Advanced Programming for Scientific Computing

Hierarchical Model reduction on bifurcations:
a novel approach using POD and Domain Decomposition

Advisor: Professor Simona Perotto
Co-advisor: Yves Antonio Brandes Costa Barbosa

Authors: Luca Zampieri, Andrea Zanoni

Academic year 2018-2019
## Contents

1 Introduction .................................................. 7  
2 Theory ........................................................... 9  
   2.1 Advection-Diffusion-Reaction (ADR) problem ......................... 9  
   2.2 Hierarchical Model (HiMod) reduction method ......................... 11  
      2.2.1 HiMod discretization for parallelepipeds ..................... 13  
      2.2.2 HiMod discretization for cylinders .......................... 14  
   2.3 Proper Orthogonal Decomposition (POD) method ....................... 15  
      2.3.1 Computation of the reduced basis ............................ 17  
   2.4 Domain Decomposition (DD) method ................................ 17  
      2.4.1 Dirichlet-Neumann method ................................. 18  
   2.5 Recap ......................................................... 19  
3 Code .............................................................. 21  
   3.1 Code structure ................................................. 21  
      3.1.1 Core/ folder .............................................. 22  
      3.1.2 gmsh/ folder ............................................ 23  
      3.1.3 TestSuite/ folder ....................................... 23  
   3.2 Installation ................................................. 24  
   3.3 Main classes .................................................. 26  
      3.3.1 Matlab .................................................. 26  
      3.3.2 C++ .................................................. 31  
   3.4 Working Flow .................................................. 35  
      3.4.1 A look at main.m file .................................... 35  
      3.4.2 Going through the DD.Solver.solve() function ............. 38  
   3.5 Reproducing results ........................................... 39  
   3.6 Licensing & rights ............................................ 39  
4 Numerical results ................................................. 41  
   4.1 Geometries .................................................. 42  
      4.1.1 Parallelepiped ........................................... 42  
      4.1.2 Cylinder ............................................... 44  
      4.1.3 Bifurcation with rectangular section ......................... 45  
   4.2 Domain Decomposition iterations .................................. 48  
   4.3 POD analysis ................................................ 49  
5 Conclusion ...................................................... 51
A Modifications file by file
   A.1 C++ code ................................................................. 53
   A.2 Matlab code ............................................................ 55

B Tree Structure ......................................................... 57

Acknowledgement ....................................................... 61

Bibliography ............................................................ 63
Abstract — In this work we explore a way to expand the Hierarchical Model Reduction (HiMod) to bifurcated domain by solving the bifurcation itself with Proper Orthogonal Decomposition (POD) and linking the solution through Domain Decomposition (DD). An implementation coupling Matlab and C++ is also provided.

Index terms — Numerical analysis, Hierarchical Model Reduction (HiMod), Proper Orthogonal Decomposition (POD), Domain Decomposition (DD), Redbkit, lifeV

Dinanzi a me non fuor cose create
se non eterne, e io eterno duro.
Before me there were no created things,
Only eterne, and I eternal last.

Dante Alighieri
Chapter 1

Introduction

Many engineering problems exhibit a spatial dimension which is dominant with respect to the others. One of the most important examples being haemodynamics, where the dominant direction is given by the blood flow. Transverse directions are less important than the dominant one and a full 3D simulation can be very expensive from a computational point of view. In this case a reduced order model can be very useful to reduce the computational cost while preserving most of the accuracy of the solution. The basic concept of the hierarchical model reduction method (HiMod) is to employ different discretization techniques along the predominant and the transverse directions. The variables are separated via a generalized Fourier expansion along the transverse direction, thus reducing a fully 3D problem into a system of 1D problems coupled by the transverse information. The actual unknowns are the coefficients of the Fourier expansion, that are mono-dimensional functions and can be discretized using 1D finite elements.

HiMod has already been studied and applied to haemodynamics problems, providing good results [7] [3]. However, it has been noticed that it cannot be employed in domains which present bifurcations and tackling this issue is very important in order to be able to approximate the whole circulatory system. An idea, proposed by Professor Perotto and colleagues, to overcame this restriction is to decompose the computational domain, between branches and the bifurcation itself: in the branches we can still apply HiMod and in the actual bifurcation we have to employ a different technique. In order to have an accurate solution also in the bifurcation, but with the aim of maintaining a contained computational cost, we can use 3D finite elements coupled with a reduced basis method, like proper orthogonal decomposition (POD) [1] [2].

Reduced basis methods consider parameterized problems and consist on projecting the problem onto a reduced space containing the most relevant features of the phenomena. In order to compute a basis of this new space, several problems assigning different values to the parameters have to be solved, providing a large number of high-fidelity solutions, then in POD we apply singular value decomposition to a matrix constructed using these solutions.

In this work we want to experiment this idea, starting from the simplest cases. Let $\Omega$ be a domain and $\Gamma_{in}$, $\Gamma_{out}$ and $\Gamma_{lat}$ be respectively the inflow, outflow and lateral boundary, then we want to solve the following Advection-Diffusion-Reaction problem:
\[
-\mu \Delta u + \mathbf{b} \cdot \nabla u + \sigma u = f \quad \text{in } \Omega \\
u = u_{in} \quad \text{on } \Gamma_{in} \\
\mu \nabla u \cdot \mathbf{n} = 0 \quad \text{on } \Gamma_{out} \\
u = 0 \quad \text{on } \Gamma_{lat}.
\]

Initially, the simplest geometries are considered, a parallelepiped and a cylinder, and they are decomposed in three smaller parts. For both cases, HiMod is employed in the left and right parts and POD is used in the middle part. Finally, and most importantly, the computation of the solution in a domain containing a bifurcation is considered. The bifurcation is simple, it is obtained as the union of four smaller parallelepipeds and similarly to the previous cases, HiMod is applied to the external parts and POD is used in the middle part.

The code for the HiMod method is written in C++ using the LifeV library, while the code for the POD method is written in Matlab using the redbKIT library. In order to combine these two techniques, domain decomposition (DD) is employed, more precisely the Dirichlet-Neumann method and the code for this part is written in Matlab. The communication of the boundary conditions between C++ and Matlab is handled using text files.

The work is organized in three chapters

- Chapter 2: introduces the problem at hand, ADR, and presents the theory that underlies the three used methods: HiMod, POD and DD.
- Chapter 3: presents the structure of the code written in C++ and Matlab.
- Chapter 4: shows numerical results obtained by the implementation of these methods.
Chapter 2

Theory

In this chapter we present the theoretical concepts, that underpin the algorithm we implement.
We want to solve an advection-diffusion-reaction problem employing two numerical methods, hierarchical model (HiMod) reduction and proper orthogonal decomposition (POD), in different regions of the domain which is either parallelepipedic or cylindrical. This would represent, in the context of haemodynamics, the input vein, the bifurcation and the output(s) veins. The solutions in each part is then linked through the use of a domain decomposition method.

This chapter is divided in four part:

1. The problem that will be solved is presented: Advection-Diffusion-Reaction.
2. The innovative technique, HiMod, is derived for the geometries and configuration at hand.
3. The reduced basis method, used in the central part of the domain, is explained.
4. The Domain Decomposition used to stick together the different solutions is visited.

For the non-interested reader, a short recap is done at the end of the chapter, aiming to give a minimal understanding of what is then implemented in the code.

2.1 Advection-Diffusion-Reaction (ADR) problem

Let us consider a physical domain \( \Omega \) shaped like a parallelepiped, a cylinder or a rectangular bifurcation and decompose its boundary \( \partial \Omega \) in \( \Gamma_{in}, \Gamma_{out} \) and \( \Gamma_{lat} \) such that
\[
\Gamma_{in} \cap \Gamma_{out} = \emptyset, \quad \Gamma_{in} \cap \Gamma_{lat} = \emptyset, \quad \Gamma_{out} \cap \Gamma_{lat} = \emptyset
\]
and
\[
\Gamma_{in} \cup \Gamma_{out} \cup \Gamma_{lat} = \partial \Omega,
\]
similarly to Figure 2.1 and define \( \Gamma_D = \Gamma_{in} \cup \Gamma_{lat} \).

Let \( \mu \in L^\infty(\Omega) \) with \( \mu \geq \mu_0 \geq 0 \) a.e. in \( \Omega \), the diffusivity coefficient, \( b = [b_1 \ b_2 \ b_3]^T \in L^\infty(\Omega; \mathbb{R}^3) \) the convective field and \( \sigma \in L^\infty(\Omega) \) the reaction coefficient. For the well-posedness of an ADR problem we assume \( \nabla \cdot b \in L^\infty(\Omega) \) and \(-\frac{1}{2} \nabla \cdot b + \sigma \geq 0 \) a.e. in \( \Omega \) (see [6] or [5] for more details). Finally, let \( f \in L^2(\Omega) \) be the forcing term, \( u_{in} \) be the
solution at the inflow boundary and \( n \) be the outward unit normal vector. Then the strong formulation of the problem reads

\[
\begin{aligned}
-\mu \Delta u + b \cdot \nabla u + \sigma u &= f \quad \text{in } \Omega \\
u &= u_{in} \quad \text{on } \Gamma_{in} \\
\mu \nabla u \cdot n &= 0 \quad \text{on } \Gamma_{out} \\
u &= 0 \quad \text{on } \Gamma_{lat}.
\end{aligned}
\]  

(2.1)

We assume that the function

\[
u_D = \begin{cases} 
  u_{in} & \text{on } \Gamma_{in} \\
  0 & \text{on } \Gamma_{lat}
\end{cases}
\]

belongs to \( H^{1/2}(\Gamma_D) \), we consider an extension \( \tilde{u}_D \) of \( u_D \) in \( H^1(\Omega) \) and we set \( w = u - \tilde{u}_D \), then, applying integration by parts formula, the weak formulation for \( w \) is

\[
\text{find } w \in V = H^1_{0,\Gamma_D}(\Omega) \text{ such that for all } v \in V
\]

\[
\int_{\Omega} [\mu \nabla w \cdot \nabla v + (b \cdot \nabla w + \sigma w) v] d\Omega = \int_{\Omega} f v d\Omega - \int_{\Omega} [\mu \nabla \tilde{u}_D \cdot \nabla v + (b \cdot \nabla \tilde{u}_D + \sigma \tilde{u}_D)v] d\Omega.
\]  

(2.2)

Now we define the bilinear form \( a \)

\[
a(w, v) = \int_{\Omega} [\mu \nabla w \cdot \nabla v + (b \cdot \nabla w + \sigma w) v] d\Omega
\]

and the linear functional \( F \)

\[
F(v) = \int_{\Omega} f v d\Omega
\]

so that formulation (2.2) can be written as

\[
\text{find } w \in V = H^1_{0,\Gamma_D}(\Omega) \text{ such that for all } v \in V
\]

\[
a(w, v) = F(v) - a(\tilde{u}_D, v).
\]  

(2.3)

Finally, once \( w \) has been computed, the solution \( u \) is obtained as \( u = w + \tilde{u}_D \).
2.2 Hierarchical Model (HiMod) reduction method

The hierarchical model reduction or HiMod is based on the construction of a 1D model for the dominant direction coupled with a modal expansion that describes transverse dynamics. This separation of variables is also exploited in the decomposition of the computational domain in the Cartesian product of a supporting fiber aligned with the dominant dynamic, and a set of transverse fibers parallel to the secondary dynamics. In particular, the problem is discretized with finite element along the supporting fiber and using a modal expansion along transverse fibers.

Let $\Omega \subset \mathbb{R}^3$ be a domain that can be decomposed in the following way

$$\Omega = \bigcup_{x \in \Omega_{1D}} \{x\} \times \gamma_x$$

where

- $\Omega_{1D}$ is the one-dimensional supporting domain, described only by the independent variable $x$;
- $\gamma_x \subset \mathbb{R}^2$ represents the two-dimensional section or transverse fiber, that also depends on $x$.

In this work, the supporting domain is a segment like $\Omega_{1D} = (x_0, x_1)$. In order to map the physical domain $\Omega$ into a reference domain $\hat{\Omega}$, we define the mapping function

$$\psi^{-1}_x : \gamma_x \to \hat{\gamma}$$

between the fiber $\gamma_x$ and a reference fiber $\hat{\gamma}$ (see fig. 2.2). We denote by $z = (x, y)$ and $\hat{z} = (x, \hat{y})$ a generic point in $\Omega$ and the corresponding point in $\hat{\Omega}$ respectively, and we introduce the mapping function

$$\hat{\Psi}^{-1} : \Omega \to \hat{\Omega}$$

defined by

$$\hat{\Psi}^{-1}(x, y) = (x, \psi^{-1}_x(y))$$

with $x \in \Omega_{1D}$ and $y \in \gamma_x$. In this work, thanks to the geometries of the domain, we employ the linear transformation

$$\hat{y} = \psi^{-1}_x(y) = \frac{1}{L(x)}y$$

where $L(x)$ is the measure of the fiber $\gamma_x$. Finally, the Jacobian associated with the map $\hat{\Psi}$ is of the form

$$\hat{\mathbf{J}}(\hat{z}) = \nabla \hat{\Psi} = \begin{bmatrix} 1 \\ \frac{\partial \psi_x}{\partial x} \\ \nabla \hat{y} \psi_x \end{bmatrix} \in \mathbb{R}^{3 \times 3}$$

and we define $\hat{J} = |\det \hat{\mathbf{J}}|$.

In order to discretize the ADR problem, we need to introduce a reduced functional space. As far as it concerns the supporting fiber, $\Omega_{1D}$ is associated to the functional space $V_{1D} \subset H^1(\Omega_{1D})$, which is compatible with the boundary conditions enforced on the inlet section, while for the transverse fibers we define a modal basis $\{\hat{\varphi}_k\}_{k \in \mathbb{N}} \subset H^1(\hat{\gamma})$, orthonormal with respect to the $L^2$ scalar product on $\hat{\gamma}$. Then the functional space for the transverse expansion is defined as $V_{\hat{\gamma}} = \text{span}\{\hat{\varphi}_k\}$. The number of modes $m \in \mathbb{N}$ used to
write the expansion of the functions has to be chosen a priori, and the reduced functional space $V^m$, which is the reduced counterpart of $V$, is obtained combining these two spaces

$$V^m = \left\{ v^m(x, y) = \sum_{k=1}^{m} v_k(x) \hat{\phi}_k(\psi^{-1}_x(y)), \quad v_k \in V_{1D}, \ x \in \Omega_{1D}, \ y \in \gamma_x \right\}. \quad (2.4)$$

Now let $T_h$ be a partition of the supporting domain $\Omega_{1D}$ in sub-intervals $e_i = (x_{i-1}, x_i)$ of length $h_i = x_i - x_{i-1}$ with $h = \max h_i$. The discrete counterpart of the functional space $V_{1D}$ is a finite element space $V^h_{1D} \subset V_{1D}$ associated to the partition $T_h$

$$V^h_{1D} = X^r_h = \left\{ v_h \in C^0(\Omega_{1D}) : v_h|_{e_i} \in \mathbb{P}^r(e_i) \ \forall \ e_i \in T_h \right\} \quad (2.5)$$

and define $N_h = \dim V^h_{1D}$. We call $\zeta_l$ a generic basis function of the discrete space $V^h_{1D}$, then the final discrete reduced functional space is

$$V^m_{N_h} = \left\{ v^m_h(x, y) = \sum_{k=1}^{m} \sum_{l=1}^{N_h} u_{k,l} \zeta_l(x) \hat{\phi}_k(\psi^{-1}_x(y)), \quad x \in \Omega_{1D}, \ y \in \gamma_x \right\}. \quad (2.6)$$

The discrete representation of the unknown is

$$u^m_h(x, y) = \sum_{k=1}^{m} \sum_{l=1}^{N_h} u_{k,l} \zeta_l(x) \hat{\phi}_k(\psi^{-1}_x(y))$$

and the test function is

$$v^m_h(x, y) = \zeta_p(x) \hat{\phi}_j(\psi^{-1}_x(y))$$

with $p = 1, \ldots, N_h$ and $j = 1, \ldots, m$.

Now we follow [3] in order to write the final discretization of the problem for both parallelepiped and cylindrical domains and we obtain a linear system characterized by an $mN_h \times mN_h$ block matrix where each $N_h \times N_h$ block preserves the typical sparsity pattern of the finite element approximation.
Figure 2.3: Sketch of the linear system corresponding to the discrete HiMod reduced formulation (image from [3]).

2.2.1 HiMod discretization for parallelepipeds

For a parallelepiped domain, it is convenient to use Cartesian coordinates, such that the map can be written as \( \hat{\Psi}^{-1} = \left[ \hat{\Psi}_x^{-1} \hat{\Psi}_y^{-1} \hat{\Psi}_z^{-1} \right]^T \) and a good choice for the reference fiber is the square \( \hat{\gamma} = (-1, +1) \times (-1, +1) \). Moreover the gradient of the map is

\[
\hat{F}^{-T} = \begin{bmatrix}
1 & \hat{D}_y & \hat{D}_z \\
0 & \hat{J}_y & \hat{D}_{yz} \\
0 & \hat{D}_{zy} & \hat{J}_z
\end{bmatrix}
\]

where \( \hat{D}_y = \frac{\partial \hat{\Psi}^{-1}}{\partial x} \circ \hat{\Psi}, \hat{D}_z = \frac{\partial \hat{\Psi}^{-1}}{\partial x} \circ \hat{\Psi}, \hat{J}_y = \frac{\partial \hat{\Psi}_y^{-1}}{\partial y} \circ \hat{\Psi}, \hat{D}_{yz} = \frac{\partial \hat{\Psi}_z^{-1}}{\partial y} \circ \hat{\Psi}, \hat{D}_{zy} = \frac{\partial \hat{\Psi}_y^{-1}}{\partial z} \circ \hat{\Psi} \) and \( \hat{J}_z = \frac{\partial \hat{\Psi}_z^{-1}}{\partial z} \circ \hat{\Psi} \). Then the problem reads

find \( u_{k,l} \) with \( k = 1, \ldots, m, l = 1, \ldots, N_h \) such that for all \( p = 1, \ldots, N_h, j = 1, \ldots, m \)

\[
\sum_{k=1}^{m} \sum_{l=1}^{N_h} u_{k,l} \int_{\Omega_{1D}} \left\{ r_{kj}^{11}(\hat{x}) \zeta_l'(\hat{x}) \zeta_p'(\hat{x}) + r_{kj}^{10}(\hat{x}) \zeta_l(\hat{x}) \zeta_p(\hat{x}) + r_{kj}^{01}(\hat{x}) \zeta_l(\hat{x}) \zeta_p(\hat{x}) \right\} d\hat{x} = \int_{\gamma} \int_{\Omega_{1D}} \int_{\gamma} f(\hat{x}, \hat{y}, \hat{z}) \zeta_p(\hat{x}) \zeta_l(\hat{y}, \hat{z}) d\hat{x} d\hat{y} d\hat{z} \tag{2.7}
\]
where \( d\hat{\gamma} = d\hat{y}d\hat{z}, \hat{J} = |\det \hat{F}| \) and

\[
\begin{align*}
\psi^{10}_{k}(\hat{x}) &= \int_{\gamma} \mu \hat{\phi}(\hat{y}, \hat{z}) \hat{\phi}_{j}(\hat{y}, \hat{z}) d\hat{\gamma} \\
\psi^{0}_{k}(\hat{x}) &= \int_{\gamma} \left[ \hat{D}_{y} \hat{\phi}(\hat{y}, \hat{z}) \frac{\partial \hat{\phi}_{j}}{\partial \hat{y}}(\hat{y}, \hat{z}) + \hat{D}_{z} \hat{\phi}(\hat{y}, \hat{z}) \frac{\partial \hat{\phi}_{j}}{\partial \hat{z}}(\hat{y}, \hat{z}) \right] d\hat{\gamma} \\
\phi^{0}_{k}(\hat{x}) &= \int_{\gamma} \left[ \hat{D}_{y} \hat{\phi}(\hat{y}, \hat{z}) \frac{\partial \hat{\phi}_{j}}{\partial \hat{y}}(\hat{y}, \hat{z}) + \hat{D}_{z} \hat{\phi}(\hat{y}, \hat{z}) \frac{\partial \hat{\phi}_{j}}{\partial \hat{z}}(\hat{y}, \hat{z}) + \hat{D}_{z} \hat{D}_{y} \frac{\partial \hat{\phi}_{j}}{\partial \hat{y}}(\hat{y}, \hat{z}) \right) d\hat{\gamma}.
\end{align*}
\]

2.2.2 HiMod discretization for cylinders

For a cylindrical domain, it is convenient to use cylindrical coordinates, such that the map \( \Psi^{-1} = \left[ \hat{\psi}_{x}^{-1} \quad \hat{\psi}_{\theta}^{-1} \quad \hat{\psi}_{r}^{-1} \right]^T \) and a good choice for the reference fiber is the unit circle \( \hat{\gamma} = [0, 1] \times [0, 2\pi) \). Moreover the gradient of the map is

\[
\hat{F}^{-T} = \begin{bmatrix}
1 & \hat{D}_{r} & \hat{D}_{\theta} \\
0 & \hat{J}_{r} & \hat{J}_{\theta} \\
0 & \hat{D}_{r\theta} & \hat{J}_{\theta}
\end{bmatrix}
\]

where \( \hat{D}_{r} = \frac{\partial \hat{\psi}_{r}^{-1}}{\partial \hat{y}} \circ \hat{\Psi} \), \( \hat{D}_{\theta} = \frac{\partial \hat{\psi}_{\theta}^{-1}}{\partial \hat{y}} \circ \hat{\Psi} \), \( \hat{J}_{r} = \frac{\partial \hat{\psi}_{x}^{-1}}{\partial \hat{r}} \circ \hat{\Psi} \), \( \hat{D}_{r\theta} = \frac{\partial \hat{\psi}_{r\theta}^{-1}}{\partial \hat{r}} \circ \hat{\Psi} \), and \( \hat{J}_{\theta} = \frac{\partial \hat{\psi}_{x}^{-1}}{\partial \hat{\theta}} \circ \hat{\Psi} \). Then the problem reads

find \( u_{k,l} \) with \( k = 1, \ldots, m \), \( l = 1, \ldots, N_h \) such that for all \( p = 1, \ldots, N_h \), \( j = 1, \ldots, m \)

\[
\sum_{k=1}^{N_h} \sum_{l=1}^{m} u_{k,l} \int_{\Omega_{1D}} \left\{ r^{11}_{k,l}(\hat{x}) \phi_{l}(\hat{x}) \phi_{p}(\hat{x}) + r^{10}_{k,l}(\hat{x}) \phi_{l}(\hat{x}) \phi_{p}(\hat{x}) + r^{01}_{k,l}(\hat{x}) \phi_{l}(\hat{x}) \phi_{p}(\hat{x}) \right\} d\hat{x} = \int_{\gamma} R(\hat{x}) \hat{f}(\hat{x}, \hat{r}, \hat{\theta}) \hat{\phi}_{l}(\hat{x}) d\hat{\gamma} \]  

(2.8)
where $R = R(\hat{x})$ is the physical radius of the domain, $d\hat{\gamma} = \hat{r}d\hat{r}d\hat{\theta}$, $\hat{J} = |\det \hat{R}|$ and

$$
\begin{align*}
r_{k_j}^{11}(\hat{x}) &= \int_{\gamma} \mu R(\hat{x}) \hat{r} \hat{\varphi}_k(\hat{r}, \hat{\theta}) \hat{\varphi}_j(\hat{r}, \hat{\theta}) \hat{J} d\hat{\gamma} \\
r_{k_j}^{10}(\hat{x}) &= \int_{\gamma} R(\hat{x}) \left[ \mu \hat{D}_r \hat{\varphi}_k(\hat{r}, \hat{\theta}) \hat{\varphi}_j(\hat{r}, \hat{\theta}) + \mu \hat{D}_r \hat{\varphi}_k(\hat{r}, \hat{\theta}) \frac{\partial \hat{\varphi}_j}{\partial \hat{\theta}}(\hat{r}, \hat{\theta}) \\
&\quad + \mu \hat{D}_\theta \hat{\varphi}_k(\hat{r}, \hat{\theta}) \frac{\partial \hat{\varphi}_j}{\partial \hat{\theta}}(\hat{r}, \hat{\theta}) + b_1 \hat{r} \hat{\varphi}_k(\hat{r}, \hat{\theta}) \hat{\varphi}_j(\hat{r}, \hat{\theta}) \right] \hat{J} d\hat{\gamma} \\
r_{k_j}^{01}(\hat{x}) &= \int_{\gamma} \mu R(\hat{x}) \left[ \hat{D}_r \hat{\varphi}_k(\hat{r}, \hat{\theta}) \hat{\varphi}_j(\hat{r}, \hat{\theta}) + \hat{D}_\theta \frac{\partial \hat{\varphi}_k}{\partial \hat{\theta}}(\hat{r}, \hat{\theta}) \hat{\varphi}_j(\hat{r}, \hat{\theta}) \right] \hat{J} d\hat{\gamma} \\
r_{k_j}^{00}(\hat{x}) &= \int_{\gamma} R(\hat{x}) \left[ \mu (\hat{D}_r^2 + \hat{D}_{\theta r}^2 + \hat{J}_r^2) \frac{\partial \hat{\varphi}_k}{\partial \hat{r}}(\hat{r}, \hat{\theta}) \hat{\varphi}_j(\hat{r}, \hat{\theta}) \\
&\quad + \mu (\hat{D}_r \hat{D}_\theta + \hat{J}_r(\hat{D}_{\theta r} + \hat{D}_{\theta \theta}) \frac{\partial \hat{\varphi}_j}{\partial \hat{\theta}}(\hat{r}, \hat{\theta}) \\
&\quad + \mu \left( \hat{D}_r \hat{D}_\theta + \hat{J}_r \hat{D}_{\theta r} + \hat{D}_{\theta r} \hat{J}_\theta \right) \frac{\partial \hat{\varphi}_k}{\partial \hat{\theta}}(\hat{r}, \hat{\theta}) \hat{\varphi}_j(\hat{r}, \hat{\theta}) \\
&\quad + \mu \left( \hat{D}_\theta \hat{D}_\theta + \hat{J}_\theta \hat{D}_{\theta \theta} + \hat{D}_{\theta \theta} \hat{J}_\theta \right) \frac{\partial \hat{\varphi}_k}{\partial \hat{\theta}}(\hat{r}, \hat{\theta}) \hat{\varphi}_j(\hat{r}, \hat{\theta}) \\
&\quad + \sigma \hat{r} \hat{\varphi}_k(\hat{r}, \hat{\theta}) \hat{\varphi}_j(\hat{r}, \hat{\theta}) \right] \hat{J} d\hat{\gamma}.
\end{align*}
$$

2.3 Proper Orthogonal Decomposition (POD) method

Proper Orthogonal Decomposition is another way to tackle the problem of decreasing the computational cost. We focus on steady parameterized problems, where the parameters concern the problem data, like shape of the domain, coefficients of the equation, boundary conditions, and we solve them several times assigning different values to the parameters in the so-called Offline phase. Then, in the Online phase, we solve the actual problem, applying a POD reduction technique. POD method has different denominations, depending on the context of interest it is also called Principal Component Analysis (PCA), Singular Value Decomposition (SVD), Karhunen-Loève transform (KLT).

Let $\Omega \subset \mathbb{R}^3$ be the physical domain and $V$ be a suitable Hilbert space. A parameterized PDE is a partial differential equation that depends on a vector of parameters $\mu \in P \subset \mathbb{R}^p$ and its weak formulation reads

$$
given \mu \in P, \text{ find } u(\mu) \in V \text{ such that for all } v \in V \quad a(u(\mu), v; \mu) = F(v; \mu). \tag{2.9}\)
$$

We assume that all the hypotheses necessary to guarantee the well-posedness are satisfied, so problem (2.9) admits a unique solution. To solve (2.9) we introduce the discrete space
$V_h \subset V$ with finite dimension $N_h$ and basis functions $\{\varphi_i\}_{i=1}^{N_h}$. The unknown $u_h$ of the discrete weak formulation can be written as

$$u_h(\mu) = \sum_{i=1}^{N_h} u_i(\mu) \varphi_i$$

and the test functions are

$$v_h = \varphi_j$$

for all $j = 1, \ldots, N_h$. If we want to compute the solution of the full-order problem, we have to solve the linear system

$$A(\mu) u(\mu) = f(\mu) \quad (2.10)$$

where

- $A(\mu) \in \mathbb{R}^{N_h \times N_h}$ with $(A(\mu))_{ij} = a(\varphi_j, \varphi_i; \mu)$ is the parametric matrix of the system;
- $f(\mu) \in \mathbb{R}^{N_h}$ with $(f(\mu))_j = F(\varphi_j; \mu)$ is the parametric force term;
- $u(\mu) \in \mathbb{R}^{N_h}$ with $(u(\mu))_i = u_i(\mu)$ is the vector with the coefficients of the parametric solution.

The idea consists on avoiding finding the solution of this linear system, which can be computationally very expensive, and looking for a solution that can be expressed using a smaller number of basis functions. We want to construct a basis $\{\varphi_i\}_{i=1}^{N}$ generating the subspace $V_N \subset V_h \subset V$, which contains the meaningful features of the physical phenomenon. Therefore, we can obtain a solution with a reduced computational cost, which maintains its accuracy. Indeed, once the reduced relevant basis is available, we have to solve a linear system of dimension $N$ in order to find the solution $u_N(\mu) \in V_N$.

Expressing the unknown as

$$u_N(\mu) = \sum_{i=1}^{N} \tilde{u}_i(\mu) \varphi_i$$

and taking as test functions

$$v_n = \varphi_j$$

for all $j = 1, \ldots, N$, the linear system is

$$\tilde{A}(\mu) \tilde{u}(\mu) = \tilde{f}(\mu) \quad (2.11)$$

where

- $\tilde{A}(\mu) = \Phi^T A(\mu) \Phi \in \mathbb{R}^{N \times N}$ is the reduced parametric system matrix;
- $\tilde{f}(\mu) = \Phi^T f(\mu) \in \mathbb{R}^{N}$ is the reduced parametric force term;
- $\tilde{u}(\mu) \in \mathbb{R}^{N}$ with $(\tilde{u}(\mu))_i = \tilde{u}_i(\mu)$ is the vector with the coefficients of the reduced parametric solution;
- $\Phi = [\varphi_1 \ \varphi_2 \ \ldots \ \varphi_N] \in \mathbb{R}^{N_h \times N}$ is the projection matrix, whose columns are the elements of the reduced basis.

Reduced basis methods are characterized by two distinct phases:
• **Offline phase:** it consists of the resolution of \( N_s \) full problems (2.10) for different values of the parameters in order to obtain a large number of high-fidelity solutions. Since we are solving \( N_s \) problems with an accurate method, this phase can be very expensive from a computational point of view. In the end we employ a technique, which in this case is the proper orthogonal decomposition, in order to compute the reduced basis.

• **Online phase:** it consists of the resolution of the reduced problem (2.11) for new values of the parameters. In contrast with the previous one, this phase is very cheap.

### 2.3.1 Computation of the reduced basis

Let \( S = \{\mu_1, \mu_2, \ldots, \mu_{N_s}\} \) be the parameter sample set, then we solve the full problem (2.10) \( N_s \) times accurately, in order to have \( N_s \) high-fidelity solutions, called snapshots. We define the response matrix

\[
U = [u_1 \ u_2 \ \ldots \ u_{N_s}] \in \mathbb{R}^{N_h \times N_s}
\]

where \( u_i \) is the vector with the coefficients of the solution \( u_h(\mu_i) \) for all \( i = 1, \ldots, N_s \). Now we assemble a new matrix \( C \) that contains information about the deviation of \( U \) with respect to the mean value

\[
C = U - \frac{1}{N_s} \sum_{i=1}^{N_s} \begin{bmatrix}
  u_1(\mu_i) & u_1(\mu_i) & \cdots & u_1(\mu_i) \\
  u_2(\mu_i) & u_2(\mu_i) & \cdots & u_2(\mu_i) \\
  \vdots & \vdots & \ddots & \vdots \\
  u_{N_h}(\mu_i) & u_{N_h}(\mu_i) & \cdots & u_{N_h}(\mu_i)
\end{bmatrix} \in \mathbb{R}^{N_h \times N_s}
\]

and we apply the Singular Value Decomposition to \( C \)

\[
C = \Phi \Sigma \Psi^T
\]

where

- \( \Phi \in \mathbb{R}^{N_h \times N_h} \) is the matrix whose columns \( \phi_i \) for \( i = 1, \ldots, N_h \) are the left singular vectors;
- \( \Sigma \in \mathbb{R}^{N_h \times N_s} \) is the matrix whose elements in the diagonal are the singular values \( \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(N_h, N_s)} \geq 0 \);
- \( \Psi \in \mathbb{R}^{N_s \times N_s} \) is the matrix whose columns \( \psi_i \) for \( i = 1, \ldots, N_s \) are the right singular vectors.

Finally, the POD reduced basis is represented by the first \( N \) left singular vectors \( \{\phi_i\}_{i=1}^{N} \).

This section was inspired by the thesis of C. Ferrando [1] and C. Cova [2].

### 2.4 Domain Decomposition (DD) method

Domain decomposition method can be used for any kind of discretization method for partial differential equations in order to:

- make the computation of the solution more efficient on parallel computer platforms;
• solve heterogeneous or multiphysics problems, which are governed by different equations in different regions of the domain;
• use different techniques in different regions of the domain.

The basic idea consists in subdividing the computational domain in a certain number of sub-domains on which smaller problems are solved and this can be done in two ways:

• disjoint sub-domains;
• overlapping sub-domains.

In this work we employ DD with the aim of applying different numerical techniques, HiMod and POD, in different regions of the domain and we use disjoint sub-domains, in particular we apply the Dirichlet-Neumann method. Let Ω be the physical domain, which can be parallelepiped or a cylinder, then we divide it in three parts Ω₁, Ω₂ and Ω₃ such that

\[ \overline{\Omega}_1 \cup \overline{\Omega}_2 \cup \overline{\Omega}_3 = \overline{\Omega}, \quad \overline{\Omega}_1 \cap \overline{\Omega}_2 = \Gamma_1, \quad \overline{\Omega}_2 \cap \overline{\Omega}_3 = \Gamma_2, \quad \Omega_1 \cap \Omega_2 = \emptyset, \quad \Omega_2 \cap \Omega_3 = \emptyset \]

and we apply the Dirichlet-Neumann method before to Ω₁ and Ω₂ and after to Ω₂ and Ω₃ separately. Note that this is a simplified version in which we iterate successively on each part of the domain, without waiting for convergence in the different sub-domains since it was faster and accurate.

2.4.1 Dirichlet-Neumann method

Let $L$ be a linear operator and consider the following problem

\[
\begin{align*}
Lu &= f \quad \text{in } \Omega \\
u &= u_{in} \quad \text{on } \Gamma_{in} \\
\nabla u \cdot n &= 0 \quad \text{on } \Gamma_{out} \\
u &= 0 \quad \text{on } \Gamma_{lat}.
\end{align*}
\]

(2.12)

We divide the domain Ω in Ω₁ and Ω₂ and we call Γ₀ the interface

\[ \overline{\Omega}_1 \cup \overline{\Omega}_2 = \overline{\Omega}, \quad \overline{\Omega}_1 \cap \overline{\Omega}_2 = \Gamma_0, \quad \Omega_1 \cap \Omega_2 = \emptyset. \]

We denote by \( \Gamma_{lat,i} = \Gamma_{lat} \cap \partial \Omega_i \) the intersection between \( \Gamma_{lat} \) and the boundary of the sub-domain \( \Omega_i \) and by \( n \) the outward unit normal vector to \( \Omega_i \) with \( i = 1, 2 \) and we use the convention \( n = n_1 = -n_2 \) in the interface \( \Gamma_0 \). Then we can set up the following iterative algorithm where \( u_1 \) and \( u_2 \) represent the solution respectively in \( \Omega_1 \) and \( \Omega_2 \)

given \( \nabla u_2^{(0)} \cdot n \) on \( \Gamma_0 \), for \( k \geq 1 \) before solve the problem

\[
\begin{align*}
Lu^{(k)}_1 &= f \quad \text{in } \Omega_1 \\
u^{(k)}_1 &= u_{in} \quad \text{on } \Gamma_{in} \\
\nabla u^{(k)}_1 \cdot n &= \nabla u^{(k-1)}_2 \cdot n \quad \text{on } \Gamma_0 \\
u^{(k)}_1 &= 0 \quad \text{on } \Gamma_{lat,1}
\end{align*}
\]

\[ L \]
and after solve the problem

\[
\begin{align*}
Lu_2^{(k)} &= f & \text{in } \Omega_2 \\
\psi_2^{(k)} &= \psi_1^{(k)} & \text{on } \Gamma_0 \\
\nabla \psi_2^{(k)} \cdot n &= 0 & \text{on } \Gamma_{out} \\
\psi_2^{(k)} &= 0 & \text{on } \Gamma_{lat,2}.
\end{align*}
\]

This algorithm is consistent, so when the sequences \( \{ \psi_1^{(k)} \} \) and \( \{ \psi_2^{(k)} \} \) converge, then their limit coincide with the solution of the initial problem \((2.12)\).

### 2.5 Recap

This section aims at giving a very quick (although incomplete) recap of what has been said in this Theory chapter, such that a reader that wishes to skip the previous part would still have an idea of what are the variable of interest for the implementation.

For the HiMod part, we will need the number and kind of elements to discretize the problem in the x-direction and a number of modes to represent the solution in the transverse direction.

For the POD, we need a dimension for the reduced space (alternatively/complementary a tolerance that will determine that dimension), and a number of parameters that we would like to vary.

For the Domain Decomposition, we only need to iterate through the sub-domains, each time passing the data at the boundary to the next sub-domain.
Chapter 3

Code

The initial idea was to provide a Matlab framework that could then be linked with the Matlab library Higamod developed by Yves Barbosa, which explains why the main language is Matlab. However, to solve the HiMod part, LifeV was used since already developed. The use of Matlab also explains the use of a Matlab library, Redbkit, since it was an already developed feature for POD and Finite Elements and their licences were compatible. LifeV is then used as an external library, with some minimal changes. This implementation is thus neither an extension of LifeV nor redbkit.

The work done was to create code to combine all the techniques together (Matlab), expand the branch 20131017_HiMod2 of LifeV (C++) by Sofia Guzzetti such that it includes our problem, modify Redbkit (Matlab) to suit our needs and create test cases. To resume, the implementation mainly made of two languages (Matlab, C++) with two complementary languages (gmsh and FreeFem++) for validation:

- Matlab is the main language, it links the code together, run the different parts and solves the Finite Element with POD.
- C++ code solves the problem with HiMod on the different branches (2 or 3) of the domain.
- FreeFem++ is just for validation purpose, it creates a ground-truth solution.
- gmsh code is just for creating mesh.

Modification file by file is available in Appendix A and an almost complete tree is available in Appendix B. Note the existence of two tutorials for the use of podHimod: part1 [15] and part2 [16] and one video about the installation [14].

3.1 Code structure

The main tree of the repository is as follow:
podHimod/
  Core/
  gmsh/
  TestSuite/
  .gitignore
  INSTALL.md
  README.md
  setPath.m

Core/ contains the core code of the implementation, i.e. every thing that is needed to
make the code work, be it C++ or Matlab. Gmsh/ folder contains the files to generate
needed meshes and the meshes themselves once generated. TestSuite/ folder contains
the implementations of the results shown below in the numerical results section.
All of these folders are investigated more in details in the following sections.

3.1.1 Core/ folder

Core/ structure is as follow:

Core/
  cpp_folder/
    HiMod_LifeV/
      build/
      data/
      include/
      lib/
      src/
      Makefile
      executable files
  matlab_folder/
    solver classes (.m)
    files modified from Redbkit (.m)
    helper functions (.m)

where lib/ contains the compiled libraries i.e. lifeV and parmetis, cpp_folder/ contains
all the cpp code and matlab_folder/ contains the matlab code necessary to run the code,
in particular in contains the solvers classes.
3.1.2 gmsh/ folder

This folder stores the meshes that are done to model the different problems, including the meshes created on the go by the code. The mesh are not version controlled because of their size and the ease with which we can usually produce. It will be mentioned if meshes that need to be stored are needed.

Here is its structure:

```
gmsh/
|-- README.md
|-- build_meshes.sh
|-- rectangles/
|  |-- rectangles.geo files
|  |  |-- structured_rectangle3D.geo
|  |-- cylinders/
|  |  |-- cylinders.geo files
|-- bifurcationT/
|  |-- bifurcation.geo files
```

From the files `.geo` it is easy to construct the corresponding mesh (.msh) file that we need using gmsh.
The only interesting files would be `build_meshes.sh` that will create all the meshes that are necessary for the examples from the `.geo` files and `structured_rectangle3D.geo` which is made in such a way that it can be easily called from command line to create almost any kind of parallelepipeds, and it will be called by Matlab when needed.

3.1.3 TestSuite/ folder

This folder contains three examples shown in the section about numerical results, one for each shape of the domain.

Here is its structure (where we show the structure of the sub-folder only for the test with the rectangle):
Most of the sub-folders in the test folders are created by the class DD_Solver if they are non-existent beforehand.

- **BC/** stores the values at the nodes in .txt files for the boundary conditions to be exchanged. These files will be overwritten continuously.
- **data/** stores the problem data that is fed to C++ code such as dataLeft.pot and dataRight.pot.
- **Figure/** stores vtk solution files that can be later visualized.
- **log/** stores the log files of all the operations done while dealing with C++, Makefiles or C files. They are especially useful when debugging, but still give a lot of information when everything is working smoothly about solver and other details.
- **ReferenceSolution/** stores FreeFem++ computed solution and related files.
- **Snapshots/** stores the data for the POD.
- **sol/** stores the solutions files.

3.2 Installation

cf. file INSTALL.md of the project for more details.
Installation has been tested on Ubuntu 16.04 and 18.04. There also a video tutorial on the installation [13].

**summary**

The instructions are divided in 3 parts:

1. Get the matlab part working
2. Get the C++ part working (hardest part)
3. Get the whole code working

**Step by step**

Short rapid instructions (here tested on ubuntu 16.04)

What you need:

- Ubuntu 16.04 or similar
- podhimod Folder (either from zip or from github)
- glibc toolchains and modules folder

Assuming you have all that here are some instructions:

- Install Matlab with **Partial Differential Equation Toolbox** and **Statistics and Machine Learning Toolbox**
- You should have a compatible version of gcc. For Matlab R2018a this would be version 6.3 but 5.4 works well.

From there:

1. Install gmsh: ‘sudo apt install gmsh’
2. Move toolchains and modules to root and get permissions on contained folder:
   ```
   cd gcc5−glibc/
   sudo mv −i u /
   chmod −Rv u+x /u/sw/
   ```
3. Get redbKIT ([http://redbkit.github.io/redbKIT/](http://redbkit.github.io/redbKIT/)), run setPath.m and make.m (remember to save your path if you don’t want to run setPath.m again). Run the test folder under **problems/**
4. (optional, just a check ) check that the C++ part is working:
   ```
   export mkProfile=/u/sw && source $mkProfile/etc/profile && module load gcc−glibc/5 && module load trilinos && module load muparser && module avail
   you should see all the loaded modules, then:
   cd HiMod_LifeV/ && make
   ```
5. run podhimod/setPath.m
6. check the main.m, datafile.m, datafile_exact.m, training_datafile.m, dataLeft.pot, dataRight.pot. In particular check that the meshes in main.m are all already created. If not, go to folder podhimod/gmsh/ and create them running gmsh, either manually or by running

```
sh build_meshes.sh
```

in gmsh/ folder.

7. It should be installed, run main!

### 3.3 Main classes

The main classes we have implemented are the solvers that perform the three methods explained in the section about theory: domain decomposition, hierarchical model reduction and finite element with proper orthogonal decomposition.

The structure of the solvers is the following

![Solver Class Hierarchy](image)

Where we notice that even DD_Solver.m inherits from SolverAbstract.m, this was made to ensure a given structure even for the englobing class DD_Solver.m

#### 3.3.1 Matlab

SolverAbstract.m is an abstract class which defines the base class of the solver. It contains the main information needed in all the solvers and the basic methods that are implemented in the derived classes. The code is

```matlab
classdef (Abstract) SolverAbstract < handle

    properties (Abstract)
    name; % name of object
    kind; % kind of object
    end

    properties (Access = public)
    fileToReadBdryCond;
```

26
fileToWriteBdryCond;
figFolder; % string naming the repository where the figures
    % will be saved
BCFolder; % string naming the repository where the boundary
    % conditions will be exchanged
coreFolder; % relative path to the core folder
solFolder; % name of the solution folder
dataFolder; % name of the data folder
logFolder; % name of the log folder
end

methods (Abstract)
solve(obj); % solve the corresponding problem
end

methods
function setup(obj) [...] % setup the object
function writeBdryCond(obj) [...] % write boundary conditions
    % in a .txt file
function [...] = readBdryCond(obj,˜,˜) [...] % read boundary
    % conditions from
    % a .txt file
end
end

From SolverAbstract we derive three classes:

- **DD Solver**: it applies the domain decomposition technique, calling the solvers of the different parts and exchanging the boundary conditions, and computes the numerical error with respect to a reference solution.
- **FemPodSolver**: it solves an advection-diffusion-reaction problem applying the finite element method with proper orthogonal decomposition, both offline and online phase.
- **HiModCppSolver**: it handles the C++ code, calling the HiMod solver for the advection-diffusion-reaction problem.

The code of these three classes is the following.

```matlab
classdef DD_Solver < SolverAbstract

properties (SetAccess = public, GetAccess = public)
    name = 'unNamed'; % Name of the solver
    kind = 'DD_solver';
    numSubDomains; % Number of subdomains (2 or 3 or 4)
    DDMaxIter; % Domain decomposition iterations
        % (~5 should be enough)
```

fileWithHimodCoordinates; % string naming the file that stores
  % the HiModCoordinates

%%% SubObject of the class
subProblemObjects; % Stores the subObjects
ObjectLeft; % Left Object
ObjectMiddle; % Middle object
ObjectRight; % Right object
ObjectRightUp; % Up object in the T bifurcation
ObjectRightDown; % Down object in the T bifurcation

%%% Approximated exact solution object
ObjectAlmostExact; % fem object that can be used to compute
  % the exact solution
referenceSolution; % variable where the reference solution is
  % stored after being computed or read

FreeFemFolder; % Name of the FreeFem folder with reference mesh
  % and solution, mass and stiff matrices

% Reference solution computed in FreeFem
refPts;
ffSol;
massMat;
stiffMat;

end

methods (Access = public)

%%% Constructor
function obj = DD_Solver (name) [...] % Setup some values for the subObjects
function [] = setup(obj) [...] % Clean up the temporary files to avoid confusion
function [] = cleanTmpFiles(obj) [...] % Solve the domain decomposition problem using Dirichlet–Neumann method
function times = solve (obj, withPOD) [...] % Compute the reference solution in Matlab
function computeFemReferenceSolution(obj) [...] % Compute the error with respect to the reference solution computed in Matlab
function relErrL2 = compareSolutions(obj, readNan2zero) [...] % Import the reference solution computed in FreeFem
function importReferenceSolution(obj) [...] % Compute the error with respect to the reference solution computed in FreeFem
function [relErrL2, relErrH1] = computeError(obj, readNan2zero)
classdef FemPodSolver < SolverAbstract

properties (Access = public)

name = 'FEM_POD_Solver_Object';
kind = 'fem';
dim = 3; % spatial dimension of the problem
fem = 'P1'; % finite element space
model = 'ADR'; % 'ADR' or 'BifurcationADR'

meshName; % name of the .msh file (or how we want to call it)
meshDir; % mesh directory

readNan2zero; % Boolean, whether we convert Nans to 0 when
% reading boundary conditions
writeNan2zero; % Boolean, whether we convert Nans to 0 when
% writing boundary conditions

datafile; % .m file containing the data of the problem
trainingDatafile; % datafile used for solve_POD()
DATA; % structure with the data of the problem

mesh; % Structure containing the mesh infos
fespace; % finite element space
femSolution; % Solution
L2Error; % L2Error w.r.t. the exact solution (if provided)
H1Error; % H1Error w.r.t. the exact solution (if provided)

xCoordinatesHiMod; % coordinates of right and left object
yCoordinatesHiMod; % coordinates of right and left object
zCoordinatesHiMod; % coordinates of right and left object

inflowBdryCond; % Boundary condition for inflow
% (also parameters for POD)
outflowBdryCond; % Boundary condition for outflow
% (also parameters for POD)
outflowUpBdryCond; % Boundary condition for outflowUp
% (also parameters for POD)
outflowDownBdryCond; % Boundary condition for outflowDown
% (also parameters for POD)
% For POD
snapshotRepositoryName; % string containing the directory where
% the snapshots will be stored for
% training POD
trainingDim; % training dimension for POD
minValueInflow; % Minimum value for inflow parameters
maxValueInflow; % Maximum value for inflow parameters
minValueOutflow; % Minimum value for outflow parameters
maxValueOutflow; % Maximum value for outflow parameters
minValueCoeff; % Minimum value for ADR−coefficient
maxValueCoeff; % Maximum value for ADR−coefficient
interpolation; % set if the parameters of the POD are the
% least−squares polynomials
degreeInterpolation % degree of the least−squares polynomial
% for inflow and outflow boundary condition
tol_POD_U; % Tolerance for POD
% (w.r.t. relative magnitude of eigenvalues)
coefParamSize; % size of the POD−parameter linked to the
% ADR coefficients
V_POD; % matrix with the POD reduced basis
OfflineTrainingSolutions; % name of the file where the
% solutions have to be saved
% in the offline phase

end

methods (Access = public)

% Constructor
function obj = FemPodSolver (name, meshName) [...] % Run some preliminary setup
function setup(obj) [...] % Solve the ADR problem using finite elements and POD
function [time, U, FE_SPACE] = solve_POD(obj, iter, maxIter) [...] % Solve the ADR problem using finite elements
function [time, U, FE_SPACE, errorL2, errorH1] = solve(obj, iter, maxIter) [...] Clear old snapshots computed in the offline phase of POD
function clearSnapshots(obj) [...] % Offline phase of POD
function [] = trainPod(obj, newSnapshots, withPlot) [...] % Compute the values at the boundary and print in a .txt file
function writeBdryCond(obj) [...] % Read boundary conditions from a .txt file
function [] = readBdryCond(obj, fileName, kindOfCondition) [...] % Set boundary conditions from the analytical expression of a function
function [] = setBdryCondFromFunc(obj, func, kindOfCondition)
    [...] 
end
end

classdef HiModCppSolver < SolverAbstract

properties (Access = public)
nname = 'HiModCppSolver';
kind = 'himod';
meshType; % empty for rectangular and 'Cylinder' for cylinder
showLog; % boolean determining whether cpp log output is shown
pathToMatlabMain; % path to matlab executed file
activateModulesString; % string to activate modules on linux
loadModulesString; % string to load modules on linux
cppFolder; % name of the folder where % the cppCode resides
end

methods (Access = public)

% Constructor
function obj = HiModCppSolver (name) [...] 
% Run the main C++ file
function time = solve(obj, iter, maxIter) [...] 
% Compile C++ code
function makeCpp(obj, argument) [...] 
% Run a C++ script to get coordinates of the HiMod grid
function [] = getCoordinates(obj) [...] 
end
end

3.3.2 C++

The C++ code is written to solve an advection-diffusion-reaction problem using HiMod technique and it is an extension of the LifeV library. HiModSolverAbstract is a template abstract class with the main information and the virtual methods implemented in the derived classes. The code is

template <int N>
class HiModSolverAbstract
{
public:

    typedef RegionMesh<LinearLine> mesh_Type;
    typedef MatrixEpetraStructured<Real> matrix_Type;
    typedef VectorEpetraStructured vector_Type;
    typedef PreconditionerIfpack prec_Type;
    typedef boost::shared_ptr<prec_Type> precPtr_Type;

    typedef std::function<Real (const Real&, const Real&, const Real&, const Real&, const ID&)> function_Type;

    typedef boost::shared_ptr<HiModAssembler<mesh_Type, matrix_Type, vector_Type, N>> assembler_ptrType;

    HiModSolverAbstract( const assembler_ptrType& HM,
                         const GetPot& dataFile_ );

    virtual void solveLeft() = 0;
    virtual void solveRight() = 0;

    virtual void writeBC() = 0;
    virtual void writeSolution() = 0;

    protected:

    // Class which collect all the data useful to solve the problem
    boost::shared_ptr<DataHandler> data;

    boost::shared_ptr<Epetra_Comm> Comm;

    boost::shared_ptr<prec_Type> precPtr;
    Teuchos::RCP<Teuchos::ParameterList> belosList;
    LinearSolver linearSolver;

    assembler_ptrType HM;

    DOF DataFESpace;
// Number of degrees of freedom of the 1D finite element space
UInt numdof;

// MapEpetra used
MapEpetra Map;

// Blocks for the linear system
std::vector<UInt> block_row;
std::vector<UInt> block_col;

// Matrix and right hand side vector of the linear system
boost::shared_ptr<MatrixXd> systemMatrix;
boost::shared_ptr<VectorXd> rhs;

// Solution of the linear system
boost::shared_ptr<VectorType> solution;

// Values of the right hand side in the points of the grid
boost::shared_ptr<VectorType> f_interpolated;

// Values in the points of the grid of the function on the inflow boundary for right and the outflow boundary for left
std::vector<std::vector<Real>> evaluate_bc;

};

From HiModSolverAbstract we derive two classes

• HiModSolverRectangular: for domains with a parallelepiped shape.
• HiModSolverCircular: for domains with a cylindrical shape.

They are derived from different specializations of the template abstract class, respectively rectangular (0) and circular (1).
The code of these two classes is the following.

class HiModSolverRectangular : public HiModSolverAbstract<0>
{
public:

// Constructor
HiModSolverRectangular(const assembler_ptrType& HM,
const GetPot& dataFile);

// Function to solve the problem
void solveLeft() override;
void solveRight() override;

// Function to print the boundary conditions in a .txt file
void writeBC() override;

// Function to print the solution in a .txt file
void writeSolution() override;
// Return the solution
boost::shared_ptr<vector_Type> getSolution() const;

private:

// Number of quadrature points
Real nquadY;
Real nquadZ;

// MapEpetra used
MapEpetra Map_3D;

// Solution in the quadrature nodes expanding modal functions
vector_Type solution_3D;

};

class HiModSolverCircular : public HiModSolverAbstract<1> {
public:

// Constructor
HiModSolverCircular( const assembler_ptrType& HM_,
const GetPot& dataFile);

// Function to solve the problem in general
void solveLeft() override;
void solveRight() override;
// Function to print the boundary conditions in a .txt file
void writeBC() override;
// Function to print the solution in a .txt file
void writeSolution() override;
// Return the solution
vector_Type getSolution() const;

private:

// Number of quadrature points
Real nquadRho;
Real nquadTheta;

// MapEpetra used
MapEpetra Map_3D;

// Solution in the quadrature nodes expanding modal functions
vector_Type solution_3D;

};
3.4 Working Flow

This section explains how things work in practice.

For simplicity, we will assume to be working with a simple geometry with just left, middle and right part, be it rectangular or circular. It generalizes to a T bifurcation by replacing right with both up and down.

A simplified general workflow would be as schematized in figure 3.1.

![Workflow Diagram](image)

Figure 3.1: Schema of the workflow to solve a problem

The files we have to specify by hand are (from a project folder):

- main.m, where all the problem is setup, including meshes.
- datafile.m, training_datafile.m, datafile_exact.m (this last only if we want to compute the reference solution with Matlab), where the differential problem is defined: forcing term, governing equation, boundary conditions, which will be the POD parameters...
- data/dataLeft.pot, data/dataRight.pot, where governing equation are defined for the HiMod part, together with element size and kind and number of modes
- ReferenceSolution/refSolFreeFem.edp (only if we want to compute the reference solution with FreeFem)

3.4.1 A look at main.m file

Let us go through a standard main.m file.
First, let us create the general solver, i.e. the DD Solver:

```matlab
MySolver = DD_Solver('DomainDecomposition StrikesBack');
```

We then set some properties of the Solver, such as the number of sub-domains, the number of Domain Decomposition iterations we want to do and relative path to sub-folders we plan to use (there are default values and the solver will create a folder if non-existent and necessary).

```matlab
%% Set properties of the solvers
MySolver.numSubDomains = 3;
MySolver.DDMaxIter = 5;
MySolver.fileWithHimodCoordinates = ' ./data/HiModCoordinatesCartesian.txt';
MySolver.figFolder = 'Figures/';
MySolver.coreFolder = ' ..../Core/';
MySolver.FreeFemFolder = 'ReferenceSolution/';
```

We then setup the three solvers separately with their necessary information. First the left solver:

```matlab
MySolver.displaySectionMessage('Setting properties for left solver');

MySolver.ObjectLeft = HiModCppSolver('Left');
MySolver.ObjectLeft.meshType = '';
pathToMatlabMain];
```

Then the middle solver, where we also set the training dimension for the POD and the bounding values for the parameters of the POD:

```matlab
MySolver.displaySectionMessage('Setting properties for middle solver');

MySolver.ObjectMiddle = FemPodSolver('Middle', 'structured_cube_t1_0625');
MySolver.ObjectMiddle.model = 'ADR';
MySolver.ObjectMiddle.meshDir = ' ..../gmsh/rectangles/';
MySolver.ObjectMiddle.logFolder = MySolver.logFolder;
MySolver.ObjectMiddle.datafile = 'datafile';
MySolver.ObjectMiddle.trainingDatafile = 'training_datafile';

% parameters for POD
MySolver.ObjectMiddle.snapshotRepositoryName = '. / Snapshots';
MySolver.ObjectMiddle.trainingDim = 1000;

MySolver.ObjectMiddle.minValueInflow = -10;
MySolver.ObjectMiddle.maxValueInflow = +10;

MySolver.ObjectMiddle.minValueOutflow = -10;
```
MySolver.ObjectMiddle.maxValueOutflow = +10;
MySolver.ObjectMiddle.minValueCoef = 1;
MySolver.ObjectMiddle.maxValueCoef = 2;

MySolver.ObjectMiddle.interpolation = false; % set if the 
parameters of the POD are the interpolating polynomials
MySolver.ObjectMiddle.degreeInterpolation = 3;
MySolver.ObjectMiddle.tol_POD_U = 1e−2;

And, the right solver:

MySolver.displaySectionMessage('Setting properties for right 
solver');
MySolver.ObjectRight = HiModCppSolver('Right');
MySolver.ObjectRight.meshType = '';
  pathToMatlabMain];

We can then setup the exact solution solver, that will compute a solution 
using uniquely finite elements:

MySolver.displaySectionMessage('Setting properties for exact 
solver');
MySolver.ObjectAlmostExact = FemPodSolver('almostExactSolution',
  'structured_rect_3x1x1');

MySolver.ObjectAlmostExact.meshDir = 'ReferenceSolution/';
MySolver.ObjectAlmostExact.datafile = 'datafile_exact';
MySolver.ObjectAlmostExact.logFolder = MySolver.logFolder;
MySolver.ObjectAlmostExact.model = 'ADR';
MySolver.ObjectAlmostExact.setup();

MySolver.ObjectAlmostExact.setBdryCondFromFunc(@(y,z) 5*(y.*(1−y
  )∗z.*(1−z)), 'dirichlet inflow');

We do a finishing setup for all the solvers, like creating the folders, setting 
up the coordinates of the boundaries at which the BC will be exchanged, ect...

MySolver.setup(); % does the setup of the subsolvers as well ( 
  except AlmostExactSolver)

We train the the POD model

MySolver.ObjectMiddle.trainPod(true);

We solve the problem with DD

times = MySolver.solve(true); % true if with POD, false if 
without
We get the reference solution, by either computing the solution or importing an already computed solution (e.g. with FreeFem++)

```matlab
%MySolver.computefemReferenceSolution();
MySolver.importReferenceSolution();
```

And we finally check the results. CompareSolutions() being for the computed solution and computeError() being for the imported solution:

```matlab
%% Compute error
relErrL2 = MySolver.compareSolutions(true);
disp(['Relative L2 error: ', num2str(relErrL2*100), '%']);

[relErrL2, relErrH1] = MySolver.computeError(true); % true if readNan2zero, false otherwise
disp(['Relative L2 error: ', num2str(relErrL2*100), '%']);
disp(['Relative H1 error: ', num2str(relErrH1*100), '%']);
```

### 3.4.2 Going through the DD_Solver.solve() function

The main part of the code is carried out by the function **DD_Solver.solve()**. This function will iterate through the Domain Decomposition loop and solve the three problems, each time updating the Boundary Conditions as simplified in the following schema \ref{fig:ddSolver}.

![Diagram](image.png)

**Figure 3.2: DD_Solver.solve() simplified workflow**
3.5 Reproducing results

Once you have installed the code and everything is working, go to folder TestSuite/ and you should find the following three folders:

- Test_rectangle_1
- Test_cylinder_1
- Test_bifurcation_1

Run the main/main_cylinder/main_bifurcation.m situated in each of these folders and results similar to the ones reported in this report shall be obtained.

Modifications

If we want an estimate of the error we have two choices: compute a reference solution with FreeFem++ and then load it to compare the solutions or compute a reference solution directly in Matlab using the redbKIT library.

Un-comment the line:

```
%MySolver.importReferenceSolution();
```

to compare with an accurate reference solution previously computed with FreeFem++ (Warning: it can be computationally intensive)

By default, a coarse reference solution is computed with matlab and previously computed data is loaded. Comment:

```
load('ReferenceSolution/MySolver_withReferenceSolutions.mat');
```

to avoid loading previously computed results.

By setting the interpolation value to true in the following line, it is possible to test POD with polynomial coefficients as parameters:

```
MySolver.ObjectMiddle.interpolation = false;
```

Then by playing with the available parameters in main.m, you should be able to reconstruct all presented results. **Reminder: the number of elements and modes in Himod have to be modified in data/dataLeft.pot and data/dataRight.pot**

3.6 Licensing & rights

The two main dependencies of our code are: LifeV (LGPL) and Redbkit (BSD 2 simple). The license will be decided soon and updated on github.\[13\].
Chapter 4

Numerical results

In this chapter we discuss the results obtained using the implementation of the code described in the previous chapter. We present three geometries

1. Parallelepiped
2. Cylinder
3. Bifurcation with rectangular section

and for all of them we show the results of one test case.

As far as it concerns the HiMod method, we vary:

- The number of modes for the expansion in the transverse directions
- The number of elements for the discretization of the 1D supporting domain along the dominant direction

while, regarding the POD method, the parameters can take two forms:

- The solution in the nodes of the inflow and outflow boundaries
- The coefficients of the least square polynomial of the solution at the inflow and outflow boundaries.

In order to compute the numerical error, the full 3D finite element solution is assumed as the reference solution. It is computed using the software FreeFem++, which also provides the mass matrix $M$ and the stiffness matrix $A$. Let $u_{num}$ and $u_{ref}$ be respectively the numerical and the reference solution in the points of the grids $T_{num}$ and $T_{ref}$, then we interpolate the solution $u_{num}$ in order to know its values in the points of the grid $T_{ref}$ using the Matlab function griddata, which applies linear interpolation by default, and we get $\tilde{u}_{num}$. Finally, the relative $L^2$ and $H^1$ errors are computed as

\[
\text{relative } L^2 \text{ error} = \frac{\| \tilde{u}_{num} - u_{ref} \|_{L^2(\Omega)}}{\| u_{ref} \|_{L^2(\Omega)}} = \frac{\sqrt{(\tilde{u}_{num} - u_{ref})^T M (\tilde{u}_{num} - u_{ref})}}{\sqrt{u_{ref}^T M u_{ref}}}
\]

\[
\text{relative } H^1 \text{ error} = \frac{\| \tilde{u}_{num} - u_{ref} \|_{H^1(\Omega)}}{\| u_{ref} \|_{H^1(\Omega)}} = \frac{\sqrt{(\tilde{u}_{num} - u_{ref})^T (M + A) (\tilde{u}_{num} - u_{ref})}}{\sqrt{u_{ref}^T (M + A) u_{ref}}}
\]

The images shown later are obtained visualizing the solution through the software ParaView [11].

First of all, the results on the three geometries are shown, then the influence of DD iterations and POD nodes is reviewed.
4.1 Geometries

4.1.1 Parallelepiped

We consider the advection diffusion reaction problem (2.1) with coefficients

- diffusivity $\mu = 1$
- convective field $\mathbf{b} = [5 \ 0 \ 0]^T$
- reaction $\sigma = -0.2$

where the domain $\Omega$ is a parallelepiped of dimensions 3x1x1, i.e.

- length along the $x$ direction $L_x = 3$
- length along the $y$ direction $L_y = 1$
- length along the $z$ direction $L_z = 1$

the condition on the inlet boundary is

$$u_{in}(y, z) = 5yz(1 - y)(1 - z)$$

and the force term is

$$f(x, y, z) = 10\mathbf{1}_{(0, 4, 1, 8)}(x)\mathbf{1}_{(0, 8, 1)}(y) + 10\mathbf{1}_{(0, 4, 1, 8)}(x)\mathbf{1}_{(0, 0, 2)}(y).$$

Figure 4.1 shows a section of the reference solution.

In Figures (4.2, 4.3) we plot a section of the solution using $m = 40$ modes for the expansion in the transverse directions and $h = 0.05$ for the size of the elements of the discretization of the 1D supporting domain along the dominant direction in the HiMod method. Regarding the POD method, we still use $h = 0.05$ for the size of the elements in the 3D grid and we compare different choices for the parameters. In fig. 4.2 we use the solution in the nodes of the inflow and outflow boundaries as parameters and in fig. 4.3 we use the coefficients of the third order least square polynomial of the solution at the inflow and outflow boundaries.

While the latter does not provide a good result, probably because the solution at the interface cannot be well approximated by a third order polynomial, the former looks very similar to the reference solution and a relative $L^2$ error of 5% has been reached.

![Section of the reference solution for the test case with parallelepiped domain.](image-url)
Figure 4.2: Section of the solution for the test case with parallelepiped domain where the parameters of POD are the solutions in the nodes of the inflow and the outflow boundaries,

Figure 4.3: Section of the solution for the test case with parallelepiped domain where the coefficients of the third order least square polynomial of the solution at the inflow and outflow boundaries.
4.1.2 Cylinder

We consider the advection diffusion reaction problem (2.1) with coefficients

- diffusivity \( \mu = 1 \)
- convective field \( b = [5 \ 0 \ 0]^T \)
- reaction \( \sigma = -0.2 \)

where the domain \( \Omega \) is a cylinder with the following dimensions

- length along the \( x \) direction \( L_x = 3 \)
- radius \( R = 0.5 \)

the condition on the inlet boundary is

\[
u_{in}(r, \theta) = 0.5(0.5^2 - r^2)\]

and the force term is

\[
f(x, r, \theta) = 10 \chi_{(0.4,1.8)}(x) \chi_{(0.3,0.5)}(r).\]

Figure 4.4 shows a section of the reference solution.

![Figure 4.4: Section of the reference solution for the test case with cylindrical domain.](image)

In Figures 4.5 and 4.6 we plot a section of the solution using \( m = 40 \) modes for the expansion in the transverse directions and \( h = 0.05 \) for the size of the elements of the discretization of the 1D supporting domain along the dominant direction in the HiMod method. Regarding the POD method, we still use \( h = 0.05 \) for the size of the elements in the 3D grid and we compare different choices for the parameters. In fig. 4.5 we use the solution in the nodes of the inflow and outflow boundaries as parameters and in fig. 4.6 we use the coefficients of the third order least square polynomial of the solution at the inflow and outflow boundaries. While the latter does not provide a good result, probably because the solution at the interface cannot be well approximated by a third order polynomial, the former looks very similar to the reference solution and a relative \( L^2 \) error of 4 % has been reached.
4.1.3 Bifurcation with rectangular section

We consider the advection diffusion reaction problem (2.1) with coefficients

- diffusivity $\mu = 1$
- convective field $b = \begin{bmatrix} 5\mathbb{1}_{(0,1)}(y) & 5\mathbb{1}_{(1,2)}(y) - 5\mathbb{1}_{(-1,0)}(y) & 0 \end{bmatrix}^T$
- reaction $\sigma = -0.2$

where the domain $\Omega$ is a bifurcation with rectangular section obtained by the union of four cubes with dimension

- length of the side $L = 1$

the condition on the inlet boundary is

$$u_{in}(y, z) = 5yz(1-y)(1-z)$$
and the force term is

\[ f(x, y, z) = 10\mathbf{1}_{(0.4,1.8)}(x)\mathbf{1}_{(0.8,1)}(y) + 10\mathbf{1}_{(0.4,1.8)}(x)\mathbf{1}_{(0,0.2)}. \]

Figure 4.7 shows a section of the reference solution:

![Figure 4.7: Section of the reference solution for the test case with bifurcation domain.](image)

In Figures 4.8 and 4.9 we plot a section of the solution using \( m = 40 \) modes for the expansion in the transverse directions and \( h = 0.05 \) for the size of the elements of the discretization of the 1D supporting domain along the dominant direction in the HiMod method. Regarding the POD method, we still use \( h = 0.05 \) for the size of the elements in the 3D grid and we compare different choices for the parameters. In fig. 4.8 we use the solution in the nodes of the inflow and outflow boundaries and in fig. 4.9 we use the coefficients of the third order least square polynomial of the solution at the inflow and outflow boundaries. The former looks very similar to the reference solution and the relative \( L^2 \) error is 6%.
Figure 4.8: Section of the solution for the test case with cylindrical domain where the parameters of POD are the solutions in the nodes of the inflow and the outflow boundaries.
Regarding the latter, we can see that the least square approximation works well in the interfaces between the middle block and the upper and lower ones where the solution can be well approximated by a third order polynomial, while it has some problems in the interface between the middle block and the left one in fig. 4.9 similarly to the parallelepiped (fig. 4.3) and cylinder (fig. 4.6) cases.

### 4.2 Domain Decomposition iterations

The number of iterations for the Domain Decomposition method has been decided by the necessary number to reach a given convergence. In Figure 4.10 we plot the relative errors ($L^2$ and $H^1$) after each iteration in the case of the parallelepiped. For this test case, we can see that three iterations are sufficient to reach convergence. For all our test cases, four or five iterations were enough to reach convergence.
4.3 POD analysis

Since we saw that POD method, with solution in the nodes of the inflow and the outflow boundaries as parameters, produces good results, now we want to analyze how the error and the computational time change when we vary the number of nodes, and therefore the mesh in the middle part. This analysis has only been done for the parallelepiped in subsection 4.1.1 but is easily generalized to the other geometries. In Figure 4.11 we compare the time both in the offline and online phase to the relative $L^2$ error. We can see that the computational time increases linearly with the number of parameters, while the plot of the relative $L^2$ error as a function of the number of parameters is a hyperbola. Consequently, we can reach a good compromise choosing around $N = 150$ nodes in the inflow and outflow boundaries.
Figure 4.11: Plot of the offline and online computational time [s] for the POD method and the relative error ($L^2$ and $H^1$) of the solution as a function of the number of nodes $N$ in the inflow and outflow boundaries.

Note that the log-log plot is not linear for the $L^2$ error, which is probably due to the constant number of modes and nodes in the Hi-modes parts that does not allow to go under a certain value.
Chapter 5

Conclusion

In this work, the use of HiMod has been extended to domains which contain a bifurcation. The method consists in decomposing the domains between branches and the internal bifurcation and applying HiMod only to the external parts (branches), while employing a reduced basis method (POD), to the internal part. Simple geometries have been considered: a parallelepiped, a cylinder and a bifurcation with rectangular section; and simple equation has been solved: the Advection-Diffusion-Reaction. Starting from these simple cases, we have shown that the idea of combining these two different numerical techniques provides good results and, at the same time, maintains a contained computational cost. Since one of the most important area where HiMod is applied, is haemodynamic, the main future goal is the approximation of the full blood system. Therefore, this work can be extended following two main directions. Firstly, one can solve other problems, in particular the Navier-Stokes equations. Indeed, despite its anatomical complexity, blood can be considered, up to an approximation, as an incompressible Newtonian fluid. Secondly, one can consider bifurcations in more complex domains, which give a more realistic representation of an artery or a vein.

Considerations

Here are a few considerations for next students of PACS or an interested reader. As a general advice, sticking to a unique programming language is easier, not only because you can concentrate on that unique language but also for installation, work and for the ease of use. Moreover, we would advice to use open source software: not only for the benefits it does to society, but also for the ease of reproducibility. When a code take advantages of widely used open source software, a lot of help is available online and many frameworks have been put into place to insure a certain standardization of your work.
Appendix A

Modifications file by file

A.1 C++ code

Include/:

- DataHandler.hpp (new)
  Class which collects all the information to solve an advection-diffusion-reaction problem using HiMod method.

- HiModAssembler2.hpp (modified)
  It includes HiModAssemblerCircular2.hpp and HiModAssemblerRectangular2.hpp, the modified versions of HiModAssemblerCircular.hpp and HiModAssemblerRectangular.hpp.

- HiModAssemblerCircular2.hpp (modified)
  Added functions to handle Dirichlet and Neumann boundary conditions on the inflow and outflow boundary.

- HiModAssemblerRectangular2.hpp (modified)
  Added functions to handle Dirichlet and Neumann boundary conditions on the inflow and outflow boundary. Modified exporter in order to include domains where the origin is not (0,0,0).

- HiModExporterVTK2.hpp (modified)
  Added M_origin variable in order to export solution where the origin with respect to the x direction is different from zero.

- HiModSolverAbstract.hpp (new)
  Abstract class for the solver of the advection-diffusion-reaction problem using HiMod method.

- HiModSolverCircular.hpp (new)
  Class derived from the abstract solver for cylindrical domains.

- HiModSolverRectangular.hpp (new)
  Class derived from the abstract solver for parallelepiped domains.

- ModalSpaceCircular2.hpp (modified)
  Added function to compute the Fourier coefficients when the analytic expression of
the function is unknown, but we only know the values of the function in the points of a grid.

- **ModalSpaceRectangular2.hpp** (modified)
  Added function to compute the Fourier coefficients when the analytic expression of the function is unknown, but we only know the values of the function in the points of a grid.

- **muparser_function_cylindrical.hpp** (modified)
  Class to represent a function which depends on time and cartesian coordinates.

- **muparser_function.hpp** (modified)
  Class to represent a function which depends on time and cylindrical coordinates.

- **QuadratureRule2.hpp** (modified)
  Added a quadrature rule.

- **Timer.hpp** (new)
  Class to measure the computational time.

For **HimodAssembler**, ModalSpace check also the PACS project of Sofia Guzzetti [4] and her Master thesis [3].

**src/**:

- **DataHandler.cpp** (new)
  Class which collects all the information to solve an advection-diffusion-reaction problem using HiMod method.

- **HiModSolverCircular.cpp** (new)
  Class derived from the abstract solver for cylindrical domains.

- **FEDefinitions2.cpp** (modified)
  Added a quadrature rule.

- **HiModSolverRectangular.cpp** (new)
  Class derived from the abstract solver for parallelepiped domains.

- **getCoordinates.cpp** (new)
  Scripts which saves the cartesian coordinates of the grid used by the HiMod method.

- **main.cpp** (new)
  Script which solves the advection-diffusion-reaction problem using the HiMod method for parallelepiped domains.

- **getCoordinatesCylinder.cpp** (new)
  Scripts which saves the cylindrical coordinates of the grid used by the HiMod method.

- **mainCylinder.cpp** (new)
  Script which solves the advection-diffusion-reaction problem using the HiMod method for cylindrical domains.
HiModExporterVtk2.cpp (modified)
Added M_origin variable in order to export solution where the origin with respect to the x direction is different from zero.

ModalSpaceCircular2.cpp (modified)
Added function to compute the Fourier coefficients when the analytic expression of the function is unknown, but we only know the values of the function in the points of a grid.

HiModSolverAbstract.cpp (new)
Abstract class for the solver of the advection diffusion reaction problem using HiMod method.

ModalSpaceRectangular2.cpp (modified)
Added function to compute the Fourier coefficients when the analytic expression of the function is unknown, but we only know the values of the function in the points of a grid.

A.2 Matlab code

ADR.ApplyBC.m (modified)
Changed Dirichlet and Neumann boundaries in inflow, outflow and wall boundaries.

ADR.POD.Solver.m (modified)
Changed the call to the function which includes the boundary conditions.

ADR.Solver.m (modified)
Changed the call to the function which includes the boundary conditions.

BC_info.m (modified)
Changed Dirichlet and Neumann boundaries in inflow, outflow and wall boundaries.

DD.Solver.m (new)
Class derived from the abstract solver which solves the full problem.

FemPodSolver.m (new)
Class derived from the abstract solver which solves an advection-diffusion-reaction problem using the POD method.

HiModCppSolver.m (new)
Class derived from the abstract solver which handles the solution of an advection-diffusion-reaction problem using the HiMod method in C++.

my_FFimportfilemesh_3D.m (copied)
Function taken from the Matlab library Higamod developed by Yves Barbosa which imports the reference solution (.sol) and the relative mesh (.mesh), mass matrix (.txt) and stiffness matrix (.txt).

numberCoefficients.m (new)
Given the degree of a 2D polynomial, this function returns the general number of monomials and coefficients.

polyfit2D.m (new)
Equivalent to the Matlab function polyfit for 2D polynomials.
• polyval2D.m (new)
  Equivalent to the Matlab function polyval for 2D polynomials.

• SolverAbstract.m (new)
  Abstract class for a solver.
Appendix B

Tree Structure

Here the tree structure of the project is shown. All repetitive files or file of no interest for the structure have been removed for readability.

|-- Core
 |   |-- cpp_folder
 |   |   |-- HiMod_LifeV
 |   |   |   |-- Makefile
 |   |   |   |-- build
 |   |   |   |-- data
 |   |   |   |   |-- include
 |   |   |   |   |   |-- DataHandler.hpp
 |   |   |   |   |   |-- HiModAssembler2.hpp
 |   |   |   |   |   |-- HiModAssemblerCircular2.hpp
 |   |   |   |   |   |-- HiModAssemblerRectangular2.hpp
 |   |   |   |   |   |-- HiModExporterVtk2.hpp
 |   |   |   |   |   |-- HiModSolverAbstract.hpp
 |   |   |   |   |   |-- HiModSolverCircular.hpp
 |   |   |   |   |   |-- HiModSolverRectangular.hpp
 |   |   |   |   |   |-- ModalSpaceCircular2.hpp
 |   |   |   |   |   |-- ModalSpaceRectangular2.hpp
 |   |   |   |   |   |-- QuadratureRule2.hpp
 |   |   |   |   |   |-- README.md
 |   |   |   |   |   |-- Timer.hpp
 |   |   |   |   |   |-- muparser_function.hpp
 |   |   |   |   |   '-- muparser_function_cylindrical.hpp
 |   |-- lib
 |   |   |-- libev
 |   |   '-- parmetis
 |   |-- main
 |   |-- mainCylinder
 |   |-- src
 |   |   |-- DataHandler.cpp
 |   |   |-- FEDefinitions2.cpp
 |   |   |-- HiModExporterVtk2.cpp
 |   |   |-- HiModSolverCircular.cpp
 |   |   |-- HiModSolverRectangular.cpp
 |   |   |-- ModalSpaceCircular2.cpp
-- ModalSpaceRectangular2.cpp
-- getCoordinates.cpp
-- getCoordinatesCylinder.cpp
-- main.cpp
  '-- mainCylinder.cpp

'-- activate_modules_linux.sh

matlab_folder
  '-- ADR_ApplyBC.m
  '-- ADR POD Solver.m
  '-- ADR Solver.m
  '-- BC info.m
  '-- DD Solver.m
  '-- FemPodSolver.m
  '-- HiModCppSolver.m
  '-- SolverAbstract.m
  '-- my FF import file mesh 3D.m
  '-- numberCoefficients.m
  '-- polyfit2D.m
    '-- polyval2D.m

-- INSTALL.md
-- README.md
-- TestSuite
  '-- Test bifurcation T_1
    '-- BC
      '-- ReferenceSolution
        '-- 3D bifurcation T.edp
          '-- A.txt
          '-- M.txt
        '-- MySolver with Reference Solution.mat
          '-- ffMesh.mesh
          '-- ffSolution.sol
            '-- structured bifurcation T.msh
        '-- Snapshots
          '-- solutionSnapshots.h5
        '-- data
          '-- HiModCoordinates Cartesian.txt
            '-- dataLeft.pot
            '-- dataRightDown.pot
            '-- dataRightUp.pot
          '-- datafile.m
          '-- datafile_exact.m
        '-- log
          '-- log Left 1.txt
          '-- log RightDown 1.txt
          '-- log RightUp 1.txt
          '-- main bifurcation.m
            '-- training datafile.m
  '-- Test cylinder 1
    '-- BC

58
172 directories, 1062 files
Acknowledgement

Our profound gratitude goes to Prof. Simona Perotto and Yves Barbosa for their patience with us and their gentle help.
A big thanks to the contributors of Gmsh [9], paraView [11], LifeV [12] and RedbKIT [10] for their incredible softwares and great documentation, without which it would have been impossible for us to make any advances.
Bibliography


[13] Url of the project on github (might still be private) https://github.com/LucaZampieri/podHimod

[14] installation tutorial: https://www.youtube.com/watch?v=n5xUSBIKURA&t=0s&index=4&list=PLBSLkDQ8eWkAvrS8C5_MufqeArPw8F8Cu

[15] tutorial use part 1: https://www.youtube.com/watch?v=ybBimVy85-g&t=0s&index=3&list=PLBSLkDQ8eWkAvrS8C5_MufqeArPw8F8Cu

[16] tutorial use part 2: https://www.youtube.com/watch?v=uJ4b-d8DVG0&t=0s&index=2&list=PLBSLkDQ8eWkAvrS8C5_MufqeArPw8F8Cu