3D HI-POD applied to the Advection-Diffusion-Reaction problem

Authors:
Cristina COVA
Lorenzo SALA

Professors of the course:
Luca FORMAGGIA
Carlo DE FALCO

December 3, 2015
Abstract

The aim of this project is the implementation in *LifeV*\(^1\) of a Hi-POD solver for advection-diffusion-reaction problems on 3D parallelepiped and cylindrical domains.

Hi-POD exploits both hierarchical model reduction method (Hi-Mod) and proper orthogonal decomposition (POD) to combine them in a second order reduction approach.

In the first part of this report we will describe the main features of the Hi-POD method, then we will discuss the programming design and the structure of the code. Moreover we will provide numerical experiments to validate the Hi-POD approach with respect to the full Hi-Mod method. Finally, we present a tutorial for understanding how to use the novel method in *LifeV*.

\(^1\)Scalable scientific C++ finite element library developed in MOX, EPFL and Emory [www.lifeV.org].
## Contents

1 The Hi-POD method ........................................ 2
  1.1 Hi-Mod .................................................. 2
    1.1.1 Geometrical setting .................................. 3
    1.1.2 Functional setting and reduced problem .......... 4
    1.1.3 Full discretized reduced problem ................. 6
  1.2 Hi-POD ................................................ 7
    1.2.1 Offline ............................................ 8
    1.2.2 Online ............................................ 9

2 Programming design of Hi-POD in LifeV ................. 11
  2.1 Hi-Pod Assembler class ................................ 12
  2.2 Hi-Pod Solver class .................................... 15
  2.3 Hi-Pod Serial Dense Utilities ........................ 20
  2.4 Converter Utilities .................................... 21

3 Numerical Test and Results .............................. 22
  3.1 Test case 1 - The parallelepiped domain ............ 22
  3.2 Test case 2 - The circular domain .................... 27

4 Tutorial .................................................. 32
Chapter 1

The Hi-POD method

The aim of this chapter is to introduce the main features of the Hi-POD method. This technique is based on two different model reduction methods merged together: Hi-Mod and Proper Orthogonal Decomposition (POD).

We introduce firstly the fundamental notations and definitions of the Hi-Mod method and then its combination with POD generating Hi-POD. From now we refer to the following advection-diffusion-reaction problem:

\[
\begin{aligned}
-\mu(x) \Delta u(x) + b(x) \nabla u(x) + \sigma(x) u(x) &= f(x) \quad \text{in } \Omega \\
u &= g \quad \text{su } \Gamma_D \\
\mu \frac{\partial u}{\partial n} &= h \quad \text{su } \Gamma_N
\end{aligned}
\]  

with \( \Gamma_D \cap \Gamma_N = \emptyset \), \( \Gamma_D \cup \Gamma_N = \partial \Omega \) and \( \Gamma_D \) is the portion of the border with Dirichlet condition, while \( \Gamma_N \) has Neumann boundary condition.

1.1 Hi-Mod

In this section we present the Hierarchical Model (Hi-Mod) Reduction method, which solve partial differential problems with a reduced computational cost and without sacrificing accuracy.

Hi-Mod provides a reliable approximation of a phenomenon characterized by a dominant direction, for instance the blood flow in an artery. Thus, the idea is to neglect those dimensions of the domain where the solution doesn’t show a relevant behaviour and to adopt a finite element discretization along the main direction while a modal expansion along the transversal one. In this way, theoretically, the complexity of the problem decrease and we can treat it as 1D case.
1.1.1 Geometrical setting

For simplicity in the presentation of how acts the method we will refer always to a Lipschitzian domain \( \Omega \in \mathbb{R}^n \), \( n = \{2, 3\} \), such that \( \Omega = \Omega_x \times \Sigma_x \), where \( \Omega_x \) is the support fiber 1D, meanwhile \( \Sigma_x \) represents the transversal section at \( x \).

Generally the transversal section is a function of the variable \( x \) running along the mainstream. The amplitude of the domain may consequently vary along the horizontal direction.

It is useful to introduce a reference domain (see Fig.(1.1)) \( \hat{\Omega} = \Omega_x \times \hat{\Sigma} \), characterized by the same supporting fiber. The partial differential problem to be reduced is actually solved on \( \hat{\Omega} \) where the computations are easier.

We introduce a map \( \gamma_x(\cdot) : \Sigma_x \rightarrow \hat{\Sigma} \) given by

\[
\hat{y} = \gamma(y), \quad \hat{y} \in \hat{\Sigma}, \quad y \in \Sigma_x
\]

that induces a global map \( \Psi : \Omega \rightarrow \hat{\Omega} \). Assume \( \gamma \in C^1(\Sigma_x) \), we can define the Jacobian matrix \( J \) associated to the map \( \Psi \) (Figure 1.2),

\[
J(x) = \begin{bmatrix}
1 & 0 \\
\frac{\partial \gamma_x(y)}{\partial x} & \nabla_y \gamma_x(y)
\end{bmatrix}
\]

In particular we introduce the notation:

\[
\hat{J}(y) = \nabla_y \gamma_x(y)
\]
1.1.2 Functional setting and reduced problem

We consider the general Advection-Diffusion-Reaction (ADR) problem 1.1:

\[
\begin{cases}
-\mu(x)\Delta u(x) + b(x)\nabla u(x) + \sigma(x)u(x) = f(x) & \text{in } \Omega \\
u = g & \text{su } \Gamma_D \\
\frac{\partial u}{\partial n} = h & \text{su } \Gamma_N
\end{cases}
\]

with the following hypothesis:

\[\mu \in L^\infty(\Omega), \ b(x) \in [L^\infty(\Omega)]^n, \ \sigma \in L^\infty(\Omega) \text{ e } \text{div}(b) \in L^\infty(\Omega).\]

To guarantee the well-posedness of the problem we assume that \(-\frac{1}{2}\text{div}(b)+\sigma \geq 0 \ a.e. \ in \ \Omega \) and \(f \in L^2(\Omega).\) Then we introduce the Sobolev spaces

\[H^1(\Omega) = \{v \in L^2(\Omega) : \nabla v \in [L^2(\Omega)]^n\}.\]

and

\[H^1_{0,\Gamma_D}(\Omega) = \{v \in H^1(\Omega) : v|\Gamma_D = 0\}.\]

The weak formulation of the problem reads:

find \(u \in V : a(u, v) = F(v) \ \forall v \in V. \quad (1.2)\)

For completeness, we recall the Lax-Milgram theorem.

**Theorem 1 (Lax-Milgram)** Let \(V\) bet a (real) Hilbert space, \(a(u,v) : V \times V \rightarrow \mathbb{R}\) a continuous and coercive bilinear form and \(F(v) : V \rightarrow \mathbb{R}\) a linear continuous functional.

The reader can find the proof in [9].

In our problem \(V = H^1_{0,\Gamma_D}(\Omega)\) and

\[a(u,v) = \int_{\Omega} [\mu \nabla u \cdot \nabla v + (b \cdot \nabla u)v + \sigma uv]d\omega + \int_{\Gamma_D} \chi uv\]

and the functional

\[F(v) = \int_{\Omega} fvd\omega\]

\(F \in V^*\) where \(V^*\) denotes the dual space of \(V.\)

Since the hypothesis of the theorem (Th. 1) are satisfied, the problem (1.2) has a unique solution.

Whereas Hi-Mod acts in a different way depending on the direction of the domain, it’s necessary to introduce new different functional spaces. Whereas the aim is to decompose it, we associate a 1D Sobolev space \(V^{1D} \subset H^1(\Omega_x)\) with the mainstream and the \((n - 1)\) dimensional space \(H^1(\Sigma)\) with the transverse reference section. Space \(V^{1D}\) is set in order to impose on the support fiber \(\Omega_x\) the boundary conditions that are naturally induced by the boundary conditions on the whole domain. Using the isometric equivalence results we can obtain that
\[ H^1(\Omega) = H^1(\Omega_x) \otimes H^1(\Sigma). \]

We introduce a family of modal orthonormal functions \( \{ \varphi_i \}_{i \in \mathbb{N}} \) with respect to the \( L^2(\Sigma) \) scalar product. These functions form a basis for \( H^1(\Sigma) \). Hence a new representation for \( H^1_{0, \Gamma_D}(\Omega) \) is the following one:

\[
H^1_{0, \Gamma_D}(\Omega) = \{ v(x, y) = \sum_{i=1}^{\infty} v_i(x) \varphi_i(\gamma_x(y)), \text{with } v_i \in V^{1D} \}
\]

and the corresponding reduced space is

\[
V_m = \{ v_m(x, y) = \sum_{i=1}^{m} v_i(x) \varphi_i(\gamma_x(y)) \}, \quad (1.3)
\]

where the index \( m \) identifies the dimension of the enriched reduced space. When \( m \) is large, we have a more accurate solution.

Thanks to the orthogonality of the modal functions \( \varphi_i \) with respect to \( L^2(\Sigma) \) scalar product, the coefficients \( v_i \) in (1.3) are defined as:

\[
v_i(x) = \int_{\Sigma} v_m(x, \hat{y}) \varphi_i(\gamma_{\hat{y}}^{-1}(\hat{y})) d\hat{y}
\]

Introducing \( u_m \in V_m \) as an approximate solution of:

\[
a(u_m, v) = F(v) \quad \forall v \in V_m
\]

it can be proved that \( u_m \) converges to \( u \) for \( m \to \infty \).

Choosing \( v = \varphi_k(y)\zeta(x) \), the Hi-Mod reduced formulation (Eq. 1.4) reads: for \( j = 1, \ldots, m \) find \( u_j \in V^{1D} \) such that, \( \forall \zeta \in V^{1D} : \)

\[
\sum_{j=1}^{m} \int_{\Omega_x} \left[ \tilde{r}_{kj}^{11} \frac{\partial \tilde{u}_j(x)}{\partial x} \frac{\partial \zeta(x)}{\partial x} + \tilde{r}_{kj}^{01} \frac{\partial \zeta(x)}{\partial x} + \tilde{r}_{kj}^{00} u_j(x) \frac{\partial \zeta(x)}{\partial x} \right] dx
\]

\[
= \int_{\Omega_x} \left[ \int_{\Sigma} f(x, \gamma_{\hat{y}}^{-1}(\hat{y})) \varphi_k(\hat{y}) |\text{det}(\tilde{J}^{-1}(x, \hat{y}))| d\hat{y} \right] \zeta(x) dx
\]

with \( k = 0, \ldots, m \) and where coefficients \( \tilde{r}_{kj}^{s,t}(x, \hat{y}) \) are given by

\[
\tilde{r}_{kj}^{s,t}(x) = \int_{\Sigma_x} r_{kj}^{s,t}(x, \hat{y}) |\text{det}(\tilde{J}^{-1}(\hat{y}))| d\hat{y} \quad s, t = 0, 1
\]
r_{kj}^s$ coefficients are defined as:

\begin{align*}
    r_{kj}^{1,1}(x, \hat{y}) &= \mu(x, \gamma_x^{-1}(y)) \varphi_j(\gamma_x^{-1}(y)) \varphi_k(\gamma_x^{-1}(y)) \\
    r_{kj}^{1,0}(x, \hat{y}) &= \mu(x, \gamma_x^{-1}(y)) \varphi_j(\gamma_x^{-1}(y)) \frac{\partial \varphi_k(\gamma_x^{-1}(y))}{\partial x} + b_x(x; \gamma_x^{-1}(\hat{y})) \varphi_j(\gamma_x^{-1}(y)) \\
    r_{kj}^{0,1}(x, \hat{y}) &= \mu(x, \gamma_x^{-1}(y)) \frac{\partial \varphi_j(\gamma_x^{-1}(y))}{\partial x} \varphi_k(\gamma_x^{-1}(y)) + \sigma(x, \gamma_x^{-1}(y)) \varphi_j(\gamma_x^{-1}(y)) \varphi_k(\gamma_x^{-1}(y)) + \\
    &+ b_{y_1} \frac{\partial \varphi_j(\gamma_x^{-1}(y))}{\partial y_1} \varphi_k(\gamma_x^{-1}(y)) + b_{y_2} \frac{\partial \varphi_j(\gamma_x^{-1}(y))}{\partial y_2} \varphi_k(\gamma_x^{-1}(y))
\end{align*}

for $k = 1, \ldots, m$, $y = [y_1, y_2]^T$, $b = [b_x, b_{y_1}, b_{y_2}]^T$.

We observe that the obtained semi-discrete formulation allows us to reduce a multidimensional partial differential problem to a set of one-dimensional coupled problems, which are defined on the supporting fiber $\Omega_x$.

### 1.1.3 Full discretized reduced problem

In the Hi-Mod reduction it is required to discretize the finite elements only along the main direction $\Omega_x$. For this purpose we introduce a partition $\tau_h$ for $\Omega_x$ into $N_h$ subintervals $K_j = (x_{j-1}, x_j)$, $a = x_0 < x_1 < \cdots < x_{N_h-1} < x_{N_h} = b$ with $h_j = x_j - x_{j-1}$, $h = \max_j h_j$.

The 1D finite element space is

\[ X^1_h = \{ v_h \in C^0(\Omega_x) : v_h|_{K_j} \in P_1, \forall j \in \tau_h \} \]

where $P_1$ is the space of polynomials of degree one. The discrete counterpart will be therefore

\[ V^{1D}_h = X^1_h \cap V^{1D}, \quad \text{with dim}(V^{1D}_h) = N_h < \infty \]

with basis $\zeta_j(x)^{N_h}_{j=1} \subset V^{1D}_h$.

Along the transversal direction we adopt a uniform or global approach to determine the number of the modal basis that approximate the solution. This approach is based substantially on the selection of the minimum number $m$ of modes needed to describe the solution with a reliable approximation. The reliable estimate can be achieved a priori through a trial and error approach, whereas a posteriori via a goal-oriented estimation.

Fixing $m$, the discrete counterpart $V^m_h$ of the space $V_m$ can be introduced as:

\[ u^m_h \in V^m_h := \{ v^m_h(x, y) = \sum_{i=1}^{m} v^i_h \cdot \varphi_i(\gamma_x(y)), \text{con } v^i_h \in V^{1D}_h \} \]
Therefore, the final discretize Hi-Mod formulation for problem (1.1) is: for \( j = 1, \ldots, m \) find \( u^j_h \in V_h^{1D} \) such that:

\[
\sum_{j=1}^{m} \int_{\Omega} \left[ \delta_{kj}^{1,1} \frac{\partial u^j_h(x)}{\partial x} \frac{\partial \zeta_i(x)}{\partial x} + \delta_{kj}^{1,0} \frac{\partial u^j_h(x)}{\partial x} \zeta_i(x) + \delta_{kj}^{0,1} u^j_h(x) \frac{\partial \zeta_i(x)}{\partial x} + \delta_{kj}^{0,0} u^j_h(x) \zeta_i(x) \right] \, dx
\]

\[
= \int_{\Omega} \left[ \int_{\Sigma} f(x, \gamma^{-1}(y)) \varphi_k(y) \left| \det(\tilde{J}^{-1}(x, y)) \right| \, dy \right] \zeta_i(x) \, dx
\]

(1.5)

with \( k = 1, \ldots, m \) and \( i = 1, \ldots, N_h \).

Under suitable assumption on (1.5) it can be proved that \( u_m \) converges to \( u \) for \( m \to \infty \) and \( h \to 0 \).

Considering the algebraic formulation related to (1.5), we obtain a linear system whose block matrix has dimension \( m \cdot N_h \times m \cdot N_h \) [Fig. (1.3)].

![Figure 1.3: Example of a matrix pattern of Hi-Mod discretization with \( m = 4 \) and \( N_h = 10 \)](image)

### 1.2 Hi-POD

In this section we introduce the Hi-POD technique, obtained using both Hi-Mod and proper orthogonal decomposition method. In particular, we show Hi-POD applied to the standard advection-reaction-diffusion (ADR) problem (1.1) assuming that the leading dynamics is parallel to x-direction.

Let \( m \) be the number of modal basis functions used along the y and z-direction and \( N_h \) be the number of degrees of freedom associated with the finite element discretization along the x-axis.

Considering the algebraic system associated with a Hi-Mod reduction

\[
Au = f,
\]

(1.6)
where \( A \in \mathbb{R}^{mN_h \times mN_h} \) is the stiffness matrix, \( u \in \mathbb{R}^{mN_h} \) is the solution vector, \( f \in \mathbb{R}^{mN_h} \) is the force-term vector, we reduce, via a ROM (pod approach), the dimension of the linear system (1.6) in order to further increase the computational saving led by Hi-Mod reduction.

The reduced order method consists in two different phases:

- **OfflinePhase**: it’s a pre-processing phase in which we solve some different problems with Hi-Mod and store the solutions. These solutions are used to derive a reduced modal basis.

- **OnlinePhase**: we actually solve the problem of interest, projecting and solving it on the reduced modal basis.

### 1.2.1 Offline

The first important thing to do is to identify the parameters of the problems to be solved during the off-line step to build the response matrix. The solutions of these problems, in fact, will influence the choice of the proper orthogonal basis and consequently the solution computed in the on-line phase.

We recall that the solution of a single ADR problem via a Hi-Mod approach is represented by the vector

\[
\mathbf{u} = \left[ u_1(x_1), \ldots, u_1(x_{N_h}), u_2(x_1), \ldots, u_2(x_{N_h}), \ldots, u_m(x_1), \ldots, u_m(x_{N_h}) \right]^T \in \mathbb{R}^{mN_h},
\]

where \( u_i(x_j) \) denotes the i-th modal coefficient evaluated at the j-th FE node.

Using the Hi-Mod approach we solve a series of ADR problems with different problem data and boundary conditions and then we store them in a response matrix by columns:

\[
\mathbf{U} = \begin{bmatrix}
\mathbf{u}^1 & \mathbf{u}^2 & \ldots & \mathbf{u}^{N_{sol}}
\end{bmatrix} \in \mathbb{R}^{(mN_h) \times N_{sol}} \tag{1.7}
\]

where \( N_{sol} \) denotes the number of problem solved in the off-line phase.

To build the covariance matrix \( \hat{R} \) we firstly compute \( V \) starting from the response matrix \( \mathbf{U} \) and we perform an average:

\[
\mathbf{V} = \mathbf{U} - \frac{1}{N_{sol}} \sum_{i=1}^{N_{sol}} \begin{bmatrix}
\mathbf{u}^i_1(x_1) & \mathbf{u}^i_1(x_1) & \ldots & \mathbf{u}^i_{N_{sol}}(x_{N_1}) \\
\vdots & \vdots & \ddots & \vdots \\
\mathbf{u}^i_m(x_{N_h}) & \mathbf{u}^i_m(x_{N_h}) & \ldots & \mathbf{u}^i_m(x_{N_h})
\end{bmatrix} \in \mathbb{R}^{mN_h \times N_{sol}}
\]

Then we obtain the covariance matrix as follow:

\[
\hat{R} = \frac{1}{N_{sol}} \mathbf{V} \mathbf{V}^T \in \mathbb{R}^{(mN_h) \times (mN_h)}
\]

Applying the singular value decomposition to \( \hat{R} \), we derive the eigenvectors that span the subspace of the reduced solutions.
Indeed the SVD guarantees that there exists a set of real numbers \( \sigma_1, \sigma_2, \ldots, \sigma_k \) with \( \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k \), where \( k \) is the rank of matrix \( \tilde{R} \), so that

\[
\tilde{R} = \Phi \Sigma \Psi^T \tag{1.8}
\]

with \( \Phi \in \mathbb{R}^{mN_h \times mN_h}, \Psi \in \mathbb{R}^{N_{sol} \times N_{sol}}, \Sigma \in \mathbb{R}^{mN_h \times N_{sol}} \).

Columns of \( \Phi \) are called left singular vectors and columns of \( \Psi \) are called right singular vectors.

Matrices \( \Phi \) and \( \Psi \) are squared and unitary, while matrix \( \Sigma \) is pseudo-diagonal so that:

\[
\Sigma_{ii} = \sigma_i, \quad \text{for } i = 1, \ldots, k
\]

identifies the \( i \)-th singular value. We have:

\[
\Sigma = \begin{bmatrix}
D & 0_{k \times (N_{sol} - k)} \\
0_{(mN_h - k) \times k} & 0_{(mN_h - k) \times (N_{sol} - k)}
\end{bmatrix}
\]

with \( D = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_k) \in \mathbb{R}^{k \times k} \).

From (1.8) it can be derived that

\[
\tilde{R} = \Phi^k D (\Psi^k)^T \tag{1.9}
\]

where \( \Phi^k \in \mathbb{R}^{mN_h \times k}, \Psi^k \in \mathbb{R}^{N_{sol} \times k} \) are the matrices defined by:

\[
\Phi^k_{ij} = \Phi_{ij}, \quad \text{for } 1 \leq i \leq mN_h \text{ and } 1 \leq j \leq k, \quad \Psi^k_{ij} = \Psi_{ij}, \quad \text{for } 1 \leq i \leq N_{sol} \text{ and } 1 \leq j \leq k.
\]

If we denote by \( \{\varphi_i^k\}_{i=1}^{mN_h} \) and \( \{\psi_i^k\}_{i=1}^{N_{sol}} \) the column vectors of \( \Phi^k \) and \( \Psi^k \) respectively it can be stated:

\[
V \psi_i = \sigma_i \varphi_i, \quad V^T \varphi_i = \sigma_i \psi_i, \quad \text{for } i = 1, \ldots, k.
\]

Notice that \( \{\varphi_i\}_{i=k+1}^{mN_h} \) and \( \{\psi_i\}_{i=k+1}^{N_{sol}} \) are left and right singular vectors associated to null singular values.

For further details refer to [1] and [2].

### 1.2.2 Online

In the online phase we project the algebraic system \( Au = f \) onto a subspace generated by a few of the eigenvectors provided by the spectral decomposition of the correlation matrix. To detect how many eigenvalues are necessary we use the following criterion:

\[
k = \max_{m \leq n} \text{ such that } \frac{\sum_{i=1}^{m} \lambda_i}{\sum_{i=1}^{n} \lambda_i} \leq TOLL \tag{1.10}
\]

where \( n = mN_h \) and \( TOLL \) is the maximum error we admit for the reduced model.

Precisely, the reduced solution is derived by solving:

\[
\Phi^T A \Phi \tilde{u} = \Phi^T f.
\]
Since the rank of matrix $\Phi$ is not full we can rewrite

$$(\Phi^k)^T A \Phi \tilde{u} = (\Phi^k)^T f.$$ 

Solving this equation, we obtain finally the POD reduced solution $\tilde{u} \in \mathbb{R}^k$.

Thus the final full solution $u \in \mathbb{R}^{mN_h}$ is

$$u = \Phi^k \tilde{u}.$$ 

In the Alg. (1.1) we describe the flow-chart of the reduced order Hi-Pod.

**Offline:**

1. Parameters selection (sampling)
2. Solution of $N_{sol}$ Hi-Mod problems
3. Storage in the response matrix
4. Compute $\tilde{R} = \frac{1}{N_{sol}} V V^T$
5. SVD

**Online:**

1. Selection of the number of singular values based on Eq. (1.10)
2. Galerkin projection
3. Return to the original space $u = \Phi^k \tilde{u}$

Algorithm 1.1: Hi-POD offline and online algorithm.
Chapter 2
Programming design of Hi-POD in LifeV

The aim of this project is to solve an advection-diffusion-reaction problem with the Hi-POD technique, described in the previous chapter, in 3D domains.

Starting from the code existing in the LifeV environment, in particular referring to the module himod, we have developed a Hi-POD solver that solve 3D domains with rectangular and circular section. In particular we have implemented:

- an assembler to build the system according to the Hi-POD theory;
- a Hi-POD solver;
- additional tools containing some utilities for the Hi-POD algorithm and bridges to connect with Hi-Mod.

In the following section we describe in more details our programming design and the outline of our code. Henceforward for the type of the variables we refer to the following type definitions:

```
enum sectionGeometry { Rectangular, Circular } ;
typedef RegionMesh<LinearLine> mesh_Type;
typedef boost::function<Real(const Real&, const Real&, const Real&, const Real&, const ID&)> function_Type;
typedef MatrixEpetraStructured<Real> matrix_Type;
typedef VectorEpetraStructured vector_Type;
typedef VectorSmall<3> TreDvector_type;
typedef Epetra_SerialDenseMatrix DenseMatrix;
typedef Epetra_SerialDenseVector DenseVector;
```
typedef HiModAssembler<
    mesh_Type, matrix_Type, vector_Type,
    Rectangular, Block>
    himodrect_Type;

typedef HiModAssembler<
    mesh_Type, matrix_Type, vector_Type,
    Circular, Block>
    himodcirc_Type;

2.1 Hi-Pod Assembler class

The HiPodAssembler class is used to deal with the Hi-Mod algorithm and to compute the offline phase. In particular, it’s necessary to assemble and solve the system with the Hi-Mod technique and to manage the different nature of the section.

template<
    typename mesh_type,
    typename matrix_type,
    typename vector_type,
    int N, int PA>
class HiPodAssembler;

The template parameters mesh_type, matrix_type and vector_type indicate respectively the type of the mesh, the matrix and the vector we’re working with; N is a flag for the section geometry (0 for rectangular and 1 for circular for now) and PA stands for the type of parallelization for the Hi-Mod solver. For this reason this is a template class with two specialization in order to work on a parallelepiped and a cylindrical domain.

Both the specialization have the following attributes:

private:
    /// Hi-Mod informations [himod_Type = himodrect_Type,
    /// himodcirc_Type]
    boost::shared_ptr<himod_Type> HM;
    /// System matrix
    boost::shared_ptr<matrix_Type> systemMatrix;
    /// System rhs
    boost::shared_ptr<vector_Type> rhs;

and two main methods:

- **buildSystem:** to assemble the matrix and the right hand side of the problem that we solve. It recives as input the parameters and, according to the Hi-Mod algorithm, it defines the finite element space and the modal basis space (Code 2.1);

- **HiModSolver:** solve the problem using the Hi-Mod technique. Inside it calls **buildSystem** to create the matrix and the rhs and then thanks to the **PreconditionerIfpack** and **LinearSolver** modules, provided by LifeV, return the solution. (Code 2.3).

Parallelepiped

Similar to the structure of the HiModAssembler.hpp, in this specialization we have one more attribute that corrisponds to the MPI partition of the data.
Using polymorphism and overloading, instead, we have four different methods to work with constant and non constant parameter:

Listing 2.1: Declaration of buildSystem with constant parameters

```c
/**
@param Real value of mu
@param VectorSmall<3> values of beta
@param Real value of sigma
@param boost::function force term
@param boost::function Dirichlet inflow data
@param bool verbose
@param GetPot input parameter
@param std::string left boundary condition
@param std::string right boundary condition
@param std::string upper boundary condition
@param std::string lover boundary condition
@param bool flag for Dirichlet inflow
*/
void buildSystem (Real const & mu , TreDvector_type const & beta , const Real & sigma , const function_Type & f, const function_Type & g, bool verbose , const GetPot & dataFile , const std :: string & left , const std :: string & right , const std :: string & up , const std :: string & down , const bool & inflow_Dir );
```

Listing 2.2: Declaration of buildSystem with non constant parameters

```c
void buildSystem (const function_Type & mu , const function_Type & beta , const function_Type & sigma , const function_Type & f, const function_Type & g, bool verbose , const GetPot & dataFile , const std :: string & left , const std :: string & right , const std :: string & up , const std :: string & down , const bool & inflow_Dir );
```

Listing 2.3: Declaration of HiModSolver with constant parameters

```c
/**
@param Real value of mu
@param VectorSmall<3> values of beta
@param Real value of sigma
@param boost::function force term
@param boost::function Dirichlet inflow data
@param bool verbose
@param GetPot input parameter
@param bool flag for Dirichlet inflow
@param std::string left boundary condition
@param std::string right boundary condition
@param std::string upper boundary condition
@param std::string lover boundary condition
@param bool flag to export
*/
```
Listing 2.4: Declaration of HiModSolver with non constant parameters

```cpp
vector_Type HiModSolver (const function_Type & mu, const function_Type & beta, const function_Type & sigma, const function_Type & f, const function_Type & g, bool verbose, const GetPot & dataFile, const bool & inflow_Dir, const std::string & left, const std::string & right, const std::string & up, const std::string & down, bool exporter = 0);
```

Even if it’s not necessary we decided to keep the constant version because it’s much more efficient than the non constant one. Indeed in the non constant parameter version it takes a lot of time to evaluate the parameters in all the points of mesh. However the force term and the Neumann boundary conditions could be non-constant because they are not effective on computational time.

Figure 2.1: HiPodAssemblerRectangular graph
Cylinder

In this specialization we have two more attributes that are needed for the post-processing:

```cpp
/// Modal Space Circular
boost::shared_ptr<ModalSpaceCircular> MB;
/// Degrees of freedom
UInt numdof;
```

In this case, we have the constant version of `buildSystem` and `HiModSolver` and the declaration is the same of Code 2.1 and Code 2.3. The programming design is similar except for the invocation of functions related to `ModalSpaceCircular.hpp` instead of `ModalSpaceRectangular.hpp`.

![HiPodAssemblerCircular graph](figure)

Figure 2.2: HiPodAssemblerCircular graph

### 2.2 Hi-Pod Solver class

This class is completely different from the previous one, because here we define methods to solve problem independent from the section.

Its attributes are:

```cpp
private:
    /// System matrix
```
DenseMatrix A;
/// System rhs
DenseVector b;
/// Solution
DenseVector pod_sol;
/// Input parameters
GetPot dataFile;

There are three main methods to make respectively the sampling, the offline phase and the projection of the system, and two utilities:

- `rank` calculates the number of singular values which are used to reduce the system (see Eq. 1.10).
- `UniformSampling` computes an uniform sampling of a vector in a chosen range.

**Sampling**

For each non constant parameter there is a specific method that computes the uniform sampling. Instead for the constant version there is also available a smarter one based on Peclet evaluation:

```cpp
int PecletSampling(DenseVector & mu, DenseVector & betax,
                   DenseVector & betay, DenseVector & betaz, DenseVector & sigma);
```

This smart sampling works in the following way:

1. We start with an uniform sampling of all the data.

2. For half of the coefficients we select the values of \( \mu \) and \( \beta \) that correspond to the greatest value of the Peclet number (see Fig.(2.3));

![Figure 2.3: This figure shows the scheme for the selection of the highest Peclet](image)

3. We choose the remaining values from the uniform sampling excluding the ones that have already been considered.
// Initialization
int k = 0;
int elimina = 0;
Real max_value = 0;
std::vector<std::pair<int, int>> indices;
indices.emplace_back(N_SOL - 1, 0);

// Use the sampling driven by the Peclet only on half number of offline coefficients
while (k < N_SOL / 2) {
    for (UInt i = 0; i < indices.size(); i++) {
        if (max_value < beta[indices[i].first] / mu_help[indices[i].second]) {
            max_value = beta[indices[i].first] / mu_help[indices[i].second];
            mu[k] = mu_help[indices[i].second];
            betax[k] = betax_help[indices[i].first];
            betay[k] = betay_help[indices[i].first];
            betaz[k] = betaz_help[indices[i].first];
            elimina = i;
        }
    }
    auto ii = indices[elimina].first;
    auto jj = indices[elimina].second;
    if (ii - 1 > 0)
        indices.emplace_back(ii - 1, jj);
    if (jj + 1 < N_SOL)
        indices.emplace_back(ii, jj + 1);
    indices.erase(indices.begin() + elimina);
    for (UInt i = 0; i < indices.size(); i++) {
        if (indices[i].first == ii && indices[i].second == jj) {
            indices.erase(indices.begin() + i);
            break;
        }
    }
    max_value = 0;
    k++;
}

OfflinePhase

OfflinePhase is a template method that makes the offline phase of the Hi-Pod algorithm and the template parameter is an integer flag to specify the section (from now 0 corresponds to rectangular, 1 for cylindrical).

/**
@tparam HiPodAssembler<mesh_Type, matrix_Type, vector_Type, int N, Block> system data
*/
@param function_Type function related to the force term
@param function_Type function related to the Neumann inflow boundary condition
@param bool to print informations and errors

*/

template <int N> void OfflinePhase ( HiPodAssembler<
    mesh_Type, matrix_Type, vector_Type, N, Block >& HPA , const
    function_Type& f, const function_Type& g, bool verbose =
    0);

Initially OfflinePhase selects the parameters of the problems to be solved with the sampling methods described above. Then it solves \( N_{SOL} \) problems using HiPodAssembler::HiModSolver. OfflinePhase also stores each solution as an additional column of the response matrix.

for (UInt k = 0; k < N_SOL; k++) {
    // Assembling force term
    \( F_0 \) = force0[k];
    \( F_x \) = forcex[k];
    \( F_y \) = forcey[k];
    \( F_z \) = forcez[k];
    \( G_0 \) = dirdata[k];
    if (constant_param){
        // Assembling beta
        beta[0] = betax[k];
        beta[1] = betay[k];
        beta[2] = betaz[k];
        auto solution = HPA.HiPodAssembler<
            mesh_Type, matrix_Type, vector_Type, C, Block>::HiModSolver(
            \( \mu[k] \), beta, sigma[k], f, g, verbose, dataFile,
            inflow_Dir_offline, left, right, up, down);

        addColumn ( M, solution );
    } else {
        // Functions coefficients update
        \( \mu_0 \) = mu[k]; \( \mu_X \) = mx[k]; \( \mu_Y \) = my[k]; \( \mu_Z \) = mz[k];
        betaX0 = betax[k]; betaXX = bxx[k]; betaY0 = betay[k];
        betaYY = byy[k]; betaZ0 = betaz[k]; betaZZ = bzz[k];
        sigma0 = sigma[k]; sigmaX = sx[k];
        sigmaY = sy[k]; sigmaZ = sz[k];

        auto solution = HPA.HiPodAssembler<
            mesh_Type, matrix_Type, vector_Type, C, Block>::HiModSolver(
            \( \mu_{fun} \), beta_{fun}, sigma_{fun}, f, g, verbose, dataFile,
            inflow_Dir_offline, left, right, up, down);
        addColumn ( M, solution );
    }
}
Then we compute the singular values decomposition with *thinSVD*\(^1\) from SerialDenseUtilities.hpp in the *mor* sub folder. Finally we export the left singular vectors matrix and a vector containing the singular values.

```cpp
// Calculate and subtract the average of the response matrix
update(average_Matrix(M), -1.0, M);
// SVD
auto result = thinSVD(M);
// Left singular values matrix
DenseMatrix Phi;
multiply(result.second, result.first.U, Phi, 0);
```

In this way we don’t need to run the offline phase every time, but just once and then to solve an ADR problem we have only to load the matrix and the vector. To see the graph of the function invocation look at Fig. 2.4.

### Projection

This method is the heart of the online phase and calculates the solution by the projection of the system on the space generated by the reduced basis.

```cpp
DenseVector projection(const DenseMatrix& Phi, const DenseVector& diag, int& nbSV, bool calculate = 1, Real threshold = 1-1e-10)
```

*projection* allows the user to select the number of singular values (*nbSV*) or passing it or exploiting the function *rank*.

```cpp
if(calculate)
    nbSV = rank(diag, threshold);
// Reduce the system
PhiK = principalSubMatrix(Phi, Phi.M(), nbSV, Copy);
PtAP(PhiK, this->A, R, 1);
multiply(PhiK, 1, this->b, coeff, f_help, coeff);
```

Then we have to solve this new reduced system with *Epetra_LAPACK*\(^2\) using a general solver\(^3\):

```cpp
// Solve the system
Epetra_LAPACK solver;
std::vector<int> IPIV(R.M());
int INFO = 0;
solver.GESV(R.M(), 1, R.A(), R.LDA(), &IPIV[0], f_help.Values()
    , f_help.Length(), &INFO);
```

In the end we expand the reduced solution to the original space and return it to the user.

\(^1\)We choose the thin SVD because it’s more efficient on rectangular matrices.

\(^2\)Solver in the Trilinos Project: www.trilinos.org

\(^3\)GESV computes the solution to a real system of linear equations \(Ax = b\). LU decomposition with partial pivoting and row interchanges is used to solve the system.
2.3 Hi-Pod Serial Dense Utilities

`HiPodSerialDensUtilities.hpp` contains some helping functions for the Hi-Pod algorithm.

- add a column to a matrix; needed in `OfflinePhase`
  ```
  int addColumn (DenseMatrix& A, const vector<Type>& v);
  ```

- calculate the element-wise average of a matrix; needed in `OfflinePhase`
  ```
  DenseMatrix average_Matrix (const DenseMatrix& A);
  ```

- multiply a matrix for a scalar; needed in `average_Matrix`
  ```
  int dotScalar (DenseMatrix& A, Real c);
  ```
• Importers
  ◦ import a matrix from a file in the DenseMatrix format
    ```cpp
    int importDenseMatrix ( DenseMatrix& A, const std::string& path );
    ```
  ◦ import a vector from a file in the DenseVector format
    ```cpp
    int importDenseVector ( DenseVector& A, const std::string& path );
    ```

• Exporters
  ◦ export a matrix into a file in the MatrixMarket format
    ```cpp
    int exportDenseMatrix ( const DenseMatrix& A, const std::string& path );
    ```
  ◦ export a vector into a file in the MatrixMarket format
    ```cpp
    int exportDenseVector ( const DenseVector& A, const std::string& path );
    ```

2.4 Converter Utilities

For this project we use principally himod module and mor sub folder. We choose to design classes, methods and functions such that each one can be separately developed.

Since Hi-Pod is the composition of two model reductions, we have to create an interface to transform a dense matrix into a sparse matrix and a dense vector into a distribute vector and vice-versa.

For this reason we have realized the following converters:

• **StructToDenseVectorConverter**: from vector\_Type to DenseVector
  ```cpp
  Epetra_SerialDenseVector StructToDenseVectorConverter ( const boost::shared\_ptr<VectorEpetraStructured> v );
  Epetra_SerialDenseVector StructToDenseVectorConverter ( const VectorEpetraStructured& v );
  ```

• **DenseToStructVectorConverter**: from DenseVector to vector\_Type
  ```cpp
  VectorEpetraStructured DenseToStructVectorConverter ( const Epetra_SerialDenseVector& v );
  ```

• **StructToDenseMatrixConverter**: from matrix\_Type to DenseMatrix
  ```cpp
  Epetra_SerialDenseMatrix StructToDenseMatrixConverter ( const boost::shared\_ptr<MatrixEpetraStructured<Real>> & A );
  Epetra_SerialDenseMatrix StructToDenseMatrixConverter ( const MatrixEpetraStructured<Real> & A );
  ```
Chapter 3

Numerical Test and Results

3.1 Test case 1 - The parallelepiped domain

In this case we solve the problem

\[
\begin{align*}
-\mu(x) \Delta u(x) + b(x) \nabla u(x) + \sigma(x) u(x) &= f(x) \quad \text{in } \Omega \\
u &= 0 \quad \text{on } \Gamma_D \\
\frac{\partial u}{\partial n} &= 0 \quad \text{on } \Gamma_N
\end{align*}
\]

in \(\Omega = [0, 0.3] \times [0, 0.1] \times [0, 0.1]\) with

\[
\Gamma_D = \{x = 0\} \times [0, 0.1] \times [0, 0.1] \\
\cup \{0, 0.3\} \times \{y = 0\} \times [0, 0.1] \cup \{0, 0.3\} \times \{y = 0.1\} \times [0, 0.1] \\
\cup \{0, 0.3\} \times [0, 0.1] \times \{z = 0\} \cup \{0, 0.3\} \times [0, 0.1] \times \{z = 0.1\}
\]

and \(\Gamma_N = \{x = 0.3\} \times [0, 0.1] \times [0, 0.1]\).

Parameters form

\[
\mu = \mu_0 + \mu_x x + \mu_y y + \mu_z z, \quad \sigma = \sigma_0 + \sigma_x x + \sigma_y y + \sigma_z z,
\]

\[
b = \begin{bmatrix}
b_{x0} + b_x x \\
b_{y0} + b_y y \\
b_{z0} + b_z z
\end{bmatrix}, \quad f = f_0 + f_x x + f_y y + f_z z;
\]
Offline

\[ N_{\text{prob}} = 30, \quad m = 25 \]

\[ \mu_0 \in [1, 1000], \quad \mu_x, \mu_y, \mu_z \in [0, 50]; \]
\[ \sigma_0 \in [0, 100], \quad \sigma_x, \sigma_y, \sigma_z \in [0, 10]; \]
\[ b_{x0}, b_{y0}, b_{z0} \in [-100, 100]; \]
\[ b_x, b_y, b_z \in [-10, 10]; \]
\[ f_0 \in [-100, 100], \quad f_x, f_y, f_z \in [-50, 50]. \]

Online

\[ \mu = 47.75 + 2.2y + 1.7z \]

\[ \sigma = 5.4 + 0.9x \]

\[ b = \begin{bmatrix} 21.72 + 0.05x \\ 0.8 - 4.1y \\ 0.1 - 1.2z \end{bmatrix}; \]
\[ f = 86 + 35.4x + 13.9y - 2.3z. \]

We use 200 finite elements on the main direction and also the Hi-Mod solution is solved with 25 number of modes.

We have solved this problem with four different choices of number of singular values deduced from the plot of the spectrum. As we can see in Figure (3.1) some truncation can be done for example in \( nbsv = 2 \), \( nbsv = 4 \) and \( nbsv = 9 \). The choice of \( nbsv = 29 \) is the one automatically calculated with a tolerance of \( 10^{-10} \) in the calculus of the rank of the thin svd matrix.

![Figure 3.1: The picture at the top shows the singular value decomposition of snapshot matrix \( \tilde{R} \). In particular we highlight two choices we have made: 4 (bottom left) and 9 (bottom right).]
Figure 3.2: These pictures show the solution at the outflow boundary with different choices of number of singular values ($nbsv$).
Figure 3.3: These pictures show the cut of the solution on the $x-y$ plane with different choices of number of singular values ($nsv$).
In Fig. (3.2) we show the solution evaluated on the Neumann outflow boundary. As we can see 2 singular values are not enough to catch the right behaviour but increasing to 4 the number of singular values the main tendency is captured. With 9 we get a very good solution, not so different from the one with 29 singular values and the Hi-Mod solution. For this reason, between this two solutions it’s convenient to choose the 9 singular values one since the computational effort is lower.

The same discussion can be held for Fig. (3.3) that represent a section on the $x - y$ plane.

For a quantitative analysis, in the following table Tab. (3.1) and figure Fig. (3.4) we can see the errors made in all the cases analysed before.

<table>
<thead>
<tr>
<th>Error</th>
<th>$r = 2$</th>
<th>$r = 4$</th>
<th>$r = 9$</th>
<th>$r = 29$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$|u_{x,\text{himod}} - u_{x,\text{hipod}}|_{L^2(\Omega)}$</td>
<td>$1.542 \cdot 10^{-5}$</td>
<td>$1.398 \cdot 10^{-6}$</td>
<td>$5.056 \cdot 10^{-7}$</td>
<td>$1.914 \cdot 10^{-8}$</td>
</tr>
<tr>
<td>$|u_{x,\text{himod}} - u_{x,\text{hipod}}|_{L^2(\Omega)}$</td>
<td>0.8773</td>
<td>0.0795</td>
<td>0.0287</td>
<td>0.0011</td>
</tr>
</tbody>
</table>

Table 3.1: Errors computed between Hi-Mod and Hi-Pod solutions.
3.2 Test case 2 - The circular domain

In this case we solved the problem

\[
\begin{aligned}
-\mu(x) \Delta u(x) + b(x) \nabla u(x) + \sigma(x) u(x) &= f(x) \quad \text{in } \Omega \\
u &= 0 \quad \text{su } \Gamma_D \\
\mu \frac{\partial u}{\partial n} &= 0 \quad \text{su } \Gamma_N
\end{aligned}
\]

(3.2)

in \( \Omega = [0, 0.3] \times S \) where \( S \) is a circular section with \( \rho \in [0, 0.05] \) and \( \theta \in [0, 2\pi] \) with \( \Gamma_D = \{(0, 0.3) \times \{\rho = 0.05\} \times [0, 2\pi]\} \) and \( \Gamma_N = \{(x = 0) \times S\} \cup \{(x = 0.3) \times S\} \).

\[
\begin{array}{c}
\Gamma_D \\
\Gamma_N \\
\Gamma_N
\end{array}
\]

Parameters form

\[
\mu = \mu_0, \quad \sigma = \sigma_0, \quad b = \begin{bmatrix} b_x \\ b_\rho \\ b_\theta \end{bmatrix}, \quad f = f_0 + f_x x + f_\rho \rho + f_\theta \theta;
\]

\text{Offline} \quad N_{\text{prob}} = 30, \quad m = 25

\[
\begin{align*}
\mu_0 &\in [1, 1000], \\
\sigma_0 &\in [0, 100], \\
b_x &\in [-100, 100], \\
b_\rho, b_\theta &\in [-10, 10], \\
f_0 &\in [-1000, 1000], \\
f_x, f_\rho, f_\theta &\in [-1000, 1000].
\end{align*}
\]

\text{Online} \quad \mu_0 = 106.4, \quad \sigma_0 = 56.57

\[
\begin{bmatrix}
-78.7 \\
8.9 \\
-2.1
\end{bmatrix}
\]

\[
f = 1000 + 261 x - 110.3 \rho - 149.9 \theta.
\]

We use 200 finite elements on the main direction and also the Hi-Mod solution is solved with 25 number of modes.

We have solved this problem with four different choices of number of singular values deduced from the plot of the spectrum. As we can see in Figure (3.5) some truncation can be done for example in \( nbsv = 2 \), \( nbsv = 6 \) and \( nbsv = 14 \). The choice of \( nbsv = 29 \) is the one automatically calculated with a tolerance of \( 10^{-10} \) in the calculus of the rank of the thinSVD matrix.
Figure 3.5: The picture at the top shows the singular value decomposition of snapshot matrix $\tilde{R}$. In particular we highlight two choices we have made: 6 (bottom left) and 14 (bottom right).
Figure 3.6: These pictures show the solution at the outflow boundary with different choices of number of singular values ($nsv$).

(a) Hi-mod solution

(b) Solution with $nsv = 2$

(c) Solution with $nsv = 6$

(d) Solution with $nsv = 14$

(e) Solution with $nsv = 22$
Figure 3.7: These pictures show the cut of the solution on the $x - y$ plane with different choices of number of singular values ($nsv$).
As before in Fig.(3.6) there is the solution evaluated on the Neumann outflow boundary. In this case also with two singular values we can capture a right behaviour, but only qualitative. Instead with 6, 14 or 29 we have a very good solution as we can see also quantitative from the errors computed in the next table Tab. (3.2).

Again, we observe in Fig.(3.7) that also on the $x-y$ plane there are good results, and since all the last three simulations have similar tendency, we prefer the 6 singular values solution for the lower computational cost.

The errors computed in this case are lower than the first (Tab.(3.2) and Fig.(3.8)), but we can place this difference to the fact that here we solve a problem with outflow and inflow Neumann conditions that give optimal results for this algorithm.

<table>
<thead>
<tr>
<th>Error</th>
<th>$r = 2$</th>
<th>$r = 6$</th>
<th>$r = 14$</th>
<th>$r = 22$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$|u_{x_{\text{hi-mod}}} - u_{x_{\text{hi-pod}}}|_{L^2(\Omega)}$</td>
<td>$1.249 \cdot 10^{-9}$</td>
<td>$3.491 \cdot 10^{-9}$</td>
<td>$1.677 \cdot 10^{-13}$</td>
<td>$5.541 \cdot 10^{-16}$</td>
</tr>
<tr>
<td>$\frac{|u_{x_{\text{hi-mod}}} - u_{x_{\text{hi-pod}}}|<em>{L^2(\Omega)}}{|u</em>{x_{\text{hi-mod}}}|_{L^2(\Omega)}}$</td>
<td>$7.904 \cdot 10^{-2}$</td>
<td>$2.208 \cdot 10^{-5}$</td>
<td>$1.061 \cdot 10^{-9}$</td>
<td>$3.504 \cdot 10^{-12}$</td>
</tr>
</tbody>
</table>

Table 3.2: Errors computed between Hi-Mod and Hi-Pod solutions.
Chapter 4

Tutorial

In the sub folders `18rect_hipod` and `18circ_hipod` there are the tests to apply Hi-POD to a given problem. The two codes act in a similar way except for the invocation of the assembler.
Every sub folder contains a `main.cpp` and a txt file `data`.

Data

With this file the user can set every specific features of the problem:

- domain and mesh, in particular the dimensions of the domain and the number of finite elements and modes;
- preconditioner and solver;
- offline, in particular:
  - if the user wants to do the offline phase or just loads existing SVD matrices;
  - the number of problems to solve in this phase;
  - a flag for constant [= 0] or non constant [= 1] parameters (for the cylinder only the constant is available);
  - ranges for the constant and non constant parameters (for the cylinder only the constant is available);
- online, in particular
  - a flag for homogeneous Neumann [= 0] or Dirichlet [= 1] inflow boundary condition;
  - definition of the parameters of the problem;
- projection, in particular
  - a flag for the automatic [= 1] or manually [= 0] calculation of the number of singular values;
  - tolerance for the automatic calculation of the number of singular values;
  - number of singular values used in the manually calculation.
Main

The main is quite simple and executes the following passages:

i) load the data,
ii) make the offline phase (only if requested),
iii) make the online phase to compute the Hi-POD solution,
iv) compute the Hi-Mod solution,
v) export both solutions in .vtk format,
vi) compute the error $||u_{himod} - u_{hipod}||_{L^2(\Omega)}$. 
Bibliography


