deepened for different type of applications with possible comparison through experimental results. This could be useful in understanding in which scenario and for which quantities the effects of the non-Newtonian model are negligible or not.

Other types of non-Newtonian models could be investigated: some of them should be easy to implement starting from our implementation. In particular, other generalised non-Newtonian models such as Power Law [22], Carreau-Yasuda [15] and Casson [19] could be implemented and the effect of the specific model in different scenarios could be studied. Moreover, more complicated models could be analyzed and implemented, such as viscoelastic models [6] in which the viscous properties of the blood are taken into account together with the elastic properties.

Concerning the implementation, it could be useful to design a common interface for *FlowBC* and *WomersleyBC* since they both are in charge of imposing a Dirichlet boundary condition.

Finally, regarding the imposition of a Womersley velocity profile as inflow boundary condition, the case of elliptical inlet section could be studied: this can be useful in specific applications, such as cerebrospinal fluid flow [21].

A. A bash script for sequential executions

In the following we present a bash script we implemented to run multiple simulations sequentially. The user can select multiple options then the script will run a simulation for all the possible combinations of the parameters imposed. At the end of each simulation, all the output files are automatically moved and renamed in a suitable way.

```
#!/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 velocuty space
p_degree=1                                  # Dimension of pressure space
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
```

```
51
52 # Path to folders "Newtonian", "Non_Newtonian" and "Comparison"
53 dest_path=/path/where/to/store/solutions
54
55 # Path to folders "Newtonian", "Non_Newtonian" and "Comparison" inside
56 # "Womersley"
57 dest_path_Womersley=/path/where/to/store/solutions/when/bc/is/Womersley
58
59 # Path to file "lifex_test_fluid_dynamics_cylinder"
60 run_path=/path/to/executable/file
61
62 # Additional title for the folders where the solutions will be stored, it
63 # will appear at the end after a "-"
64 mark_path= to_characterize_solutions
65
66 # Set preliminary parameters (no multiple choice on these)
67 sed -i "s/    set Debug flag                =.*/    set Debug flag ↩
                    = $debug_viscosity/" $file_path
68 sed -i "s/    set Density                   =.*/    set Density ↩
                       = $density/" $file_path
69 sed -i "s/    set Velocity FE space degree =.*/    set Velocity FE space ↩
    degree = $vel_degree/" $file_path
70 sed -i "s/    set Pressure FE space degree =.*/    set Pressure FE space ↩
    degree = $p_degree/" $file_path
71 sed -i "s/    set Model                                =.*/    set Model ↩
                                 = $stabilization/" $file_path
72 sed -i "s/      set Time evolution     =.*  # Inlet time evolution/      set ↩
    Time evolution     = $inlet_type_time    # Inlet time evolution/" ↩
    $file_path
73 sed -i "s/      set Space distribution = .* # Inlet space function/      set ↩
    Space distribution = $inlet_type_space   # Inlet space function/" ↩
    $file_path
74 sed -i "s/        set Interpolation               =.*  #Dirichlet inlet ↩
    interpolation/        set Interpolation               = $interpolation  #↩
    Dirichlet inlet interpolation/" $file_path
75
76 # Set the refinement level
77 for ref in ${refinement[@]}
78 do
79
80 sed -i "s/    set Number of refinements    =.*/    set Number of refinements ↩
      = $ref       # default: 0/" $file_path
81
82    for model in ${model_flag[@]}
83    do
84
85    # Check the validity of the flag debug_viscosity
86    if [[ $debug_viscosity != 'true' && $debug_viscosity != 'false' ]]
87    then
88      echo 'ERROR: non valid viscosity flag'
89      break
90    fi
91
92    # Set the model_flag for the model (Newtonian or Carreau)
93      if [ $model == 'Newtonian' ]
94      then
95        sed -i "s/    set Non-Newtonian model flag.*/    set Non-Newtonian ↩
            model flag   = false/" $file_path
96      elif [ $model == 'Non_Newtonian' ]
97      then
98        sed -i "s/    set Non-Newtonian model flag.*/    set Non-Newtonian ↩
```

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

```
            do
              sed -i "s/    set Time step                =.*/    set Time step
                                  = $t_step/" $file_path

              # Set velocity flux if Dirichlet BC
              if [ $my_bc == 'Dirichlet' ]
              then

                   if [[ $inlet_type_space == 'Womersley' ]]
                     then
                        (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)
                        cp $dest_path_Womersley/$model/r$r-$my_bc-s$final_t-
                           vel$flux-$mark_path/fluid_dynamics.csv
                           $dest_path_Womersley/Comparison/Csv
                        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
                     else
                  for flux in ${velocity_flux[@]}
                  do
                    sed -i "s/        set Value =.* flow/        set Value =
                       $flux # flow/" $file_path

                    (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)
                    cp $dest_path/$model/r$r-$my_bc-s$final_t-vel$flux-
                       $mark_path/fluid_dynamics.csv $dest_path/Comparison/
                       Csv
                    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
                  done
                     fi

              # Set pressure otherwise
              else
                for p in ${pressure[@]}
                do
                  sed -i "s/        set Value =.* Inlet pressure/        set
                     Value = $p   # Inlet pressure/" $file_path

                   (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)
                    cp $dest_path/$model/r$r-$my_bc-s$final_t-p$p-$mark_path/
                       fluid_dynamics.csv $dest_path/Comparison/Csv
                    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
                done
              fi
            done
          done
        done
      done
    done
done
```

Listing A.1: Script to run multiple instances sequentially

Bibliography

[1] Boost c++ library. https://www.boost.org

[2] Complex_bessel library. https://github.com/joeydumont/complex_bessel

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

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

[5] A. Quarteroni, L. Dedè, A. Manzoni, C. Vergara: Mathematical modelling of the human cardiovascular system : data, numerical approximation, clinical applications. Cambridge University Press (2019)

[6] A. Ramiar, M.M. Larimi, A.A. Ranjbar: Investigation of blood flow rheology using second-grade viscoelastic model (phan-thien-tanner) within carotid artery. Bioeng Biomech. **19**, 27–41 (2017)

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

[8] 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)

[9] Bowman, F.: Introduction to Bessel functions (1958)

[10] 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)

[11] D. Arndt, W. Bangerth, B. Blais et al.: The deal.ii library, version 9.3. Journal of Numerical Mathematics 29.3 pp. 171–186 (2021)

[12] 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)

[13] 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)

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

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

[16] 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)

[17] 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)

[18] 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)

[19] J. Venkatesan, D.S. Sankar, K. Hemalatha, Y. Yatim: Mathematical analysis of Casson fluid model for blood rheology in stenosed narrow arteries. Journal of Applied Mathematics **2013** (2013)

[20] 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)

[21] L.C. Berselli, F. Guerra, B. Mazzolai, E. Sinibaldi: Pulsatile viscous flows in elliptical vessels and annuli: solution to the inverse problem, with application to blood and cerebrospinal fluid flow. SIAM Journal on Applied Mathematics **74**, 40–59 (2013)

[22] M. Hussain,S. Kar, R.R. Puniyani: Relationship between power law coefficients and major blood constituents affecting the whole blood viscosity. Journal of Biosciences **24**, 329–337 (1999)

[23] 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)

[24] 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)

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

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

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

[28] R.M. Lancellotti: Large eddy simulations in haemodynamics: models and applications. Phd Thesis, Politecnico di Milano (2015)

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

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

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

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

[33] 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)