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

Project for the course of Advanced Programming for Scientific Computing

---

Luca Caivano, Michele Precuzzi

Professor: Luca Formaggia

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

POLITECNICO
MILANO 1863

Table of contents

Introduction and Motivation

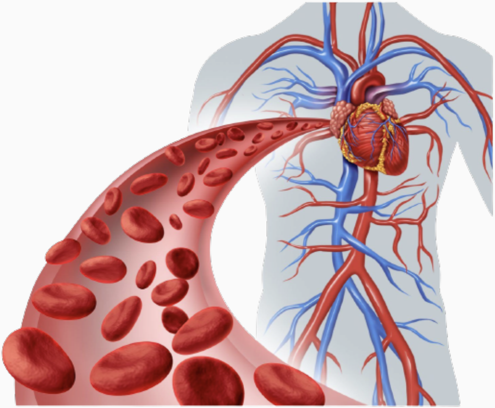


Figure: Cardiovascular system [2], **life**^{x} logo [7] and iHeart logo [6].

Introduction and Motivation

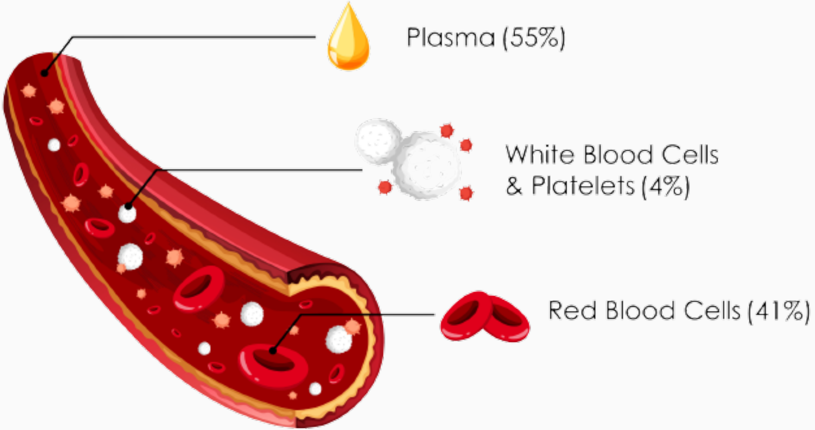


Figure: Composition of blood in vessels [4].

Mathematical modeling: the Navier-Stokes equations

Navier-Stokes equations

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

- $\mathbf{u}$ = velocity field
- p = pressure
- ρ = density
- $\varepsilon(\mathbf{u})$ = symmetric velocity gradient (rate-of-strain tensor)
- $\mathbf{f}$ = external force
- $\mathbf{g}, \mathbf{h}$ = boundary conditions
- μ = viscosity;

Mathematical modeling: the Navier-Stokes equations

Navier-Stokes equations

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

- $\mathbf{u}$ = velocity field
- p = pressure
- ρ = density
- $\varepsilon(\mathbf{u})$ = symmetric velocity gradient (rate-of-strain tensor)
- $\mathbf{f}$ = external force
- $\mathbf{g}, \mathbf{h}$ = boundary conditions
- μ = viscosity; if $\mu = \mu(\mathbf{u}) \Rightarrow$ **generalized Newtonian fluid**;

Mathematical modeling: Galerkin formulation

Navier-Stokes equations: Galerkin formulation

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 $\forall \mathbf{v}_h \in \mathbf{V}_h, q_h \in Q_h$ it holds:

$$\begin{cases} \int_{\Omega_h} \frac{1}{\Delta t}\alpha_{BDF}\mathbf{u}_h^{n+1} \cdot \mathbf{v}_h d\Omega_h + \mathcal{D}(\mathbf{u}_h^{n+1}, \mathbf{v}_h; \mu) + \int_{\Omega_h} [(\mathbf{u}_*{}^{n+1} \cdot \nabla)\mathbf{u}_h^{n+1}] \cdot \mathbf{v}_h d\Omega_h \\ - \int_{\Omega_h} p_h^{n+1} \nabla \cdot \mathbf{v}_h d\Omega_h = \int_{\Omega_h} \frac{1}{\Delta t}\mathbf{u}_{hBDF}^n \cdot \mathbf{v}_h d\Omega_h + \int_{\Omega_h} \mathbf{f}_h^n \cdot \mathbf{v}_h d\Omega_h + \int_{\Gamma_N} \mathbf{h}_h^{n+1} \cdot \mathbf{v}_h d\gamma \\ \int_{\Omega_h} q_h \nabla \cdot \mathbf{u}_h^{n+1} d\Omega_h = 0 \end{cases}$$

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

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

Carreau model for non-Newtonian fluids

Carreau formula for blood viscosity

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

$$\text{where } \dot{\gamma} = \sqrt{2\,\mathrm{tr}\left(\varepsilon(\boldsymbol{u})^2\right)},$$

$$\mu_\infty = 3.45 \times 10^{-3} Pa \cdot s, \quad \mu_0 = 5.6 \times 10^{-2} Pa \cdot s, \quad \lambda = 3.313\,\mathrm{s}, \quad n = 0.3568$$

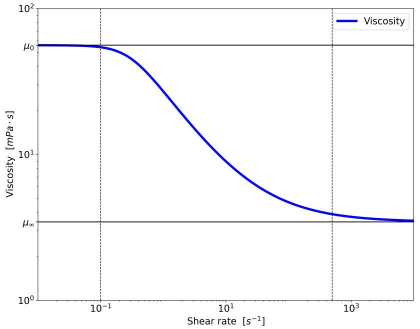


Figure: Relation between shear rate and dynamic viscosity according to Carreau model

Implementation of non-Newtonian model

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

- The function `parse_parameters` and `declare_parameters` had to be modified in order to add a new section in the prm file containing the parameters for the non-Newtonian model.
- The quantities that are used in the non-Newtonian model are computed at each quadrature node.

```
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];
  }
```

Implementation of non-Newtonian model- Carreau formula

- We implemented a function to compute the viscosity according to the non-Newtonian model:

```
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);
}
```

Implementation of non-Newtonian model

- The viscosity at every quadrature point is 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;
  }
```

- Wherever the viscosity has to be used for the calculation of some quantities the value computed according to 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;
```

Validation of the Non-Newtonian model- Steady flow

We have validated our results comparing them to the ones obtained in [9]. We performed some tests assuming first the Newtonian and then the non-Newtonian model in two cylinders with radii $R_1 = 3.1cm$ and $R_2 = 4.5cm$.

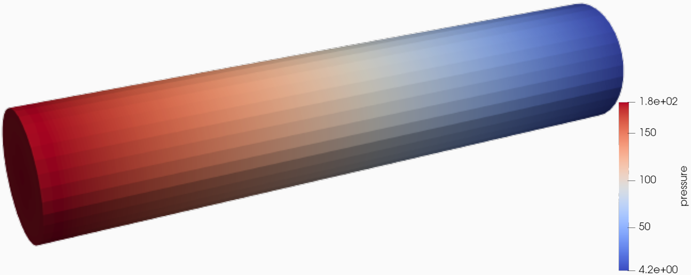


Figure: Pressure imposed as Neumann boundary condition

Validation of the Non-Newtonian model- Steady flow

	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: Comparison with the reference results for $R_1 = 3.1cm$

	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: Comparison with the reference results for $R_2 = 4.5cm$

Validation of the Non-Newtonian model- Steady flow

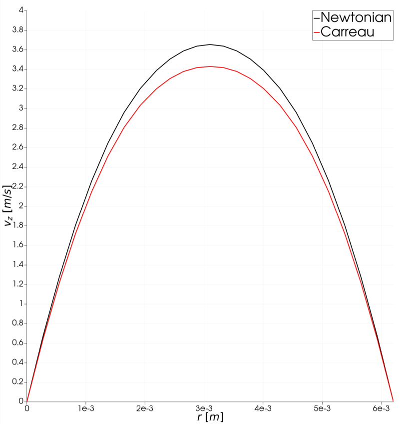


Figure: Velocity profile for $R_1 = 3.1cm$

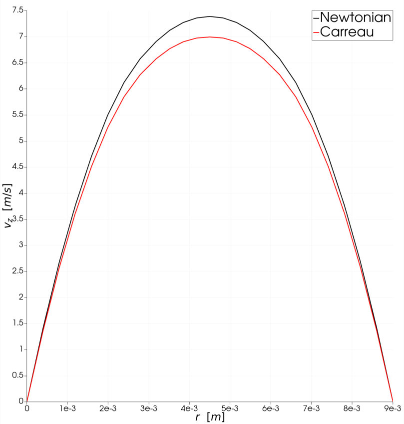


Figure: Velocity profile for $R_2 = 4.5cm$

Validation of the Non-Newtonian model- Pulsatile flow

	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: Comparison of the max flow rate with the reference results for $R_1 = 3.1cm$

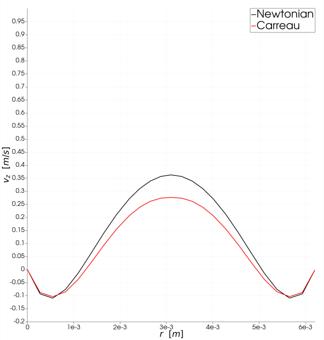


Figure: Min velocity profile for $R_1 = 3.1cm$

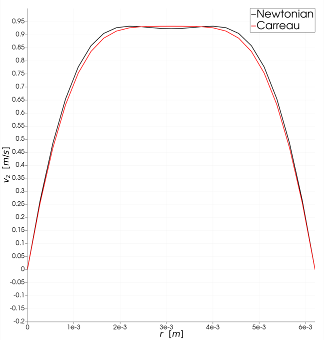


Figure: Max velocity profile for $R_1 = 3.1cm$

Validation of the Non-Newtonian model- Pulsatile flow

	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: Comparison of the max flow rate with the reference results for $R_2 = 4.5cm$

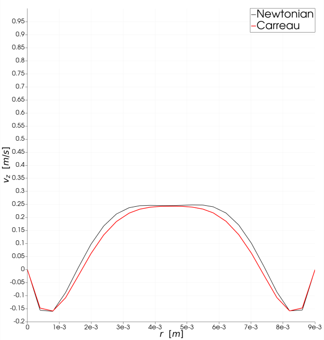


Figure: Min velocity profile for $R_2 = 4.5cm$

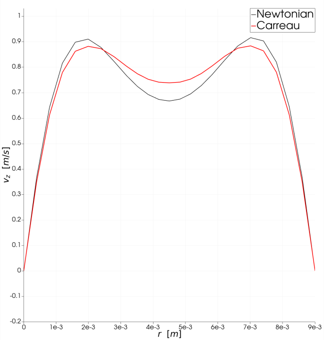


Figure: Max velocity profile for $R_2 = 4.5cm$

Viscosity distributions at minimum flowrate

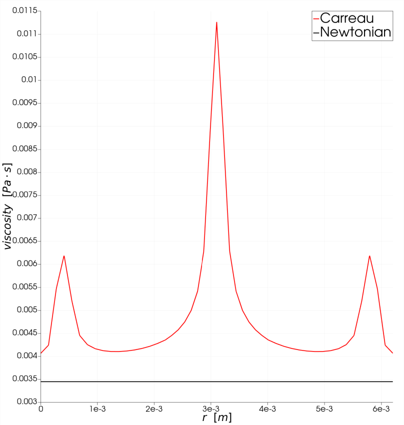


Figure: Viscosity distribution corresponding to minimum flowrate for $R_1 = 3.1cm$

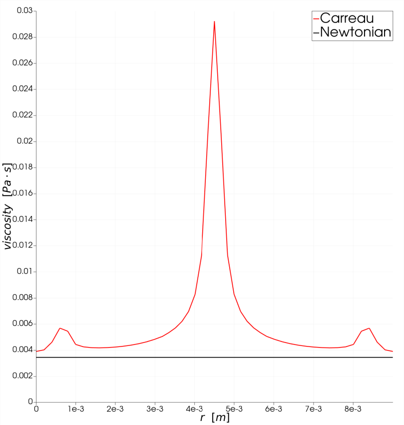


Figure: Viscosity distribution corresponding to minimum flowrate for $R_2 = 4.5cm$

Womersley velocity profiles

They are the solutions to the Navier-Stokes equations in a cylinder when an oscillating pressure gradient $\frac{\partial p}{\partial z} = Ae^{jnt}$ is imposed.

Womersley velocity profile

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

- j = imaginary unit
- R = radius of the cylinder
- J_0 = Bessel function of the first kind of order 0

Womersley velocity profiles

Womersley velocity profile

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

The **Womersley number** $\mathrm{Wo}(r) = r\sqrt{\frac{n}{\nu}}$ expresses the ratio between inertial oscillatory forces and viscous forces.

- $\mathrm{Wo} \leq 1 \Rightarrow$ **parabolic velocity profile**
- $\mathrm{Wo} \geq 10 \Rightarrow$ **flat velocity profile**
- $1 < \mathrm{Wo} < 10 \Rightarrow$ none of the previous approximations are suitable and the above formula should be used

Womersley velocity profiles

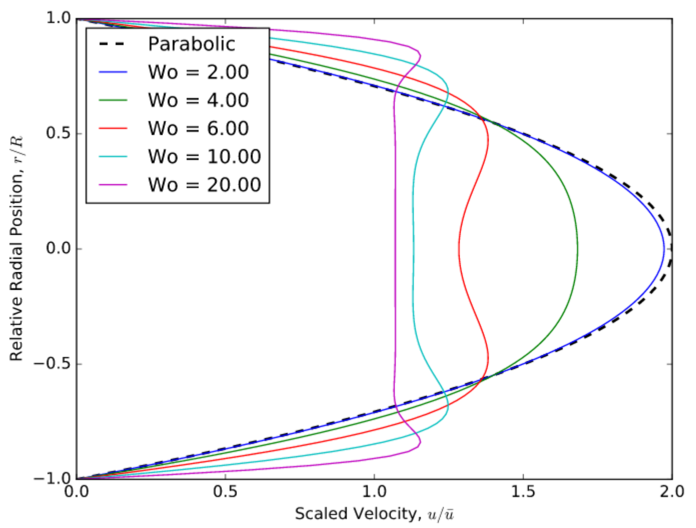


Figure: Velocity profiles for different values of Wo

Inverse Womersley problem

Given a flow rate $q\,(t)$ can we reconstruct the associated **Womersley velocity profiles** ?

Inverse Womersley problem

Given a flow rate $q(t)$ can we reconstruct the associated **Womersley velocity profiles** ?

Yes! For this purpose we approximate the flow rate $q(t)$, the pressure gradient $\sigma(t)$ and the axial velocity $v(r,t)$ with **truncated Fourier expansions**:

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

and we look for a relation among the **modes** q_n, σ_n and $v_n(r)$.

Inverse Womersley problem

Given a flow rate $q(t)$ can we reconstruct the associated **Womersley velocity profiles** ?

Yes! For this purpose we approximate the flow rate $q(t)$, the pressure gradient $\sigma(t)$ and the axial velocity $v(r,t)$ with **truncated Fourier expansions**:

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

and we look for a relation among the **modes** q_n, σ_n and $v_n(r)$.

Note: the coefficient q_0 corresponds to a steady flow, thus we can recover σ_0 and $v_0(r)$ by the **Poiseuille law**. Moreover, in order to end up with real-valued solutions, $v_{-n}(r)$ must be the **complex conjugate** of $v_n(r)$

Inverse Womersley problem: $q_n \mapsto \sigma_n \mapsto v_n(r)$

Map $q_n \mapsto \sigma_n$

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

where

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

denotes the regularized confluent hyper-geometric limit function.

Inverse Womersley problem: $q_n \mapsto \sigma_n \mapsto v_n\,(r)$

Map $q_n \mapsto \sigma_n$

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

where

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

denotes the regularized confluent hyper-geometric limit function.

Map $\sigma_n \mapsto v_n\,(r)$

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

where J_0 denotes the Bessel function of first kind of order zero.

Inverse Womersley problem: implementation

The class *FlowBC* imposes Dirichlet boundary conditions that are represented by functions **separable in space and time**.

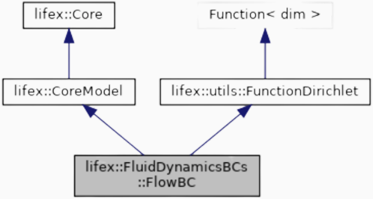


Figure: *FlowBC* inheritance diagram

Inverse Womersley problem: implementation

The class *FlowBC* imposes Dirichlet boundary conditions that are represented by functions **separable in space and time**. This is not the case for the Womersley velocity profiles, thus we needed to implement a new class, called *WomersleyBC*

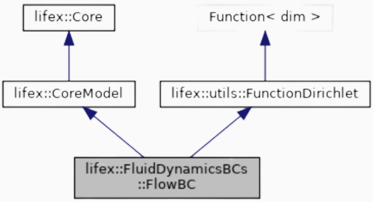


Figure: *FlowBC* inheritance diagram

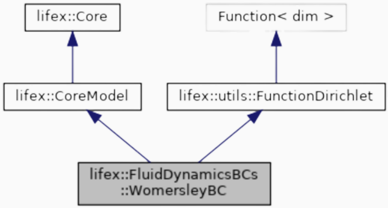


Figure: *WomersleyBC* inheritance diagram

In the *WomersleyBC* class we defined a subclass, called *WomersleyMode*, to handle inidividually each Fourier mode.

Inverse Womersley problem: implementation

WomersleyBC relevant members

- `std::vector<WomersleyMode>` to handle individually the Fourier modes $v_n(r)$
- shared pointer to a class containing the informations about the input flow rate, passed through a csv file
- a method, called `vector_value`, to impose the Womersley velocity profile as Dirichlet boundary condition

Inverse Womersley problem: implementation

WomersleyBC relevant members

- `std::vector<WomersleyMode>` to handle individually the Fourier modes $v_n(r)$
- shared pointer to a class containing the informations about the input flow rate, passed through a csv file
- a method, called `vector_value`, to impose the Womersley velocity profile as Dirichlet boundary condition

WomersleyMode relevant members

- two `double` to store real and imaginary parts of the flow rate mode q_n, together with a `size_t` to store the index n
- a method, called `value`, designed to compute the mode $v_n(r)$ exploiting the maps $q_n \mapsto \sigma_n$ and $\sigma_n \mapsto v_n(r)$

Inverse Womersley problem: `WomersleyMode::value`

```cpp
std::complex<double>  WomersleyBC::WomersleyMode::value(const
          Point<dim> & p, const unsigned int component) const
  {
        // ... members declaration and initialization ... //

    // Evaluate hypergeometric functions
    const std::complex<double> f_11 = sp_bessel::besselJ(0, 2.0 *
                                      std::sqrt(-arg_bessel));
    const std::complex<double> f_12 = (std::pow(-arg_bessel, -1 /
    2.0))*(sp_bessel::besselJ(1, 2.0 *std::sqrt(-arg_bessel)));

   // Compute sigma_n inverting the map q_n -> sigma_n
    const std::complex<double> sigma_n = (1i * omega_n * q_n) /
    ((M_PI * boundary_radius * boundary_radius) * (1 - f_12 / f_11));

    // Evaluate Bessel functions
    const std::complex<double> j_0R = sp_bessel::besselJ(0, 0.5 *
    (std::complex<double>(-std::sqrt(2), std::sqrt(2))) * wo_R);
    const std::complex<double> j_0r = sp_bessel::besselJ(0, 0.5 *
    (std::complex<double>(-std::sqrt(2), std::sqrt(2))) * wo_r);

    // Compute and return v_n exploiting the map sigma_n -> v_n
    return sigma_n / (1i * omega_n) * (1 - j_0r / j_0R);
  }
```

Inverse Womersley problem: WomersleyMode::value

```
std::complex<double>  WomersleyBC::WomersleyMode::value(const
           Point<dim> & p, const unsigned int component) const
  {
        // ... members declaration and initialization ... //

    // Evaluate hypergeometric functions
    const std::complex<double> f_11 = sp_bessel::besselJ(0, 2.0 *
                                      std::sqrt(-arg_bessel));
    const std::complex<double> f_12 = (std::pow(-arg_bessel, -1 /
    2.0))*(sp_bessel::besselJ(1, 2.0 *std::sqrt(-arg_bessel)));

   // Compute sigma_n inverting the map q_n -> sigma_n
    const std::complex<double> sigma_n = (1i * omega_n * q_n) /
    ((M_PI * boundary_radius * boundary_radius) * (1 - f_12 / f_11));

    // Evaluate Bessel functions
    const std::complex<double> j_0R = sp_bessel::besselJ(0, 0.5 *
    (std::complex<double>(-std::sqrt(2), std::sqrt(2))) * wo_R);
    const std::complex<double> j_0r = sp_bessel::besselJ(0, 0.5 *
    (std::complex<double>(-std::sqrt(2), std::sqrt(2))) * wo_r);

    // Compute and return v_n exploiting the map sigma_n -> v_n
    return sigma_n / (1i * omega_n) * (1 - j_0r / j_0R);
  }
```

Inverse Womersley problem: WomersleyMode::value

```
std::complex<double>  WomersleyBC::WomersleyMode::value(const
           Point<dim> & p, const unsigned int component) const
  {
      // ... members declaration and initialization ... //

    // Evaluate hypergeometric functions
    const std::complex<double> f_11 = sp_bessel::besselJ(0, 2.0 *
                                      std::sqrt(-arg_bessel));
    const std::complex<double> f_12 = (std::pow(-arg_bessel, -1 /
    2.0))*(sp_bessel::besselJ(1, 2.0 *std::sqrt(-arg_bessel)));

   // Compute sigma_n inverting the map q_n -> sigma_n
    const std::complex<double> sigma_n = (1i * omega_n * q_n) /
    ((M_PI * boundary_radius * boundary_radius) * (1 - f_12 / f_11));

    // Evaluate Bessel functions
    const std::complex<double> j_0R = sp_bessel::besselJ(0, 0.5 *
    (std::complex<double>(-std::sqrt(2), std::sqrt(2))) * wo_R);
    const std::complex<double> j_0r = sp_bessel::besselJ(0, 0.5 *
    (std::complex<double>(-std::sqrt(2), std::sqrt(2))) * wo_r);

    // Compute and return v_n exploiting the map sigma_n -> v_n
    return sigma_n / (1i * omega_n) * (1 - j_0r / j_0R);
  }
```

Inverse Womersley problem: WomersleyBC::vector_value

```
void WomersleyBC::vector_value(const Point<dim> &p, Vector<double>
                                &values) const
  {
    // ... members declaration and initialization ... //

    sigma0 = ... // Poiseouille law
    v0 =      ... // Poiseouille law

    velocity += v0;

    for (size_t n = 1; n <= M; ++n)
      {
        v_n       = modes[n - 1].value(p); // compute v_n
        v_n_conj = std::conj(v_n);         // compute v_{-n}
        velocity += v_n * std::exp(1i * static_cast<double>(n) *
                    t_mod) + v_n_conj * std::exp(-1i * static_cast<
                    double>(n) * t_mod); // update velocity
      }

    for (unsigned int j = 0; j < dim; ++j)
      {
          values[j] = -1.0 * velocity.real() * scaling_factor *
                      this->get_normal_vector()[j];
      }
  }
```

Inverse Womersley problem: `WomersleyBC::vector_value`

```
void WomersleyBC::vector_value(const Point<dim> &p, Vector<double>
                                &values) const
  {
    // ... members declaration and initialization ... //

    sigma0 = ... // Poiseouille law
    v0 =     ... // Poiseouille law

    velocity += v0;

    for (size_t n = 1; n <= M; ++n)
      {
        v_n      = modes[n - 1].value(p); // compute v_n
        v_n_conj = std::conj(v_n);        // compute v_{-n}
        velocity += v_n * std::exp(1i * static_cast<double>(n) *
                    t_mod) + v_n_conj * std::exp(-1i * static_cast<
                    double>(n) * t_mod); // update velocity
      }

    for (unsigned int j = 0; j < dim; ++j)
      {
          values[j] = -1.0 * velocity.real() * scaling_factor *
                      this->get_normal_vector()[j];
      }
  }
```

Inverse Womersley problem: `WomersleyBC::vector_value`

```
void WomersleyBC::vector_value(const Point<dim> &p, Vector<double>
                                &values) const
  {
    // ... members declaration and initialization ... //

    sigma0 = ... // Poiseouille law
    v0 =     ... // Poiseouille law

    velocity += v0;

    for (size_t n = 1; n <= M; ++n)
      {
        v_n      = modes[n - 1].value(p); // compute v_n
        v_n_conj = std::conj(v_n);        // compute v_{-n}
        velocity += v_n * std::exp(1i * static_cast<double>(n) *
                    t_mod) + v_n_conj * std::exp(-1i * static_cast<
                    double>(n) * t_mod); // update velocity
      }

    for (unsigned int j = 0; j < dim; ++j)
      {
          values[j] = -1.0 * velocity.real() * scaling_factor *
                      this->get_normal_vector()[j];
      }
  }
```

Inverse Womersley problem: *FluidDynamics*

Thanks to **polymorphism**, the *FunctionDirichlet* part of either *FlowBC* or *WomersleyBC* is employed

```
// 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})));

if (prm_inlet_type == "Dirichlet")
{
  if (prm_womersley)
    bcs_dir.emplace_back(prm_tag_inlet,
                         inlet_womersley_dirichlet_bc,
                         ComponentMask({true, true, true,
                         false}));

  else
    bcs_dir.emplace_back(prm_tag_inlet,
                         inlet_dirichlet_bc,
                         ComponentMask({true, true, true,
                         false}));
}
```

Inverse Womersley problem: numerical results

We passed as input a periodic flow rate $q(t)$, solved the **inverse Womersley problem** and compared the generated flow rate with the original one.

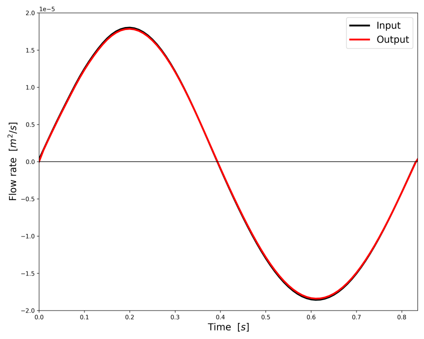


Figure: Flow rate reconstruction (no noise)

Inverse Womersley problem: numerical results

We passed as input a periodic flow rate $q(t)$, solved the **inverse Womersley problem** and compared the generated flow rate with the original one.

We repeated the test introducing a random noise to check the **robustness** of the procedure.

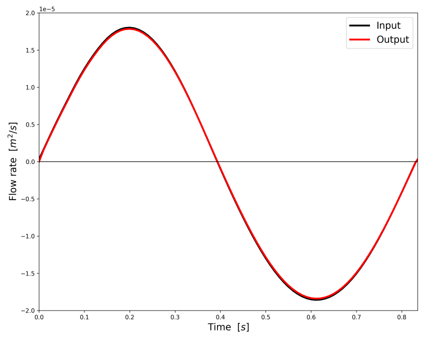


Figure: Flow rate reconstruction (no noise)

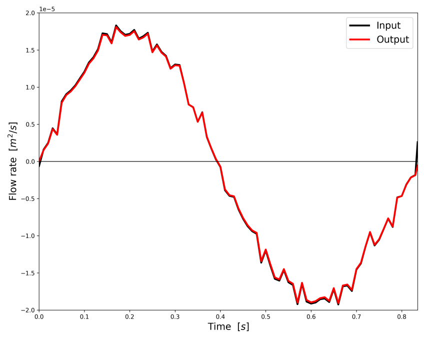


Figure: Flow rate reconstruction (with noise)

Inverse Womersley problem: numerical results

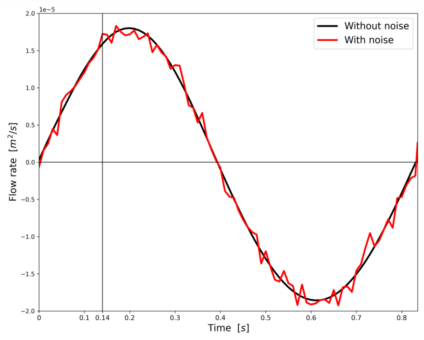


Figure: Flow rates with and without noise

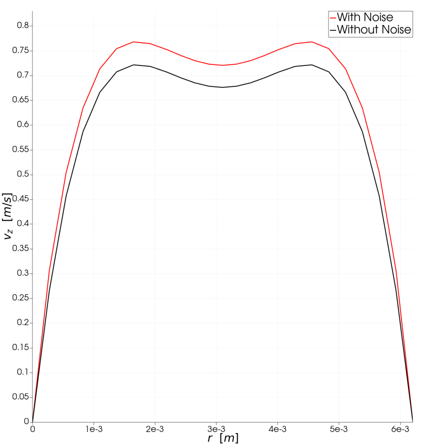


Figure: Velocity profiles at $t = 0.14s$

Inverse Womersley problem: numerical results

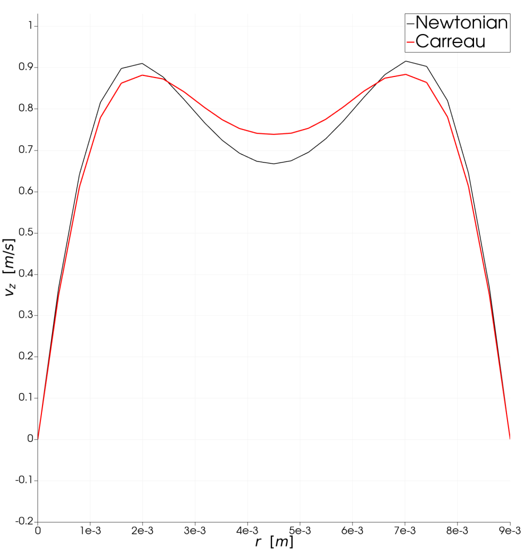
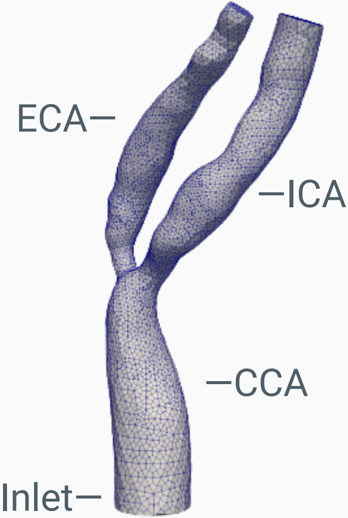


Figure: Max velocity profile for $R = 4.5cm$

Simulation on a bifurcated carotid



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

Figure: Geometry and hexahedral mesh for a bifurcated carotid.

Simulation on a bifurcated carotid- Inlet

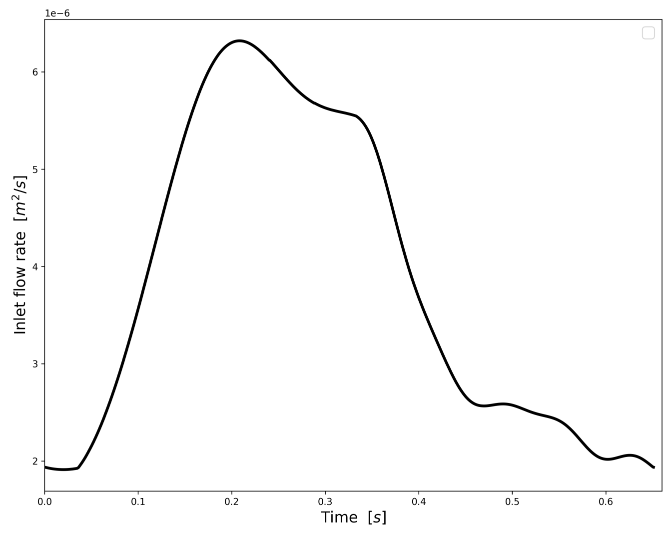


Figure: Flowrate imposed at the inlet surface as Dirichlet boundary condition.

Simulation on a bifurcated carotid- Partition of flux

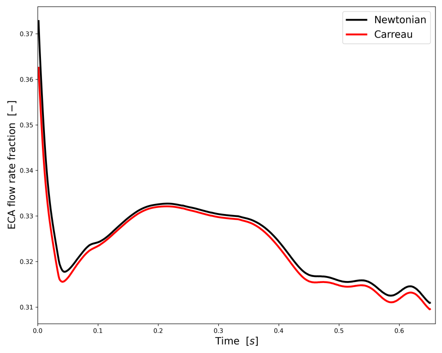


Figure: Fraction of flow passed to ECA.

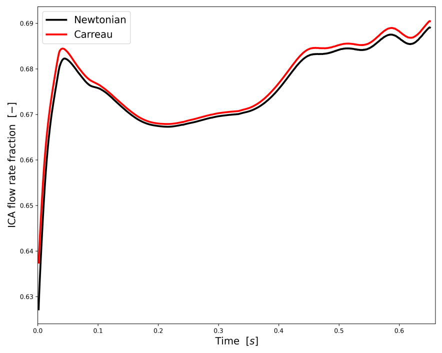


Figure: Fraction of flow passed to ICA.

Simulation on a bifurcated carotid- Pressure jump

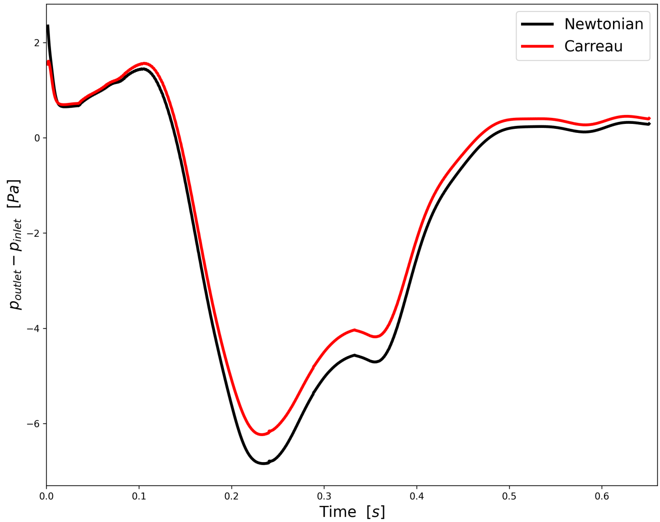


Figure: Pressure jump between inlet and outlet surfaces.

Simulation on a bifurcated carotid- Viscosity mean

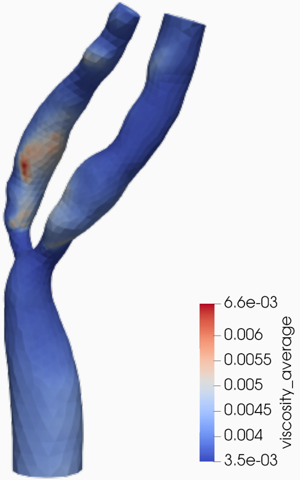


Figure: Time average distribution of viscosity for non-Newtonian model.

Conclusions

Additional skills acquired during the project:

- The project was developed adhering to the coding guidelines of the library and participating in a developer community.

Conclusions

Additional skills acquired during the project:

- The project was developed adhering to the coding guidelines of the library and participating in a developer community.
- Understanding of the usage of dynamic libraries in a very extended and versatile library such as **life^x** and learned to work on Cmakefiles.

Conclusions

Additional skills acquired during the project:

- The project was developed adhering to the coding guidelines of the library and participating in a developer community.
- Understanding of the usage of dynamic libraries in a very extended and versatile library such as **life^{x}** and learned to work on Cmakefiles.
- Understanding of the usage of a high performance computing cluster provided by MOX laboratory in order to perform computationally demanding simulations.

Conclusions

Additional skills acquired during the project:

- The project was developed adhering to the coding guidelines of the library and participating in a developer community.
- Understanding of the usage of dynamic libraries in a very extended and versatile library such as **life**$^{\mathbf{x}}$ and learned to work on Cmakefiles.
- Understanding of the usage of a high performance computing cluster provided by MOX laboratory in order to perform computationally demanding simulations.
- Implementation of a bash script in order to perform sequential simulations.

Further developments

- The differences between Newtonian and non-Newtonian models could be deepened for different type of applications with possible comparison through experimental results.

Further developments

- The differences between Newtonian and non-Newtonian models could be deepened for different type of applications with possible comparison through experimental results.
- Other types of non-Newtonian models could be investigated: some of them should be easy to implement starting from our implementation.
- More complicated models could be analyzed and implemented, such as viscoelastic models in which both the viscous and the elastic properties of the blood are taken into account.

Further developments

- The differences between Newtonian and non-Newtonian models could be deepened for different type of applications with possible comparison through experimental results.
- Other types of non-Newtonian models could be investigated: some of them should be easy to implement starting from our implementation.
- More complicated models could be analyzed and implemented, such as viscoelastic models in which both the viscous and the elastic properties of the blood are taken into account.
- Regarding the inverse Womersley problem, the case of elliptical inlet section could be studied: this can be useful in specific applications, such as cerebrospinal fluid flow.

Further developments

- The differences between Newtonian and non-Newtonian models could be deepened for different type of applications with possible comparison through experimental results.
- Other types of non-Newtonian models could be investigated: some of them should be easy to implement starting from our implementation.
- More complicated models could be analyzed and implemented, such as viscoelastic models in which both the viscous and the elastic properties of the blood are taken into account.
- Regarding the inverse Womersley problem, the case of elliptical inlet section could be studied: this can be useful in specific applications, such as cerebrospinal fluid flow.
- A common interface for FlowBC and WomersleyBC could be designed since they both are in charge of imposing a Dirichlet boundary condition.

Essential bibliography

[1] 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”. In: *Applied Mathematics and Computations* 219 (2013), pp. 5717–5729.

[2] *Cardiovascular system image*. URL: `https://askthescientists.com/cardiovascular-system-overview/`.

[3] 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”. In: *Biorheology* (1991), pp. 241–262.

[4] *Composition of blood image*. URL: `https://www.freepik.com/vectors/cell-biology`.

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

[6] *iHeart*. URL: `https://iheart.polimi.it`.

[7] *lifex library documentation*. URL: `https://lifex.gitlab.io`.

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

[9] Tabakova S., Nikolova E., Radev S. “Carreau Model for Oscillatory Blood Flow in a Tube”. In: *AIP Conference Proceedings* (2014), pp. 336–343.

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

Bonus slides

Navier-Stokes equations: Galerkin formulation

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 $\forall \mathbf{v}_h \in \mathbf{V}_h, q_h \in Q_h$ it holds:

$$\begin{cases} \int_{\Omega_h} \frac{1}{\Delta t} \alpha_{BDF} \mathbf{u}_h^{n+1} \cdot \mathbf{v}_h d\Omega_h + \mathcal{D}(\mathbf{u}_h^{n+1}, \mathbf{v}_h; \mu) + \int_{\Omega_h} [(\mathbf{u}_*{}^{n+1} \cdot \nabla)\mathbf{u}_h^{n+1}] \cdot \mathbf{v}_h d\Omega_h \\ - \int_{\Omega_h} p_h^{n+1} \nabla \cdot \mathbf{v}_h d\Omega_h = \int_{\Omega_h} \frac{1}{\Delta t} \mathbf{u}_{hBDF}^n \cdot \mathbf{v}_h d\Omega_h + \int_{\Omega_h} \mathbf{f}_h^n \cdot \mathbf{v}_h d\Omega_h + \int_{\Gamma_N} \mathbf{h}_h^{n+1} \cdot \mathbf{v}_h d\gamma \\ \int_{\Omega_h} q_h \nabla \cdot \mathbf{u}_h^{n+1} d\Omega_h = 0 \end{cases}$$

- $\Omega_h =$ discretized mesh of Ω
- $\mathbf{V}_h, Q_h =$ discrete test spaces for **velocity** and **pressure** respectively
- Given $0 = t_0 < t_1 < \cdots < t_N = T$ such that $t_{k+1} - t_k = \Delta t \; \forall k \geq 0$, we define $v^n(x) := v(x, t_n) \;\; \forall n \in \{0, 1, \ldots, N\}$
- The time derivative is approximated according to a BDF scheme:

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

Simulation on a bifurcated carotid- Velocity profile ICA

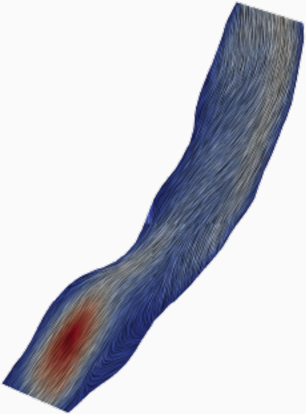


Figure: **Newtonian model**: velocity field on a slice of ICA ($t = 0.5s$).

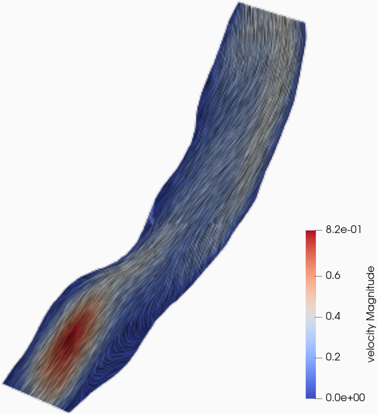


Figure: **Non-Newtonian model**: velocity field on a slice of ICA ($t = 0.5s$).

Projection of the viscosity in the output

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;
  }
```

In every cell, this output quantity is computed as the L^2 projection onto the finite element space. Every plot of the viscosity is obtained thanks to this functionality.