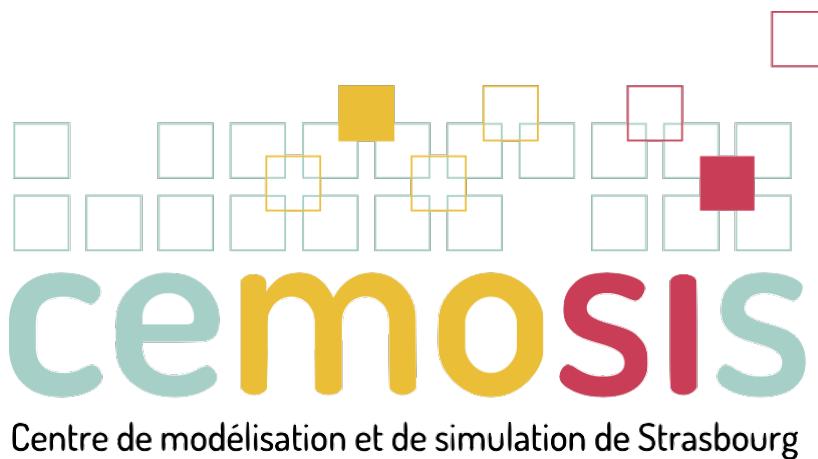

Order reduction method for a porous-elastic model and coupling with an ODE system

2nd year internship

Thomas Saigre



Master 2 CSMI

UFR de Mathématique et d'Informatique
Université de Strasbourg

supervised by

Christophe PRUD'HOMME

February 1th 2021 – July 30th 2021

Compilation of this document completed on **août 22 2021**

Contents

1	Introduction	5
1.1	Acknowledgments	5
1.2	Context of the internship	5
1.3	Organisation during internship	5
1.4	Objectives	6
I	Modelisation and simulation	7
2	Geometrical modeling	9
2.1	Anatomy of the eye	9
2.2	Initial geometry	9
2.3	Compatibility issue	11
2.4	Meshed geometry	11
2.5	Run the Salome pipeline	14
2.6	Computational performances	15
3	Verification and validation of the geometry	17
3.1	Tests with functions in the space	17
3.2	Convergence test	17
3.3	Heat transfer	18
4	Physical modeling	23
4.1	Tissue perfusion	23
4.2	Feel++ application <code>feelpp_toolbox_hdg_coupledpoisson</code>	25
5	Implementation aspects of the model	27
5.1	Tests with the application	27
5.2	Functional Mock-up Interface	27
5.3	Model 0D with dymola	27
5.4	Linear case	29
II	Methods and discretization	33
6	HDG	35
6.1	Darcy equations with IBC	35
6.2	HDG formulation	36
6.3	Discrete variationnal formulation	37
6.4	Static condensation	38
6.5	Local solver	40
6.6	Flux operator	40
6.7	Global resolution	40
6.8	Implementation in Feel++	41
7	Time-splitting algorithm	43
7.1	Discretisation	43
7.2	Variational problem for step 1	44
III	Reduced Order Methods	47
8	Theory and methods	49
8.1	Thermal-fin model	49
8.2	Reduced Basis and EIM	50
8.3	<i>A posteriori</i> error estimation	51

8.4	Greedy algorithm	52
9	Implementation	55
9.1	Implementation using only NumPy	55
9.2	Case generator	56
9.3	Pyfeelpp-mor	58
9.4	The script <code>toolboxmor.py</code>	58
9.5	Module PETSc4py	59
9.6	Adaptation to PETSc	60
9.7	Description of the code produced	61
9.8	First results	64
9.9	Results of the greedy algorithm	65
10	Applications	69
IV Appendix		71
A	Feel++	73
A.1	Building Feel++	73
A.2	Handling the software	73
A.3	Open the geometry with Paraview	75
A.4	<code>feelpp_mesh_partitioner</code>	76
A.5	Pyfeelpp	76
B	Script <code>toolboxmor.py</code>	77
B.1	Initialisation of the environment	77
B.2	Compute the offline decomposition	77
B.3	Online computations	81
Bibliography		83

1.1 Acknowledgments

Before presenting the work I made during this internship, I would like to thank the team Cemosis in the IRMA who welcomed me during these six months. Especially I thank Christophe Prud'homme who supervised me and helped me when I was lost in the new notions I discovered during the internship. Moreover, I thank Romain, Vincent, Zohra, Luca, Philippe, François, Yannick, Abdoulaye, Christophe (Trophime), and Joubine from the team, which gathered every Monday, even if the health context prevents those meeting to stand face-to-face. I also thank my colleagues intern, which I shared in the office : Killian, Khaoula, Sofian, Romain, Youssef, and Guillaume. Moreover, I thank Lilian who helped me to have a better understanding of how the eye is composed in the body [4].

Finally, I would like to thank Christophe Prud'homme, Marcel Szopos, Giovanna Guidoboni, and Lorenzo Sala who helped me [14, 8] and accepted me to direct and advise for the Ph.D. thesis that will begin in October.

1.2 Context of the internship

The eye, with its special connection to the brain and its accessibility, offers non-invasive access to a large set of measures that might help in the early diagnosis and clinical care of neurodegenerative diseases. But the characterization of ocular biomarkers representing the cerebral state is far from trivial : many factors can influence measurements that can vary among individuals.

In the framework of the Eye2brain project [16], the *Ocular Mathematical Virtual Simulator* (OMVS) is developed. This is a reliable and efficient computational framework of the Eye2Brain system, allowing for computer-aided interpretations of the clinical data.

This internship takes place after the thesis of Lorenzo Sala [17], who developed three levels of mathematical architecture of the OMVS.

1.3 Organisation during internship

The internship took place during a special context of the pandemic. Because of it, all meetings could not be held in physic, but mostly on Zoom. It didn't avoid weekly team meetings to take place every Monday, with all the collaborators of Cemosis. The main advantage of the visio-conference is that people far from Strasbourg could attempt them.

During those meetings, we shared our progress, asked questions. Sometimes, I made a more detailed presentation on different methods or models I worked on. The slides of those presentations are gathered in the `Talks` directory on the Github repository `eye2brain-doc`.

- Presentation of HDG method (on March, 8th), see chapter 6.
- Summary of the first two months of the internship (on April, 6th).
- Presentation of 3D-0D coupling and splitting algorithm (on April, 26th), see chapter 4.
- Presentation to the École Doctorale (on June, 16th), in front of the jury for the PhD thesis funding.
- Defense of the internship (on August 26th or 27th).

During the internship, I mainly worked on three Github repositories :

- `feelpp` [5], which contains the source code of Feel++ (see appendix A). I made some contributions to this repository, especially in a branch dedicated to Python development (see issue 1626). This branch has been merged with the main branch. I also documented my contribution in Feel++ documentation.
- `eye2brain`, which is the repository dedicated to the project Eye2Brain. It is possible that the function described in this report will be updated in the future ¹
- `eye2brain-doc`, containing this present report the supports of the presentation I made.

¹at the compilation of the report, the state was on commit `a4cccd2abef8bcc6c85f54e3b1ec3d0438fef9472`

1.4 Objectives

The main objective of the internship is to develop a reduced-order method for Darcy problem in mixed form, for instance, to use it for sensitivity analysis.

To reach this objective, I will have to handle some tools and get familiar with the objects used. The tools used will be detailed in this report, we can mainly name Feel++ (A), Dymola (5.2) or Salome (2.2) for the softwares used, and the HDG method (6) and the time-splitting algorithm (chapter 7) for the method studied.

This report is divided into three parts. The first one is dedicated to the models and how to simulate them, presenting the geometrical and physical modeling of the eye. In the second part, we will focus on theoretical methods developed to simulate the models described in the first part. Then the third part details a reduced-order model : the reduced basis method.

Part I

Modelisation and simulation

In this part, we will focus on the modelization of the geometry of the eye and how to simulate the models. The chapter 2 will present the geometrical modeling of the eye, and how to mesh this geometry. In chapter 3, we will study the verification and validation approach for the geometry generated. Finally, chapter 4 will present the physical modeling and the tools used to simulate it.

2.

Geometrical modeling

The geometry of the eye was made some years ago, during Lorenzo's thesis [17], with a previous version of SALOME (7.4.0), but this script is no longer compatible with the latest version (9.6.0).

With the new version, the operations of SALOME don't lead to the same result, as we will see below. Moreover, in the code produced by Lorenzo, some parts of the eye (namely the posterior and anterior chambers) were not included.

2.1 Anatomy of the eye

The eye, as shown in figure 2.1, is an organ allowing to capture light and convert it into an electric signal, allowing the brain to interpret it. At the front of the eye, we have the *cornea* interacting with the outside and two-chamber, called *anterior* and *posterior* chamber filled with the *aqueous humor*. Around the eye, moving inward, we have several layers of tissues : the *sclera*, the *choroid*, the *retina* and the *vitreous humor*. The structure which separate the vitreous from the aqueous humor is composed of *iris*, *lens*¹ and *ligament*.

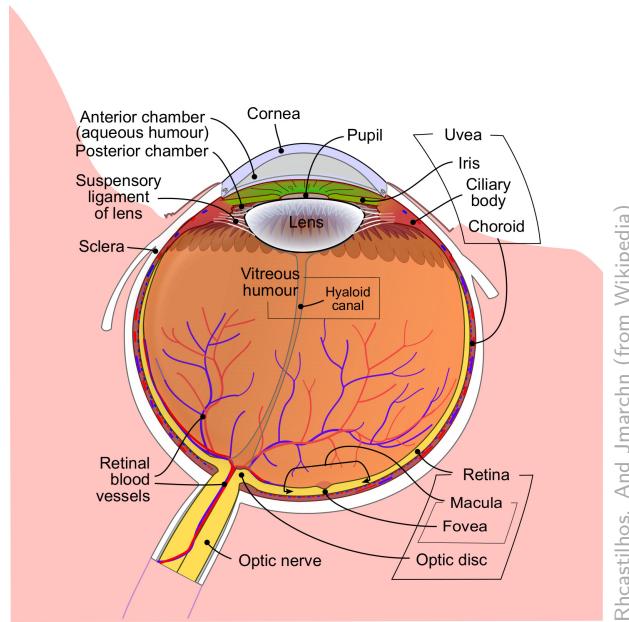


Figure 2.1: Anatomy of the eye

The *choroid* regulates the temperature and the volume of the eye. It also supplies the retina with nutrients. A large part of the blood circulating in the eye, 85% of it, because its main function is to conduct arteries and nerves to the other structures of the eye.

The *retina* is the light-sensitive layer. It is composed of many sub-layers and contains the cells that can capture the light, *rods* and *cones*. In our geometry, we are interested in the layer where the retinal ganglion is : these neurons are responsible for the transmission of the visual information from the retina to the brain through the optic nerve.

The *ophthalmic artery* supplies the blood to the eye. This vessel is located close to the optic nerve. From this artery, the *optic retinal artery* enters the optic nerve. In parallel, the *central retinal vein* drains the blood from the eye.

The *lamina cribrosa*² is an extension of the sclera, as presented on figure 2.2. It allows the retinal ganglion cells and the central retinal vessels to access the eyeball-protected environment. This part of the eye was central in the work of Lorenzo's thesis [17].

2.2 Initial geometry

The first step consists of loading a *STEP* file, which contains the geometry of the eye, untreated. This *STEP* file is large (more than 1 Mo), so it is stored with Git-lfs. This geometry is composed of many solids, representing the different parts of the eye. It is represented in figure 2.3, on an horizontal cut.

¹in french *cristallin*

²in french *lame criblée de l'œil*

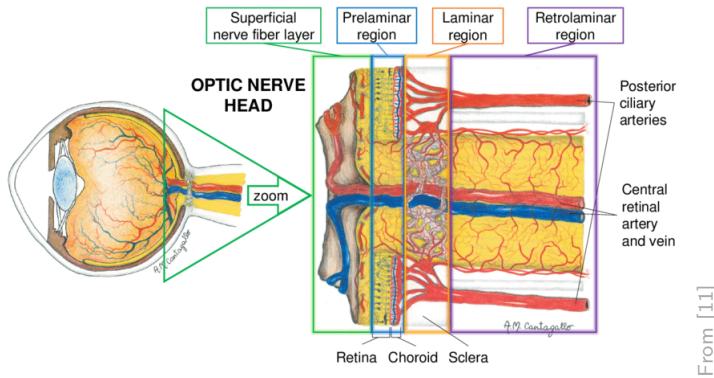


Figure 2.2: Anatomy of the back of the eye and vascular supply

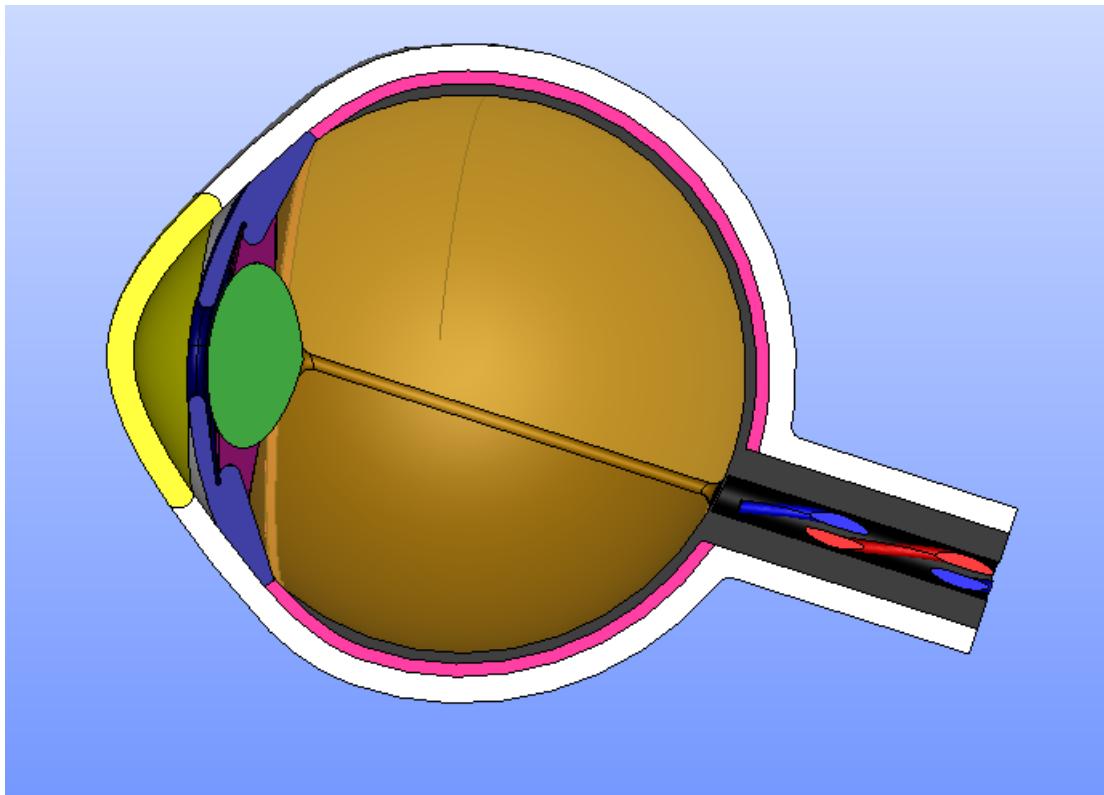


Figure 2.3: Horizontal cut of the eye, from the STEP file

The parts of the eye are like the following :

- The *cornea* (in yellow),
- the *iris* (in dark blue),
- the *vitreous humor* (in orange),
- the *ligament* (in purple),
- the *lens* (in green),
- the *choroid* (in pink),
- the *sclera* (in white),
- the *retina* (in black),
- the *optic nerve* (in grey),
- and the artery and vein (in red and blue respectively), inside the optic nerve.

In reality, the vein and artery in the optic nerve are not disposed of as in the geometry : as we can see on figure 2.1, the network of vein and artery goes further in the eye to irrigate the tissues in the blood. The actual vascular anatomy is shown on figure 2.2

The geometry from the STEP file is modified, especially at the back of the eye. New elements are created :

- The *aqueous humor* which is composed of the anterior and posterior chambers between the cornea and the

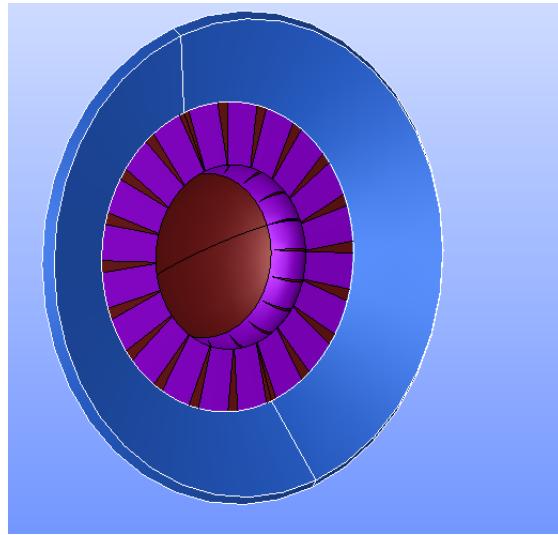


Figure 2.4: View of the ligament from the inside of the eye : on this figure, only **ligament**, **iris** and **aqueous humor** are shown.

ligament (this volume is not represented on figure 2.3 above, but it is represented in brown on figure 2.5(d)),

- The *lamina* (in light green),
- The *pia* (in cyan), the contour of the optical nerve.

One discussion was raised from the definition of the volume of the aqueous humor. As we will see later, in [10] this volume is separated into two sub-volumes : the anterior chamber corresponding to the part between the cornea and the iris, and the posterior chamber which is the part between the iris and the ligament. In the geometry loaded from the STEP file, those two chambers communicate through a hole corresponding to the pupil. From the figure in [6, Figure 1,p.163], it seems to be a hole here, allowing the aqueous humor to pass through it. After a discussion with a medicine, student [4], there is indeed a hole in this part of the eye. Furthermore, the geometry of the ligament in our geometry which has a sun-like shape, see figure 2.4, is quite different from reality. In the eye, the vitreous and aqueous humor are separated by the ligament which acts more like a sponge, while in our geometry the two *humors* are in direct contact.

The other elements are modified, for example, the sclera is cut among a plane.

The figure 2.5(a) shows an external view of the eye, which doesn't show a lot ! On figure 2.5(c) is shown a cut of the back part of the eye, where new elements are created.

Note that in the figures, the aqueous and vitreous humor is not always shown, because they would hide the inside of the eye.

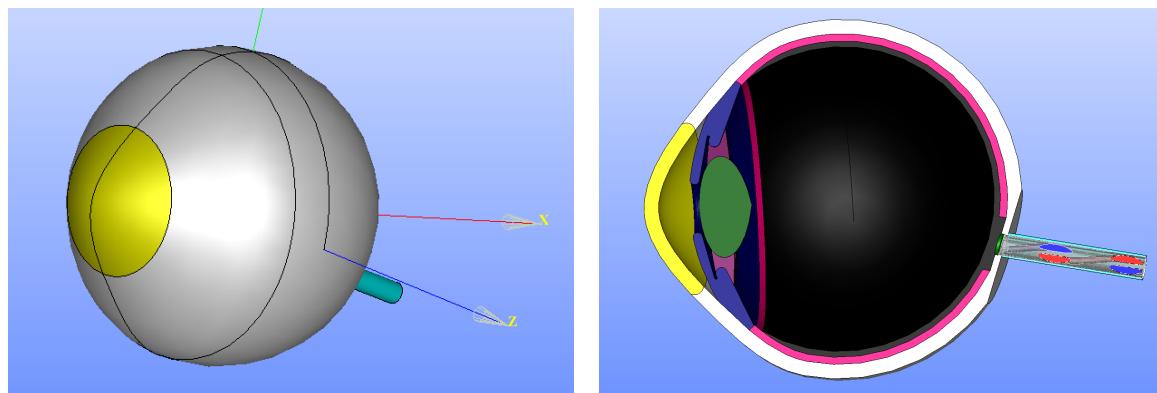
2.3 Compatibility issue

The main issue was that the code was made with SALOME v7.4.0 (referred as *v7* in the following), and the results of the operations made with the latest version (v9.6.0, referred as *v9*) are different. For example, when we extract with *v7* the face shapes of a modified version of the optic nerve (precisely the object *NewOptic_2*), it gives one face that is not extracted by *v9* anymore, this extra face is represented on figure 2.6(a). This face exists because, in the construction, the two opposite faces of the lamina are not parallel. This leads to folds on a face as we can see in figure 2.6(b), which caused an error in the code : when *v9* try to make operations on the face with the fold, such as intersection, the result is an empty object, while it is not on *v7*.

To solve this issue, we use another way to create the partition [18] using a plane to cut the optic nerve without any fold. See listing 1 to see the detail of the code.

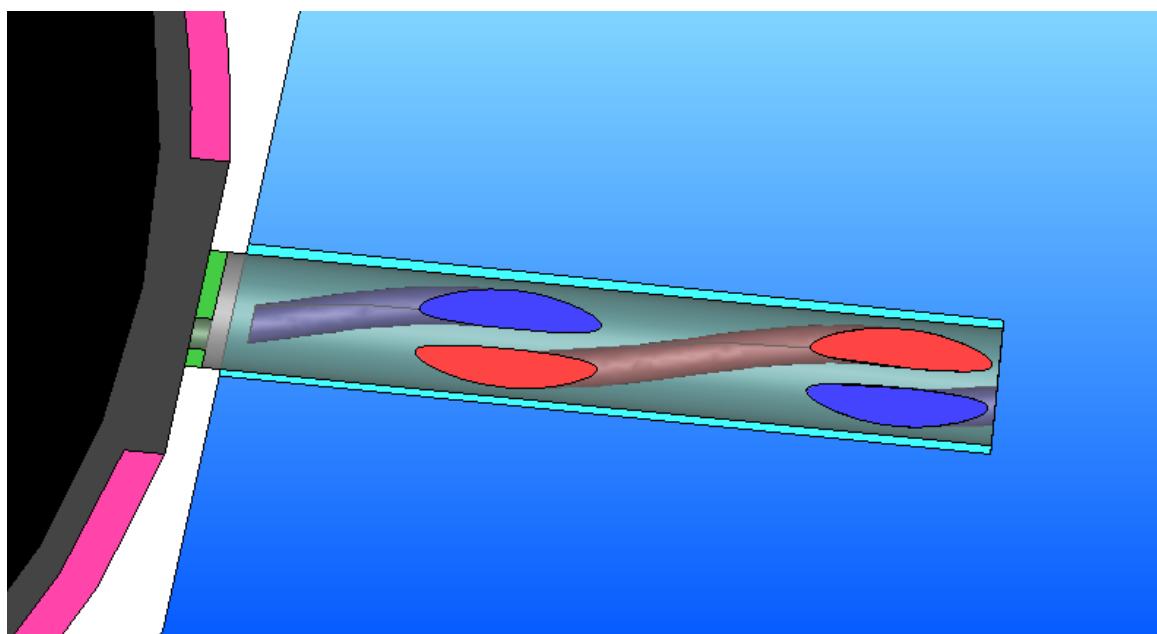
2.4 Meshed geometry

We can now mesh this geometry. With SALOME, we created groups of solids and faces, to mark them with names. Those names will be used in the future to set conditions in simulations.

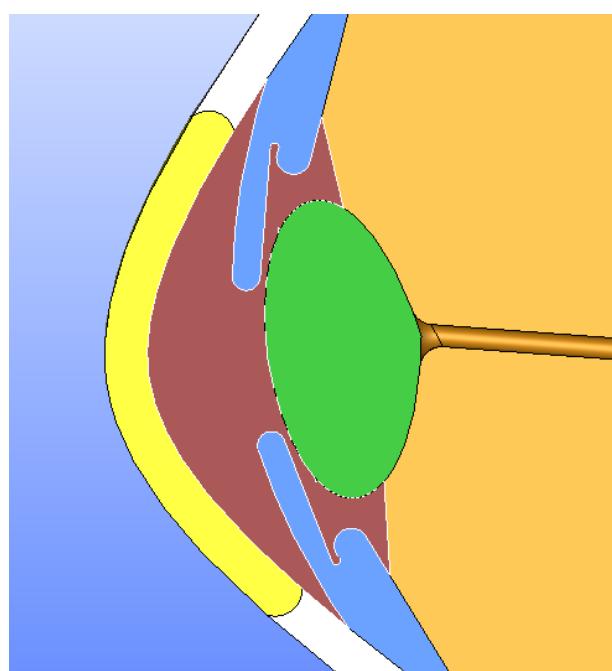


(a) External view of the eye

(b) Vertical cut of the eye



(c) Vertical cut of the back of the eye, after modification



(d) Vertical cut of the front of the eye, after modification

Figure 2.5: Geometry after modification

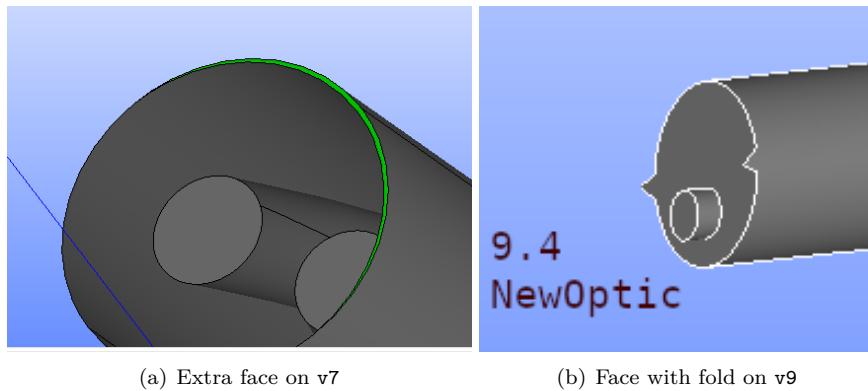


Figure 2.6: Issues on the geometry

```

1 [FaceOpt_1, FaceOpt_2, FaceOpt_3, FaceOpt_4, FaceOpt_5, FaceOpt_6, FaceOpt_7, FaceOpt_8, □
2   ↳ FaceOpt_9, FaceOpt_10] = geompy.ExtractShapes( OpticNerve_1, geompy.ShapeType["FACE"], True)
3 [EdgeOpt_1, EdgeOpt_2, EdgeOpt_3, EdgeOpt_4, EdgeOpt_5] = geompy.ExtractShapes( FaceOpt_5, □
4   ↳ geompy.ShapeType["EDGE"], True)
5 FaceToCutOpt = geompy.MakeFaces ( [EdgeOpt_1, EdgeOpt_2, EdgeOpt_4, EdgeOpt_5], True )
6 PlaneToCutOpt = geompy.MakePlaneFace(FaceToCutOpt, 10)
7 PartitionOpt = geompy.MakePartition([OpticNerve_1], [PlaneToCutOpt], [], [], geompy.ShapeType[
8   ↳ "SOLID"], 0, [], 0)
9 [laminaFace_1, laminaFace_2, laminaFace_3, laminaFace_4, laminaFace_5] = geompy.
10  ↳ ExtractShapes(Scaled_Lamina_w_hole_2, geompy.ShapeType["FACE"], True)
11 Plane_1 = geompy.MakePlaneFace(laminaFace_5, 5)
12
13 Partition_1 = geompy.MakePartition([OpticNerve], [Plane_1], [], [], geompy.ShapeType["SOLID"], □
14   ↳ 0, [], 0)
15 [PartOpticNerve_1, PartOpticNerve_2] = geompy.ExtractShapes(Partition_1, geompy.ShapeType['SOLID
16   ↳ '], True)
17 [OpticCut_1, OpticCut_2, OpticCut_3] = geompy.ExtractShapes( PartitionOpt, geompy.ShapeType[
18   ↳ "SOLID"], True )
19 NewOptic = geompy.MakePartition([OpticCut_1, PartOpticNerve_2], [], [], [], geompy.ShapeType[
20   ↳ "SOLID"], 0, [], 0)

```

Listing 1: Salome code to get the same NewOptic as 7 gave

Many faces are at the intersection between two solids of the eye. We name them after the two solids involved, classed in alphabetical order. For instance, the group of commons faces to OpticNerve and Pia is named « OpticNerve_Pia ». Table 2.1 tells whether or not such solid have interfaces or not.

Then other groups of faces are set, with the remaining faces on each solid :

- Lamina_Out
- BG_Iris
- Sclera_externalBC
- Lamina_Hole
- BC_Lens
- Selera_internalBC
- BC_Artery
- BG_Ligament
- BC_OpticNerve
- BC_Vein
- BG_Choroid
- BC_Pia
- BC_spherical_Retina
- BC_VitreousHumor
- Cornea_externalBC
- Cornea_internalBC

The crossed-out names in the previous list correspond to markers that were present in the previous geometry on v7 when aqueous and vitreous humor were not included.

The lamina gets a special mesh, with smaller elements than the other parts of the eye. This is because this part is crucial in the coupling of the model.

Finally, the 14 volume groups in the mesh are :

	AqueousHumor	Artery	Choroid	Cornea	Iris	Lamina	Lens	Ligament	OpticNerve	Pia	Retina	Sclera	Vein	VitreousHumor
AqueousHumor				✓	✓		✓	✓				✓		✓
Artery									✓					
Choroid					✓						✓	✓		✓
Cornea	✓											✓		
Iris	✓		✓					✓			✓		✓	
Lamina											✓	✓		
Lens	✓							✓						✓
Ligament	✓				✓	✓								✓
OpticNerve		✓								✓	✓	✓	✓	✓
Pia									✓			✓		
Retina				✓					✓			✓		✓
Sclera	✓		✓	✓	✓				✓	✓	✓			
Vein									✓					
VitreousHumor	✓		✓		✓		✓	✓			✓			

Table 2.1: Interfaces between solids

- Cornea
- Iris
- Ligament
- Lens
- AqueousHumor
- Sclera
- Choroid
- Retina
- Lamina
- Vein
- Artery
- Pia
- OpticNerve
- VitreousHumor

The final mesh is represented on figure 2.7, with on the one hand the total mesh and on the other hand only the mesh on the faces. On this mesh, obtained with default values set in the Salome pipeline, the mesh has 130 249 nodes and 871 499 elements.

2.5 Run the Salome pipeline

To run the script :

```
path/to/SALOME-9.6.0/salome [-t] eye.py
[args: [--hsize_eye=], [--hsize_lamina=], [--distance=], [--width=], [--hole=], [--shift=],
[--eye_length=], [corneal_thickness], [--mesh]]
```

The option `-t` runs Salome without the Graphical User Interface. The optional arguments are explained in this table :

Argument	Description	Type	Default
<code>--hsize_eye</code>	max size of the h of the computational mesh for the eye	float	1.0
<code>--hsize_lamina</code>	max size of the h of the computational mesh for the lamina	float	0.05
<code>--distance</code>	distance from retina/sclera [mm]	float	0.25
<code>--width</code>	lamina cribosa width [mm]	float	0.2
<code>--hole</code>	radius if the hole in lamina cribosa [mm]	float	0.2
<code>--shift</code>	shift hole from lamina cribosa center [mm]	float	0.3
<code>--eye_length</code>	length of main axis of the eye [mm]	float	26.1
<code>--corneal_thickness</code>	mean value of the corneal thickness [mm]	float	1.0

The option `--mesh` activates mesh generation.

To figure the weight of different nodes, this video made with Paraview shows the meshed geometry.

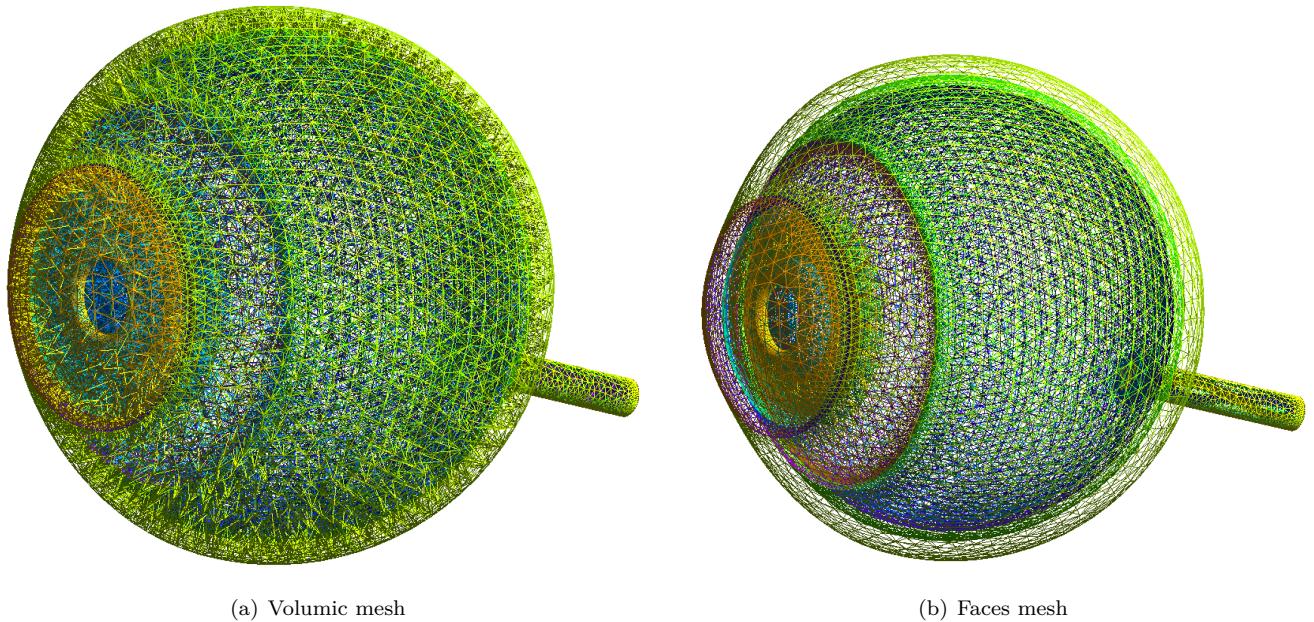


Figure 2.7: Meshed eye

2.6 Computational performances

In this section, we are interested in how the parameters `hsize_eye` and `hsizelamina` influence the time of execution and the size of the geometry obtained.

To do such measures, a little bash script is written. While it is executed, the Python script displays information in the standard output. With a redirection of this flow in a text file, we can look at it to get the information needed. For example in listing 2, we get the results for the mesh generation time when `hsize_eye` vary.

```

1 #! /bin/sh
2 DIR="/tmp/mesh"
3 mkdir $DIR
4 echo "h\tmesh" > values.csv
5 for h in 0.5 1 2 3 5 10
6 do
7     ./salome -t eye.py args:--hsize_eye=$h,--mesh > $DIR/output$h.txt
8     MESH_TIME=`grep "Mesh generation time" $DIR/output$h.txt | grep -v -E '^[[[:space:]]*#' | cut -d " " -f 5`
9     echo "$h\t$MESH_TIME" >> values.csv
10 done

```

Listing 2: Extract of the bash script

On table 2.2, we give the characteristics of the pipeline and the output mesh, for different values of `hsize_eye`, the quantity `hsizelamina` stays constant at its default value. We can see that the time to create the geometry doesn't depend on the parameters given, while mesh time does. This is logical because those parameters are not involved in the creation of geometry.

Now we are going to compare the time of execution with what Lorenzo obtained on the previous version of Salome [17, Table 6.2]. The values obtained are presented in table 2.3. As the geometry is quite different now, and as the computation has not been made on the same computer with a different version of SALOME, it is normal to find different values : we get more elements because we mesh parts of the eye that have not been meshed previously. Due to these new parts, the time to mesh is longer with v9. But we can notice that the geometry takes less time with v9.

The chapters A.3 and A.4 present tools provided by Feel++ to display the geometry with Paraview, or to work on many cores in parallel.

hszie_eye	Number of elements	Geometry time [s]	Mesh time [s]	Size of <code>med</code> file (o)
10	837 077	54.98	127.04	43 544
5	836 800	55.12	127.01	43 528
3	837 261	54.81	125.80	43 552
2	840 261	52.76	128.37	43 704
1	871 499	54.96	132.88	45 308
0.5	1 076 095	53.86	163.83	55 916

Table 2.2: Computational times and data for different `hszie_eye`

hszie_eye	hszie_lamina	Number of elements	Geometry time [s]	Mesh time [s]	Total sime [s]
0.05	1	871 499	60.20	149.23	210.99
0.025	0.5	1 224 313	60.50	209.19	271.25
0.1	2	807 396	59.53	142.75	203.80
1	1	831 061	59.22	147.36	208.14
0.05	0.25	2 816 511	60.28	572.80	634.51

Table 2.3: Computational times and data for different h size

3. Verification and validation of the geometry

In this chapter we will focus on some tests with Feel++, to figure that the geometry created in chapter 2 is correct. To begin with, we won't simulate something physical, but we will simply solve the Laplacian equation :

$$\begin{cases} \Delta u = f & \text{in } \Omega \\ u = g & \text{on } \partial\Omega \end{cases} \quad (3.0.1)$$

In a second time, in section 3.3, we will study a physical problem of heat transfer on the 3D geometry, and compare the results to a 2D case taken from the literature [10].

3.1 Tests with functions in the space

The theory of finite elements tells that if the solution u lives in the same space as the one used for the resolution, then the error on this solution is 0 (here, we will find the machine 0, which is around `1e-14`).

The application `feelpp_qs_laplacian_3d` allows to make those tests : we give to the application the desired solution, and it calculates the right member of the equation (f) and the boundary conditions (g), then calculates the norm of the error. The command to run it is :

```
mpirun -np 12 ./feelpp_qs_laplacian_3d --config-file exact.cfg --checker.solution "f(x,y,z):x:y:  
→z" --case.discretization P1 --ksp-monitor 1 --pc-type gamg --ksp-rtol 1e-13
```

we can set f , and the discretization space (P1, P2, P3).

The following table gathers the results for different inputs. Note that the higher we take the order of the discretization space, the more expensive calculation will be. To represent it, we also plot the time of execution¹. All those tests are made with the default values for the mesh, see section 2.5.

Solution	Space	$\ u - u_h\ _{L^2}$	$\ u - u_h\ _{H^1}$	time (s)
x	P1	4.723849e-14	2.238452e-13	12.22461
$x^2 + y^2$	P2	3.579756e-14	5.538407e-13	31.94711
$x^3 + y^2$	P3	6.243324e-14	2.256068e-12	134.3919
$\sin(xy^2) + z$	P1	93.44523	17.33398	10.45481
$\sin(xy^2) + z$	P2	47.70873	9.805593	31.57608
$\sin(xy^2) + z$	P3	33.87173	7.149038	138.2360

Table 3.1: Results

The three last lines show that with higher polynomial order, we get a better solution, even if the cost to obtain it is higher.

3.2 Convergence test

In this section, we focus on the convergence of the method on our geometry : we generate the mesh with different parameters and look at the error on the solution. To generate them, we use the same value for `hsize_eye` and `hsize_lamina`. For a small value of h , this leads to a gigantic mesh : it contains more than 16 million tetrahedrons ! The table 3.2 gather different characteristics of each mesh. Because of the complex geometry, we cannot have a uniform geometry. This is why when we take `h_size` twice smaller, the value of h_{\max} may not be divided by 2.

This first test of convergence is made with the solution $f(x, y, z) = \sin(xy^2) + z$, which can have great fluctuation. The plot of the errors is not shown here, but we can see that there is no difference in the order of convergence between the two orders. The order of convergence is about 1.5 in norm L^2 , and 1 for the norm H^1 , which doesn't correspond to the theoretical values. This can be due to the quadrature-order which is not high enough with this function f : the quadrature error dominates the global error.

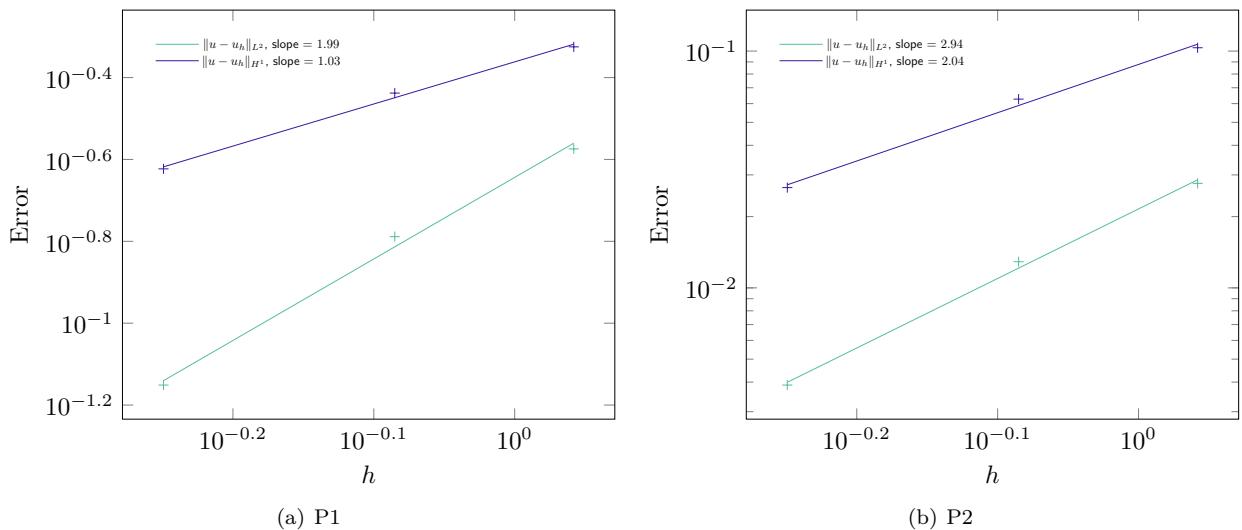
We are going to do the same test, with a simpler function $f(x, y, z) = \cos(\pi x) \sin(\pi y) \cos(\pi z)$, with an higher quadrature order (for that we had to add a new option to the `feelpp_qs_laplacian_3d` program : `--quad-order`).

¹this time corresponds to the whole Feel++ execution time, which includes the load of the mesh, the assembly of the matrix, and the calculation of errors

hsize	Number of elements	Numb. degree of freedom P1	Numb. degree of freedom P2	h_{\max}	h_{\min}	h_{avg}
1	705 012	124 221	960 207	2.61	$1.32 \cdot 10^{-3}$	0.29
0.5	879 394	156 297	1 204 622	1.42	$1.19 \cdot 10^{-3}$	0.37
0.25	2 458 290	435 787	3 368 297	1.10	$1.23 \cdot 10^{-3}$	0.36
0.2	5 014 580	879 444	6 831 402	0.82	$1.21 \cdot 10^{-3}$	0.29
0.125	16 781 882	2 909 576	22 742 810	0.56	$1.22 \cdot 10^{-3}$	0.20

Table 3.2: Description of the different meshes generated

The results of convergence is given in figure 3.1. For the plots, we add to remove the results obtained with the coarser meshes, which are not thin enough to give a proper order of convergence.

Figure 3.1: Convergence of the errors, with $f(x, y, z) = \cos(\pi x) \sin(\pi y) \cos(\pi z)$

3.3 Heat transfer

We are going to focus on the heat transfer in the eye. This work is a follow-up of what my colleagues Sarra and Hannane made during their 3rd semester project [9]. They studied heat transfer within the eye on a 2D geometry, here we will use our three-dimensional geometry.

3.3.1 Mathematic model

We focus on the heat flow, which is described by the following equation.

$$\rho_i C_{p,i} \frac{\partial T_i}{\partial t} = \nabla \cdot (k_i \nabla T_i) \quad (3.3.1)$$

where :

- i is the volume index (Cornea, VitreousHumor...),
- T_i [K] is the temperature in the volume i ,
- t [s] is the time. In a first approach, we will consider a stationnary state, so $\frac{\partial T_i}{\partial t} = 0$,
- k_i [$\text{W m}^{-1}\text{K}^{-1}$] is the thermal conductivity, ρ_i [kg m^{-3}] is the density and $C_{p,i}$ [$\text{J kg}^{-1}\text{K}^{-1}$] is the specific heat. All the values used for the differents volumes of the geometry are given in table 3.3, they are taken from [10].

Markers	k [W m ⁻¹ K ⁻¹]	ρ [kg m ⁻³]	C_p [J kg ⁻¹ K ⁻¹]
Cornea	0.58	1050	4178
Sclera	1.0042	1050	3180
AqueousHumor	0.28	996	3997
Lens	0.4	1000	3000
VitreousHumor	0.603	1100	4178
Iris, Lamina... [†]	1.0042	1050	3180

Table 3.3: Parameters for the simulation

Parameter	Value
h_{bl}	65 W m ⁻² K ⁻¹
h_{amb}	10 W m ⁻² K ⁻¹
T_{amb}	298 K
T_{bl}	310 K
E	40 W m ⁻²
σ	5.67×10^{-8} W m ⁻² K ⁻⁴
ε	0.975

Table 3.4: Parameters for boundary conditions

Remark 3.1 ([†]). All the other volumes of the eye are considered to form one only part for this simulation.

Here are the boundary condition set for the simulation :

- Neumann condition on the faces of the eye in contact with the external air :

$$-k \frac{\partial T}{\partial \mathbf{n}} = h_{\text{amb}}(T - T_{\text{amb}}) + \sigma \varepsilon (T^4 - T_{\text{amb}}^4) + E \quad (3.3.2)$$

This condition stands on the marker `Cornea_externalBC`. Notice that the boundary condition is not linear, because of radiative heat transfer.

- Robin condition on the external faces of the eye, but inside the human body :

$$-k \frac{\partial T}{\partial \mathbf{n}} = h_{\text{bl}}(T - T_{\text{bl}}) \quad (3.3.3)$$

This condition stands on the marker `Sclera_externalBC`.

The quantity T [K] is the temperature, h [W m⁻² K⁻¹] is the convection coefficient. The subscript *amb* stands for *ambiant*, and *bl* for *blood*. Finally, E [W m⁻²] is the evaporation rate, σ [W m⁻² K⁻⁴] is the Stefan–Boltzmann constant, and ε is the emissivity of the cornea (and has no dimension). Those conditions, and the values used, see table 3.4, are also taken from [10].

3.3.2 Static results

In the first part, we are going to see the results obtained for a static simulation : the term in front of the time derivation in the heat equation is null. The configuration files corresponding to this simulation can be found on the Github repository, the `cfg` file and the `json` file.

To run the simulation, we use this command (the number of cores used corresponds to the number of parts in which we partitioned the mesh).

```
mpirun -np 12 feelpp_toolbox_coefficientformpdes --config-file eye.cfg
```

The results obtained are shown in figure 3.2.

With Feel++, we can take measures on the result. Here we measured the minimal value, the maximum, and the mean of the temperature on the eye to compare it with what is given in [10, Section 3.3]. This is the left figure in figure 3.3.

On the stationary case, we find a minimal value on the whole eye of 307.37 K (34.22°C), while the article found 306.45 K (33.3°C). For the maximum, we get 309.92 K (36.77°C), in the article, it is 309.95 K (36.8°C).

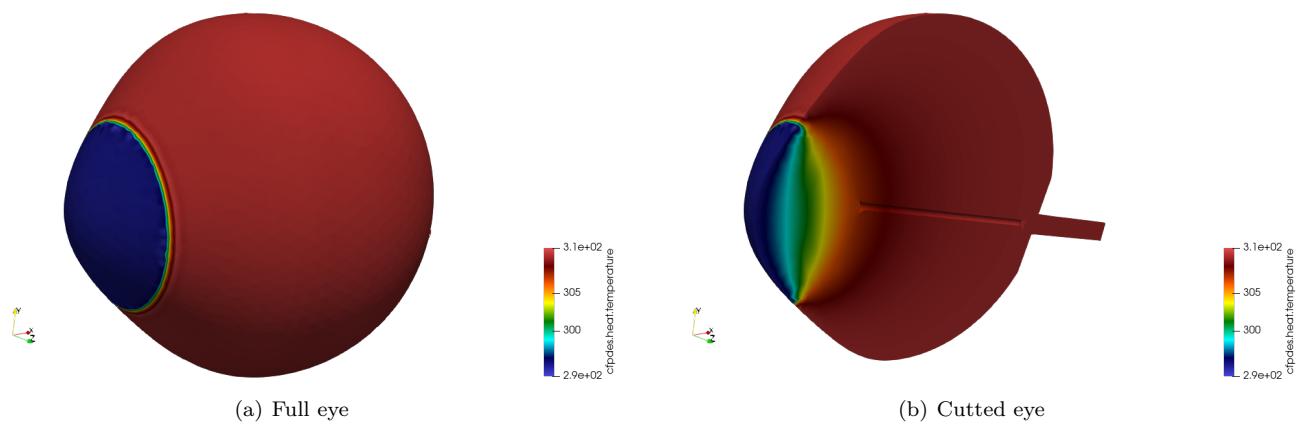


Figure 3.2: Static results

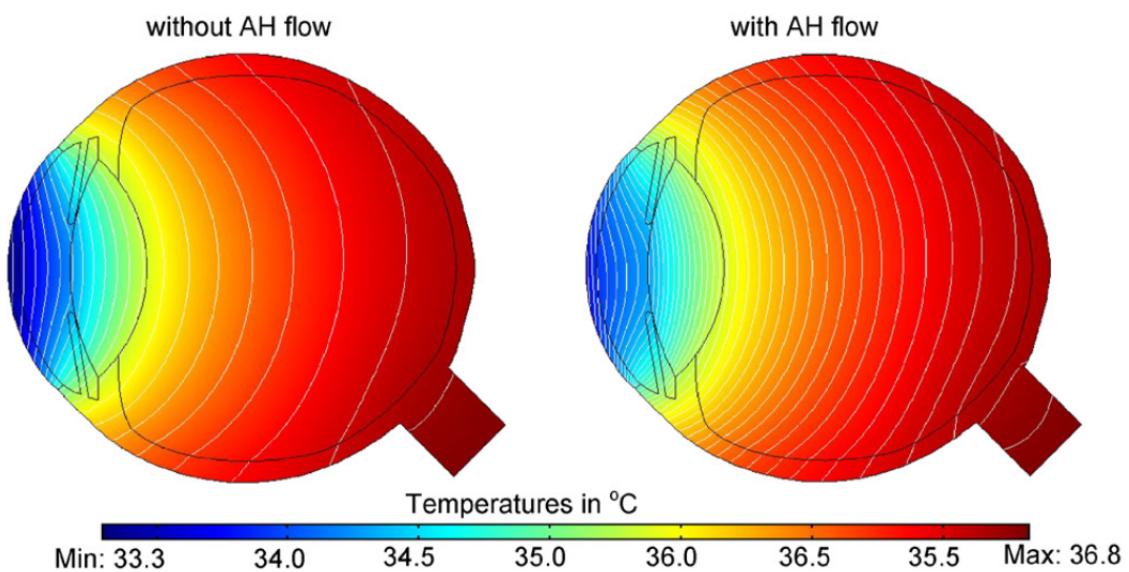


Figure 3.3: Static results

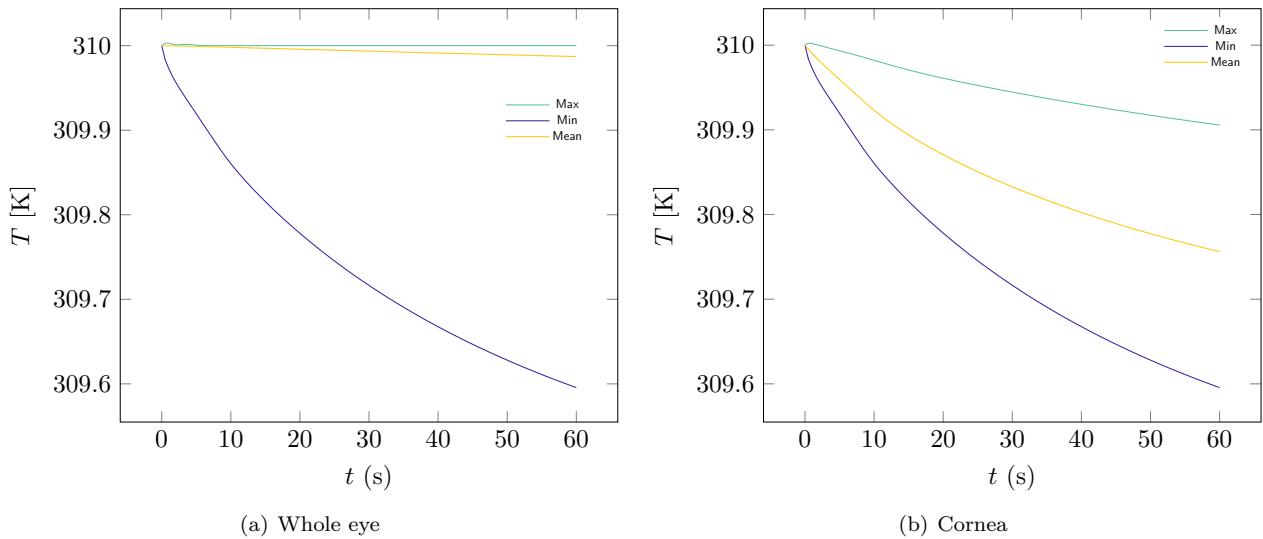


Figure 3.4: Results of the simulation over 1 minute

3.3.3 Time-dependant results

Now we are focusing on a time-dependant simulation. We have to set initial conditions in the configuration files for Feel++. On a first attempt, I gave $T^0 = T_{\text{amb}}$ in the cornea and $T^0 = T_{\text{bl}}$ in the other volumes of the eye, but the problem is that the initial temperature has a discontinuity on the border between the cornea and the rest of the eye. To avoid this problem, we can use a ramp on boundary conditions : at $t = 0$ we consider that the outside temperature is equal to T_{bl} , and progressively goes to T_{amb} , using a time *smoothstep*.

Definition 3.2 (Smoothstep – see this page). Let $a, b \in \mathbb{R}$, the function *smoothstep* is defined by :

$$\text{smoothstep}(t, a, b) = \begin{cases} a & \text{if } t \leq a \\ 3t^2 - 2t^3 & \text{if } a \leq t \leq b \\ b & \text{if } b \leq t \end{cases} \quad \text{for } t \in \mathbb{R} \quad (3.3.4)$$

The initial condition is that $T = T_{\text{bl}}$ on the whole eye, and the boundary condition is the same as given in 3.3.2, substituting T_{amb} by $T_{\text{trans}}(t)$ with

$$T_{\text{trans}}(t) = T_{\text{bl}} + \frac{\text{smoothstep}(t, 0, 10)}{10} (T_{\text{amb}} - T_{\text{bl}}) \quad (3.3.5)$$

The configuration files corresponding to this simulation can be found on Github, the cfg file and the json file.

A video of the evolution of the temperature in the eye from a different point of view is made. This video can be found on Git Lfs, on this link. This time, we can measure the evolution of the minimal and maximal temperature on the eye. We also made the same measures on a single marker : Cornea, which is the volume in contact with the ambient air. The evolution of the temperature (the min, the max, and the mean) is plotted on figure 3.4.

These two results give the conclusion that the generated geometry does not have a mesh defect. If this had been the case, we would have observed abnormal behavior in the results obtained.

4.

Physical modeling

This chapter presents the splitting operator involved in the 3D – 0D coupling, developed in [3, 17]. The example presented in the next section is taken from [2, section 2.3]. The Feel++ application implementing this coupling is the toolbox `feelpp_toolbox_hdg_coupledpoisson`.

As described in section 2.1, the lamina cribosa plays a critical role in the connection between the eye and the optic nerve.

4.1 Tissue perfusion

We model here the lamina cribosa (LC) as a porous material where blood vessels are viewed as pores in a solid matrix. We denote by $\Omega \subset \mathbb{R}^3$ the spatial domain of the LC, which is schematized as a cylinder with a hole at his center. This leads to the time-dependant problem :

$$\mathbf{j} + k_p \nabla p = 0 \quad \text{in } \Omega \times]0, T[\quad (4.1.1a)$$

$$\frac{\partial p}{\partial t} + \nabla \cdot \mathbf{j} = f \quad \text{in } \Omega \times]0, T[\quad (4.1.1b)$$

where p is the pressure, \mathbf{j} is the discharge velocity (blood perfusion velocity) and k_p is the permeability. We note that this is the Darcy problem introduced in equation (6.1.1), with $\underline{\mathcal{K}} = k_p I$ (identity matrix).

We couple the 3D-model (4.1.1) for the LC with a simplified 0D-model for the blood circulation in the posterior ciliary arteries of the lamina. The whole model is presented in figure 4.1.

We denote by $\boldsymbol{\Pi} = [\Pi_1, \Pi_2, \Pi_3]^T$ the vector of unknown pressures at the circuit nodes. The dynamic of the 0D circuite is described by :

$$\frac{d\boldsymbol{\Pi}}{dt} = \underline{\underline{A}}\boldsymbol{\Pi} + \mathbf{s} + \mathbf{b} \quad (4.1.1c)$$

where $\underline{\underline{A}}$ is a matrix representing the vascular resistances and compliances. For our model, we have :

$$\underline{\underline{A}} = \begin{bmatrix} -\frac{1}{C_1 R_{12}} & \frac{1}{C_1 R_{12}} & 0 \\ \frac{1}{C_1 R_{12}} & -\frac{1}{C_2} \left(\frac{1}{R_{12}} + \frac{1}{R_{23}} \right) & \frac{1}{C_2 R_{23}} \\ 0 & \frac{1}{C_3 R_{23}} & -\frac{1}{C_3} \left(\frac{1}{R_{23}} + \frac{1}{R_{\text{out}}} \right) \end{bmatrix} \quad (4.1.1d)$$

furthermore we have :

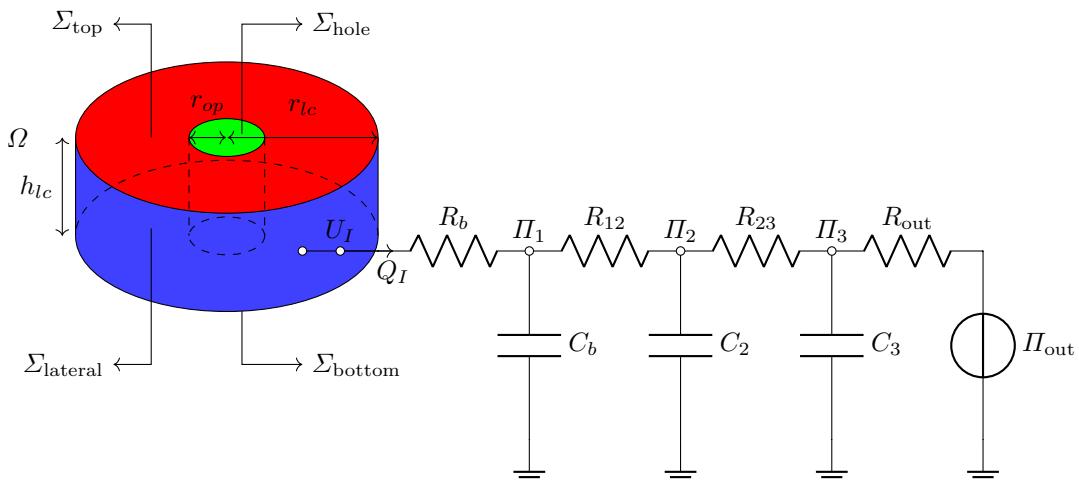


Figure 4.1: Coupled system of the LC

$$\mathbf{b} = \begin{bmatrix} Q_I \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{s} = \begin{bmatrix} 0 \\ 0 \\ \frac{\Pi_{\text{out}}}{R_{\text{out}}} \end{bmatrix} \quad (4.1.1e)$$

with $Q_I = \frac{U_I - \Pi_1}{R_1}$, where U_I is the unknown pressure on Σ_{lateral} , which is spatially uniform : Q_I and U_I are only time-dependant functions.

Remark 4.1. In other documents, the quantity U_I may be named p_I . To avoid confusion with the potential, we choose to use the notation U_I .

Remark 4.2. Equation (4.1.1c) can be retrieved using the electrical laws :

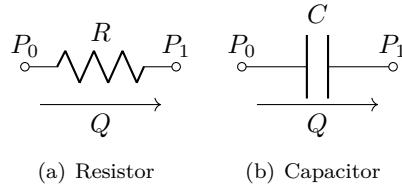


Figure 4.2: Electrical elements

- For a *linear resistor* with a resistance R (see figure 4.2(a)), the current Q through it is proportional to the voltage difference on the two poles : $Q = \frac{P_0 - P_1}{R}$.
- For a *linear capacitor*, we have this formula : $Q = C \frac{d(P_0 - P_1)}{dt}$.
- The conservative law, or Kirchhoff's law, tells that on a node of the circuit, the sum of inflow currents equals to the sum of outflow currents.

Using those laws on the three nodes Π_i of the circuit, we find the equation (4.1.1c) given above. •

Because of the coupling conditions on Σ_{lateral} , to ensure the continuity of mass and pressure, we have :

$$\int_{\Sigma_{\text{lateral}}} \hat{\mathbf{j}} \cdot \mathbf{n} = Q_I \quad p \text{ is constant on } \Sigma_{\text{lateral}} \quad (4.1.1f)$$

$$U_I = p \text{ on } \Sigma_{\text{lateral}} \quad (4.1.1g)$$

Finally, we add those boundary conditions :

$$p = p_{\text{hole}} \quad \text{on } \Sigma_{\text{hole}} \quad (4.1.1h)$$

$$\mathbf{j} \cdot \mathbf{n} = 0 \quad \text{on } \Sigma_{\text{top}} \cup \Sigma_{\text{bottom}} \quad (4.1.1i)$$

where p_{hole} is known. We also have those initial conditions :

$$p(\mathbf{x}, t = 0) = p_0(\mathbf{x}) \quad \text{in } \Omega \quad (4.1.1j)$$

$$\boldsymbol{\Pi}(t = 0) = \boldsymbol{\Pi}_0 \quad (4.1.1k)$$

In the following, the borders of the domain Ω will be named according to the condition on these borders (Γ_D for Dirichlet, Γ_N for Neumann and Γ_I for the IBC).

More generally, equation (4.1.1c) can be written $\frac{dy}{dt} = \underline{A}(\mathbf{y}, t)\mathbf{y} + \mathbf{r}(\mathbf{y}, t)$, with $\mathbf{r}(\mathbf{y}, t) = \mathbf{s}(\mathbf{y}, t) + \mathbf{b}(\mathbf{y}, t)$. The second term \mathbf{r} is composed by

- sources and sinks within the circuit, the quantity \mathbf{s}
- the contribution due to the coupling with the PDE domain, the quantity \mathbf{b} .

This 3D – 0D coupling introduces the instabilities of numerical methods and computational cost. We will use an operator splitting method. This is a time-splitting method that doesn't care about potential issues raised by spatial multiscale. The HDG formulation (see chapter 6) supports the integral boundary condition (4.1.1f) without any sub-iteration. The strategy that we will describe in chapter 7 allows us to compute at the same time the pressure on that boundary, obtaining a natural coupling between the 3D model and the 0D.

4.2 Feel++ application `feelpp_toolbox_hdg_coupledpoisson`

This section presents the way this two-steps algorithm is implemented in Feel++. The content of the code is taken from the file `coupling.hpp`. The main idea of the program is to start with objects given by the class `MixedPoisson` which solve a problem with the HDG method, then add to the matrix lines and columns corresponding to the terms due to coupling.

In the following, we will consider that there is only one interface condition in the model, as is the case on the model presented earlier. The reasoning would be the same with more conditions. We add to the finite elements matrix of the system equation (7.1.1) one line / column for the unknown U_I due to boundary condition (4.1.1g), and one line / column for the potential \hat{U}_I . In the following, to be consistent with the code, we will call Y this value.

In chapter 7, there is the details of the calculation of the variational problem. We simply insert it here :

$$(\underline{\mathcal{K}}^{-1} \mathbf{j}, \mathbf{v})_{\Omega} - (p, \nabla \cdot \mathbf{v})_{\Omega} + \langle \hat{p}, \mathbf{v} \cdot \mathbf{n} \rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} + \langle U_I, \mathbf{v} \cdot \mathbf{n} \rangle_{\Gamma_I} = 0 \quad (4.2.1)$$

$$(\nabla \cdot \mathbf{j}, w)_{\Omega} + (\partial_t p, w)_{\Omega} + \langle \tau p, w \rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} - \langle \tau \hat{p}, w \rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} - \langle \tau U_I, w \rangle_{\Gamma_I} = (f, w)_{\Omega} \quad (4.2.2)$$

$$\langle \mathbf{j} \cdot \mathbf{n}, \mu_1 \rangle_{\Gamma_{\text{int}} \cup \Gamma_N} + \langle \tau p, \mu_1 \rangle_{\Gamma_{\text{int}} \cup \Gamma_N} - \langle \tau \hat{p}, \mu_1 \rangle_{\Gamma_{\text{int}} \cup \Gamma_N} = \langle g_N, \mu_1 \rangle_{\Gamma_N} \quad (4.2.3)$$

$$\langle \mathbf{j} \cdot \mathbf{n}, \mu_2 \rangle_{\Gamma_I} + \langle \tau p, \mu_2 \rangle_{\Gamma_I} - \langle \tau U_I, \mu_2 \rangle_{\Gamma_I} - \frac{1}{|\Gamma_I|} \left\langle \frac{1}{R_b} U_I, \mu_2 \right\rangle_{\Gamma_I} + \frac{1}{|\Gamma_I|} \left\langle \frac{1}{R_b} Y, \mu_2 \right\rangle_{\Gamma_I} = 0 \quad (4.2.4)$$

$$\frac{1}{|\Gamma_I|} \left\langle C_b \frac{Y}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} - \underbrace{\left(\frac{1}{|\Gamma_I|} \left\langle \frac{1}{R_b} U_I, \mu_3 \right\rangle_{\Gamma_I} - \frac{1}{|\Gamma_I|} \left\langle \frac{1}{R_b} Y, \mu_3 \right\rangle_{\Gamma_I} \right)}_{\langle b, \mu_3 \rangle_{\Gamma_I} = \langle Q_I, \mu_3 \rangle_{\Gamma_I}} = \frac{1}{|\Gamma_I|} \left\langle C_b \frac{Y^{\text{old}}}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} \quad (4.2.5/A)$$

$$\frac{1}{|\Gamma_I|} \left\langle C_b \frac{Y}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} - \underbrace{\left(\langle \mathbf{j} \cdot \mathbf{n}, \mu_3 \rangle_{\Gamma_I} + \langle \tau p, \mu_3 \rangle_{\Gamma_I} - \langle \tau U_I, \mu_3 \rangle_{\Gamma_I} \right)}_{\langle b, \mu_3 \rangle_{\Gamma_I} = \langle Q_I, \mu_3 \rangle_{\Gamma_I}} = \frac{1}{|\Gamma_I|} \left\langle C_b \frac{Y^{\text{old}}}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} \quad (4.2.5/B)$$

We represented on figure 4.3 the finite elements matrix, with its different blocks. The blue-framed parts of the matrix correspond to the added lines / columns due to the coupling. From equation (4.1.1f), we have $\int_{\Gamma_I} \hat{\mathbf{j}} \cdot \mathbf{n} = Q_I = \frac{\hat{U}_I - \hat{Y}}{R_b}$ which can be rewritten

$$\int_{\Gamma_I} \hat{\mathbf{j}} \cdot \mathbf{n} - \frac{\hat{U}_I - \hat{Y}}{R_b} = 0 \quad (4.2.6)$$

That is why in the matrix we have the red terms added in the line corresponding to \hat{U}_I . A more detailed matrix for this problem is given in figure 7.1.

\mathbf{j}	p	\hat{p}	\hat{U}_I	\hat{Y}	
\mathbf{j}	x		
p	x		
\hat{p}	0		
\hat{U}_I	x	x	0	$\mathbf{x} - \frac{1}{R_b}$	$\frac{1}{R_b}$
\hat{Y}	0	0	0	0	$\frac{y^n}{\Delta t} + b$

Figure 4.3: EF matrix for the step 1 of the coupled problem

The term $\frac{y^n}{\Delta t}$ depends on the time-discretization chosen. More details about the time discretization are given below. To get the right term in Feel++, we use the function `polyDeriv`, which returns the right-hand side of the time scheme.

With Feel++, the EF matrix is represented by a `blockform2` named `bbf` the i -th « line » and the j -th « column » of the EF matrix can be reached with the command `bbf(i_c, j_c)`. More precisely, for the lines / columns added because of the coupling can be reached with the command `bbd(3_c, 3_c, i, j)` with i and j the index of the line / column (from 0). This is the same for the right-hand side, which is a `blockform1`.

Here are the indices of the `blockform2`. We set $j=i+1$.

$$\begin{bmatrix} 0_c,0_c & 0_c,1_c & 0_c,2_c & 0_c,3_c,0,i & 0_c,3_c,0,j \\ 1_c,0_c & 1_c,1_c & 1_c,2_c & 1_c,3_c,1,i & 1_c,3_c,1,j \\ 2_c,0_c & 2_c,1_c & 2_c,2_c & 2_c,3_c,2,i & 2_c,3_c,2,j \\ 3_c,0_c,i,0 & 3_c,1_c,i,1 & 3_c,2_c,i,2 & 3_c,3_c,i,i & 3_c,3_c,i,j \\ 3_c,0_c,j,0 & 3_c,1_c,j,1 & 3_c,2_c,j,2 & 3_c,3_c,j,i & 3_c,3_c,j,j \end{bmatrix} \quad (4.2.7)$$

The application generated is `feelpp_toolbox_hdg_coupledpoisson`¹. When I first tried to use this application on the test cases given in the repository, the results were erroneous. To seek what is incorrect in the code, I added functionality to the program : an option to decouple the PDE and the ODE on the first step of the algorithm. See section 5.1 for more details.

¹See documentation http://docs.feelpp.org/toolboxes/0.108/hdg/hdg_coupledpoisson.html

5. Implementation aspects of the model

5.1 Tests with the application `feelpp_toolbox_hdg_coupledpoisson`

In the previous section, we described the system for step 1 (see equation (7.1.1)) with and without coupling. The purpose of this is to test the application with an analytical case.

To ensure that, we added some options to the application coded in the file `coupling.hpp` :

Name	Description	Default value	Possible values
<code>coupling.mode</code>	type of coupling used	1	see below
<code>coupling.QI</code>	analytical value of Q_I	<i>1-linear</i>	expression
<code>coupling.Pi1</code>	analytical solution of the ODE (7.1.1c) Π_1	<i>1-linear</i>	expression

The default values of the options `coupling.QI` and `coupling.Pi1` are the analytical values for the test case *1-linear*. The type of coupling are described in this table :

Mode	Description
0	The decoupled system is solved
1 (default)	The coupled system is solved, using $Q_I = \int_{\Gamma_I} \hat{\mathbf{j}} \cdot \mathbf{n}$
2	The coupled system is solved, using $Q_I = \frac{U_I - \Pi_1}{R_b}$

5.2 Functional Mock-up Interface

The *Functional Mock-up Interface* (or *FMI*) defines a standardized interface to be used in computer simulations. Many tools can be used to develop FMI, such as the open-source OpenModelica, or the proprietary Dymola. During the internship, I used Dymola which we bought a license for Cemosis (especially for the ibat project). The next section presents how Dymola works.

With such software, we create from the FMI a software called *Functional Mock-up Unit* (or *FMU*). With Feel++, it is possible to generate FMU from FMI and simulate the model. With CMake, we can generate the `fmu` file from the model `mo`, adding this line in `CMakeLists.txt`.

```
feelpp_add_fmu( test3d0d CLASS test3d0d SRCS test3d0d.mo )
```

1. Name of the exporter FMU (here `test3d0d`)
2. Name of the class in the `mo` files that we want to export (here `test3d0d`)
3. Files with the code for the model (here `test3d0d.mo`, we have only one file)

We run the application `feelpp_fmi_fmu` with the following `cfg` file to export at each time step the value of Π_1 .

```
1 [fmu]
2 filename=$cfgdir/test3d0d.fmu
3 exported-variables=Pi_1.phi
4 time-initial=0
5 solver.time-step=0.1
6 solver.rtol=1e-6
```

5.3 Model 0D with dymola

To handle Modelica and Dymola, we implement the 0D part of the model described in section 4.1 : the coupling term is represented here with a constant voltage source U_I . The scheme of the model is presented in figure 5.1(a).

To measure the quantities Π_1 , Π_2 and Π_3 with Dymola, we can use a `PotentialSensor`, on the corresponding nodes. The circuit implemented in Dymola is shown on figure 5.1(b). We can notice that for the signal Π_{out} which depends on time, we have to create block `realExpression` to set the value of the signal.

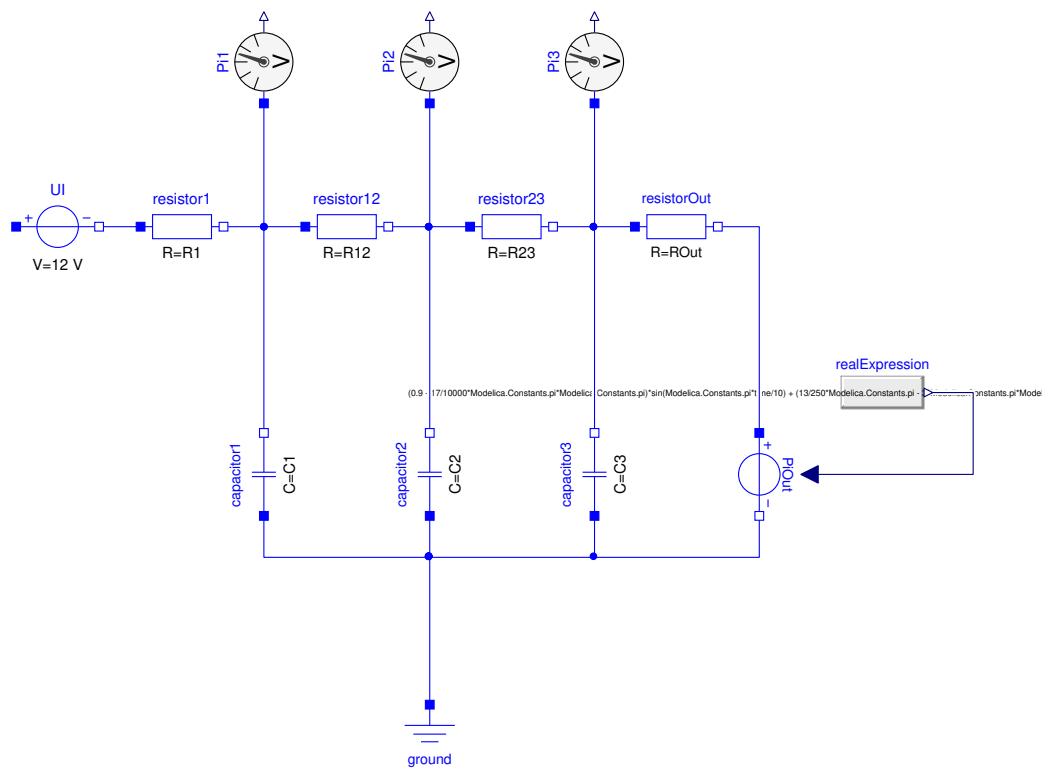
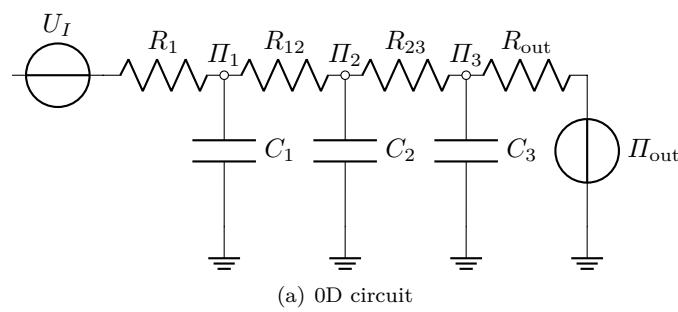


Figure 5.1: Circuit for Dymola

There are two ways of implementing a model with dymola. The first one is to use the graphical user interface : we drag and drop blocks from the left menu to create objects, and we join them with « wires » (as we can see on figure 5.1(b)). A code is generated in a `mo` file. Equations come out from all those blocks and connexions, and those equations are solved when we simulate the model.

The second way to implement a model is to write directly the `mo` file. For complex models, that can be difficult... Here is an example of two resistors connected by a wire.

```
model TwoResistors
    Modelica.Electrical.Analog.Basic.Resistor resistor1(R=R1) "resistor"
        annotation (Placement(transformation(extent={{-70,30},{-50,50}}))) ;
    Modelica.Electrical.Analog.Basic.Resistor resistor12(R=R12)
        annotation (Placement(transformation(extent={{-30,30},{-10,50}}))) ;
equation
    connect(resistor1.n, resistor12.p) ;
end TwoResistors ;
```

The `n` and `p` elements of `Resistor` correspond to the negative and positive poles of the electrical component.

Once the model is set, we go in the *simulation* tab in Dymola to run the simulation. Many parameters can be set, such as the time of simulation or the number of time steps. After the simulation is run, we can plot many variables : the attributes of the different elements, such as the potential at each pole (see figure 5.2(a) for the resistor R_{12}) ; or the quantities measured by sensors. On figure 5.2(b), the values of Π_i (for $i \in \{1, 2, 3\}$) is plotted among time, for $t \in [0, 10]$. On this figure, we can figure that the node closer to the source Π_{out} has an higher potential.

One interesting feature of the Modelica language is that we can write documentation of the case directly in the `mo` file, using HTML language. Furthermore, we can define parameters for the model. When the documentation is displayed with Dymola, all those parameters are automatically included.

```
parameter Modelica.SIunits.Resistance R1=1e3 "Resistor 1";
parameter Modelica.SIunits.Capacitance C1=1e-3 "Capacitance 1";
```

5.4 Linear case

This example has been developped to check the results of the toolbox `hdg_coupled_poisson`. The files describing it can be found on Github in `toolboxes/hdg/coupledpoisson/test-linear` directory¹.

This model is very simple : we take a parallelepipedon for the 3D domain and a simple circuit for the 0D model. The coupled model is shown on figure 5.3.

We set :

- $\Omega = \{x = (x, y, z) \in \mathbb{R}^3 \mid x \in [0, H] \text{ and } y, z \in [0, L],\}$
- $\Gamma_D = \Omega \cap \{x = 0\}$
- $\Gamma_I = \Omega \cap \{x = H\}$
- $\Gamma_N = \Omega \cap (\{y = 0\} \cup \{y = L\} \cup \{z = 0\} \cup \{z = L\})$

We set $p(x, t) = \alpha + \beta xt$ for $x = (x, y, z) \in \Omega$ and $t \geq 0$, and $\underline{\mathcal{K}} = kI$ (the identity matrix). So time dependant Darcy equation (see equation (6.1.1))

$$\frac{1}{M} \frac{\partial p}{\partial t} + \nabla \cdot \mathbf{j} = f \quad (5.4.1a)$$

$$\mathbf{j} + k \nabla p = \mathbf{0} \quad (5.4.1b)$$

gives that :

$$\mathbf{j} = -k \begin{bmatrix} \beta t \\ 0 \\ 0 \end{bmatrix} \quad f = \beta x \quad (5.4.2)$$

Let's calculate the boundary conditions (see equation (6.1.2)) :

¹It is possible that this case is changed or deleted in the future, so the links points to the state of the repository at commit `b2adc`.

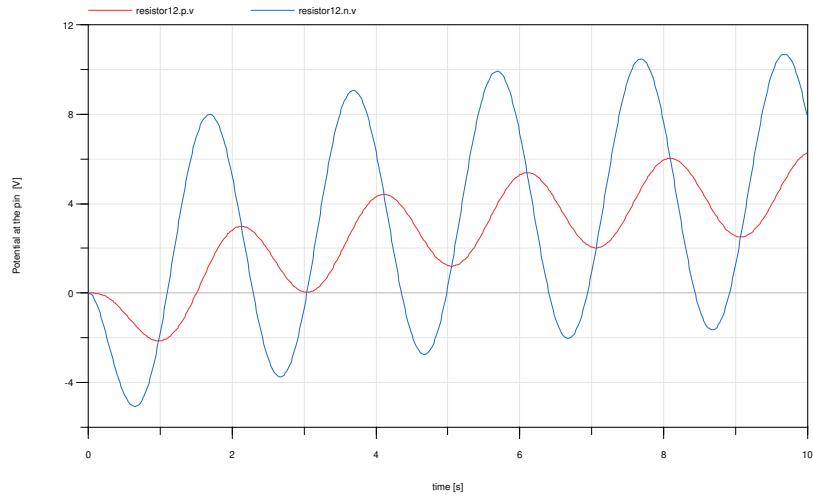
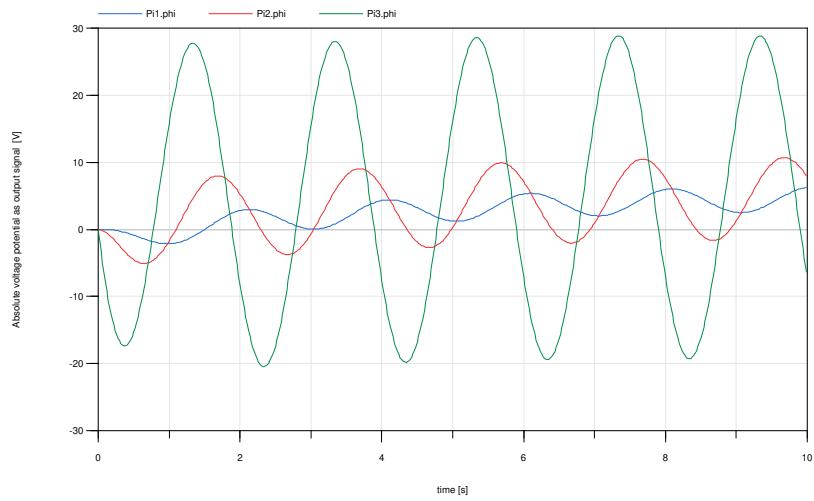
(a) Poles of the resistor R_{12} (b) Potential measured on nodes Π_1 , Π_2 and Π_3

Figure 5.2: Dymola results

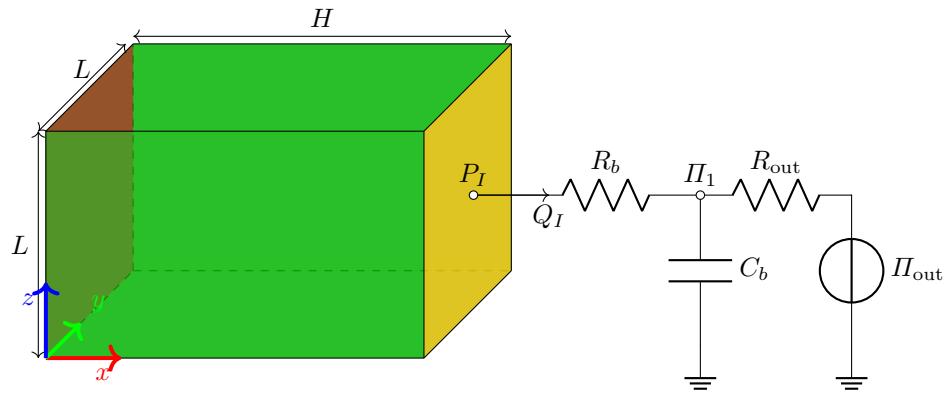


Figure 5.3: Simple 3D - 0D model

- On Γ_N , $\mathbf{j} \cdot \mathbf{n} = 0$, because $\mathbf{n} = [0, 0, -1]^T, [0, 1, 0]^T, [0, 0, 1]^T, [0, -1, 0]^T$ depending on the face of Γ_N ,
- On Γ_G , $p = \alpha$,

- On Γ_I , $\int_{\Gamma_I} \mathbf{j} \cdot \mathbf{n} = \int_{\Gamma_I} \begin{bmatrix} -\beta t \\ 0 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = -L^2 k \beta t =: Q_I$; and $p|_{\Gamma_I} = \alpha + \beta H t =: U_I$

Four our test case, we take $R_b = 1 \Omega$ and $C_b = 1 F$, to have simpler formulas.

As $Q_I(t) = \frac{U_I - \Pi_1}{R_b}$, we get that $\Pi_1 = P_I - Q_I = \alpha + \beta(H + R_b L^2 k)t$. Using Kirchhoff laws on node Π_1 , we find that

$$\Pi_{\text{out}} = \alpha + \beta [Ht + L^2 kt(R_b - R_{\text{out}}) - C_b R_{\text{out}}(H + R_b L^2 k)]$$

With these beautiful formulas, we can check that the result of the toolbox corresponds to the theory.

NB : To run 0D model with Feel++, we can use FMI. More détails are given on section 5.2.

On figure 5.4 is represented the potential obtained by the toolbox in front of the theoretical value p . the screen shot has been taken after 20 time steps. We see that there is a manifest difference between the two of them.

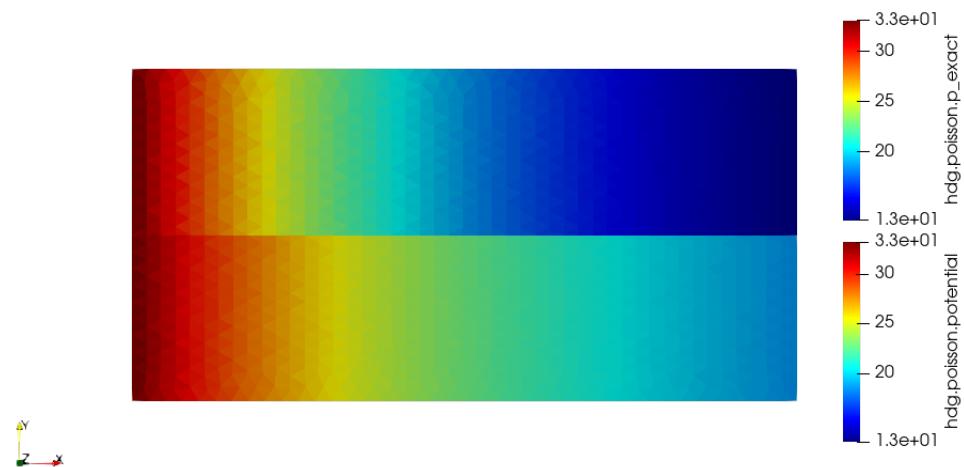


Figure 5.4: Comparaison of the potential

So something is uncorrect in the code. We tried to fix it during the internship (see branch `test-coupling` in the repository, or the issue 1621), but no correction ahs yet been made.

Part II

Methods and discretization

In this second part, we will study the methods used to solve the models. A first chapter is dedicated to the *Hybridizable Discontinuous Galerkin* (chapter 6), and a second (chapter 7) to an algorithm dealing with the coupling described in chapter 4 : the time-splitting algorithm

6.

Hybridizable Discontinuous Galerkin

6.1 Darcy equations with IBC

The Darcy equations describe the behavior of a porous medium.

Let Ω an open bounded set of \mathbb{R}^d , with $d \in \{2, 3\}$. We denote by Γ the boundary of Ω , partitioned into three disjoint subsets : Γ_D , Γ_N and Γ_I . The problem to solve is : find $\mathbf{j} \in H(\text{div}, \Omega)$ and $p \in L^2(\Omega)$ such that

$$\mathbf{j} + \underline{\mathcal{K}} \cdot \nabla p = 0 \quad \text{in } \Omega \quad (6.1.1a)$$

$$\nabla \cdot \mathbf{j} = f \quad \text{in } \Omega \quad (6.1.1b)$$

with those boundary conditions :

$$p = g_D \quad \text{on } \Gamma_D \quad (6.1.2a)$$

$$\mathbf{j} \cdot \mathbf{n} = g_N \quad \text{on } \Gamma_N \quad (6.1.2b)$$

$$\int_{\Gamma_I} \mathbf{j} \cdot \mathbf{n} = I_{\text{target}} \quad \text{on } \Gamma_I \quad (6.1.2c)$$

$$p(\mathbf{x}, t) = p(t) \quad \text{on } \Gamma_I \quad (6.1.2d)$$

where I_{target} is a given constant. The solution p is also constant on Γ_I , this property is a consequence of equation (6.1.2c). Furthermore, $f \in L^2(\Omega)$ and $\underline{\mathcal{K}} \in (L^\infty(\Omega))^{n \times n}$ is a symmetric matrix, uniformly positive defined over Ω . It represents the permeability tensor.

The condition 6.1.2b is called *integral boundary condition* and ensures the coupling between models. See figure 6.1(a) for an example of geometry for this problem.

Notation 6.1. Let $p, q \in L^2(D)$, we write $(p, q)_D = \int_D pq$ if $D \subset \mathbb{R}^d$, and $\langle p, q \rangle_D = \int_D pq$ if $D \subset \mathbb{R}^{d-1}$.

Definition 6.2. We define the following spaces :

- $H^{1/2}(\Gamma) = \{\varphi \in L^2(\Gamma) \mid \exists \varphi \in H^1(\Omega), u|_\Gamma = \varphi\}$,
- $H_{00}^{1/2}(\Gamma_N) = \{\varphi \in H^{1/2}(\Gamma) \mid \varphi = 0 \text{ on } \Gamma_D \cup \Gamma_I\}$
- $H(\text{div}, \Omega) = \{\mathbf{v} \in L^2(\Omega) \mid \nabla \cdot \mathbf{v} \in L^2(\Omega)\}$

Let $\bar{\varphi} \in H^{1/2}(\Omega)$ such as $\bar{\varphi}|_{\Gamma_D} = 0$ and $\bar{\varphi}|_{\Gamma_I} = 1$.

Let's find the variational formulation of the problem (6.1.1 – 6.1.2).

we multiply equation (6.1.1a) by $\underline{\mathcal{K}}^{-1}$. Let's take a test function $\mathbf{v} \in H(\text{div}, \Omega)$, then we have :

$$\begin{aligned} \int_{\Omega} \underline{\mathcal{K}}^{-1} \mathbf{j} \mathbf{v} + \int_{\Omega} \nabla p \mathbf{v} &= 0 \\ \int_{\Omega} \underline{\mathcal{K}}^{-1} \mathbf{j} \mathbf{v} - \int_{\Omega} p \nabla \cdot \mathbf{v} + \int_{\Gamma} \hat{p} \mathbf{v} \cdot \mathbf{n} &= 0 \end{aligned}$$

after a partial integration. We do the same calculation on equations (6.1.1b) and (6.1.2a). On equation (6.1.2c), we have for a test function $\mu \in \text{span} \langle \bar{\varphi} \rangle \oplus H_{00}^{1/2}(\Gamma_N)$:

$$\begin{aligned} \int_{\Gamma} \mathbf{j} \cdot \mathbf{n} \mu &= \int_{\Gamma_D} \mathbf{j} \cdot \mathbf{n} \mu + \int_{\Gamma_N} \mathbf{j} \cdot \mathbf{n} \mu + \int_{\Gamma_I} \mathbf{j} \cdot \mathbf{n} \mu \\ &= 0 + \langle g_N, \mu \rangle_{\Gamma_N} + I_{\text{target}} |\Gamma_I|^{-1} \langle \mu, 1 \rangle_{\Gamma_I} \end{aligned}$$

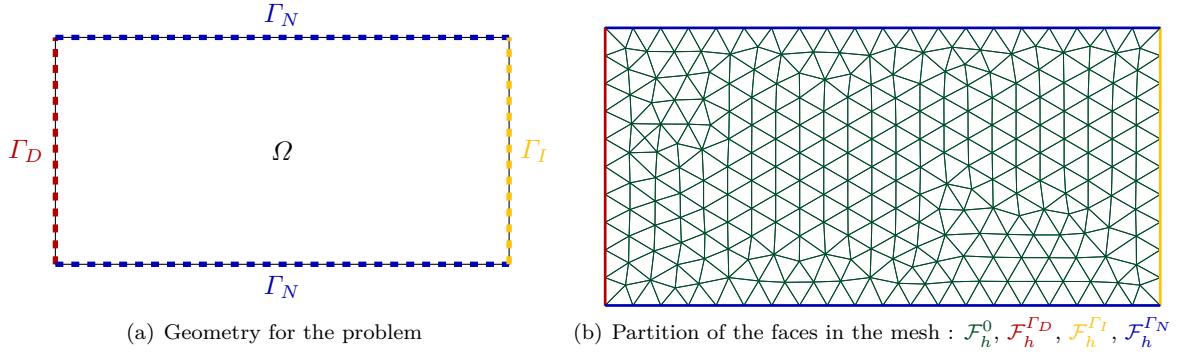


Figure 6.1: Example of geometry and mesh for the problem (6.1.1 – 6.1.2)

because of $\mu|_{\Gamma_D} = 0$, equation (6.1.2c), and μ is constant on Γ_I .

It leads to this variational problem : find $\mathbf{j} \in H(\text{div}, \Omega), p \in L^2(\Omega)$ and $\widehat{p} \in \text{span} \langle \bar{\varphi} \rangle \oplus H_{00}^{1/2}(\Gamma_N)$, such that for all $\mathbf{v} \in H(\text{div}, \Omega), w \in L^2(\Omega)$ and $\mu \in \text{span} \langle \bar{\varphi} \rangle \oplus H_{00}^{1/2}(\Gamma_N)$, we have

$$(\underline{\mathcal{K}}^{-1} \mathbf{j}, \mathbf{v})_\Omega - (p, \nabla \cdot \mathbf{v})_\Omega + \langle \widehat{p}, \mathbf{v} \cdot \mathbf{n} \rangle_{\Gamma} = 0 \quad (6.1.3a)$$

$$(\nabla \cdot \mathbf{j}, w)_\Omega = (f, w)_\Omega \quad (6.1.3b)$$

$$\langle \mathbf{j} \cdot \mathbf{n}, \mu \rangle_{\Gamma_N \cup \Gamma_I} = \langle g_N, \mu \rangle_{\Gamma_N} + I_{\text{target}} |\Gamma_I|^{-1} \langle \mu, 1 \rangle_{\Gamma_I} \quad (6.1.3c)$$

$$\langle \widehat{p}, \mu \rangle_{\Gamma_D} = \langle g_D, \mu \rangle_{\Gamma_D} \quad (6.1.3d)$$

This formulation leads to the following theorem :

Theorem 6.3. Let $f \in L^2(\Omega), g_D \in H^{1/2}(\Gamma_D), g_N \in L^2(\Gamma_N)$. The variational problem (6.1.3) has a single solution $(\mathbf{j}, p, \widehat{p}) \in H(\text{div}, \Omega) \times L^2(\Omega) \times \text{span} \langle \bar{\varphi} \rangle \oplus H_{00}^{1/2}(\Gamma_N)$ such as :

$$\begin{aligned} -\nabla \cdot (\underline{\mathcal{K}} \nabla p) &= f && \text{in } \Omega \\ p &= g_D && \text{on } \Gamma_D \\ -\underline{\mathcal{K}} \nabla p \cdot \mathbf{n} &= g_N && \text{on } \Gamma_N \end{aligned}$$

with p constant on Γ_I . Moreover, we have $\int_{\Gamma_I} -\underline{\mathcal{K}} \nabla p \cdot \mathbf{n} = I_{\text{target}}$.

The proof of this theorem can be found in [2, Theorem 3.2].

6.2 HDG formulation

The *Hybridizable Discontinuous Galerkin* (HDG) method is a finite element (FE) method where the space used for approximating leads to solutions that are not continuous, as opposed to usual FE methods. This leads to a higher number of degrees of freedom, but as we will see after, this problem will be handled by a special manipulation, the *static condensation*. The continuity of the solution on the boundary of elements is weakly imposed.

The content of this section is mostly taken from [2, Section 4].

Definition 6.4. If F is the facet at the intersection of two adjecents elements K_1 and K_2 , we define the *jump* of the normal trace of the vector-valued function $\mathbf{q} \in H(\text{div}, K)$ across F by :

$$[\![\mathbf{q}]\!]_F = \mathbf{q}^{K_1} \cdot \mathbf{n}_{\partial K_1}|_F + \mathbf{q}^{K_2} \cdot \mathbf{n}_{\partial K_2}|_F \quad (6.2.1)$$

To apply the HDG method, we need to get a triangulation of the domain Ω . We call \mathcal{T}_h this triangulation, where h is the characteristic size, which is the maximum of the diameters of the elements $K \in \mathcal{T}_h$.

In the following, we will call *face* of an element the $(d-1)$ -dimensionnal element of the triangulation (so if $d=2$, we call *face* what is usually called *edge*).

Definition 6.5. We call *skeleton* of the triangulation \mathcal{T}_h the set of all faces of the elements. The skeleton of \mathcal{T}_h is noted \mathcal{F}_h . This set is partitionned in sub-sets :



Figure 6.2: Fuctions over the finite elements spaces

- $\mathcal{F}_h^{\Gamma_D}$ is composed of all faces that belongs to Γ_D ,
- $\mathcal{F}_h^{\Gamma_I}$ is composed of all faces that belongs to Γ_I ,
- $\mathcal{F}_h^{\Gamma_N}$ is composed of all faces that belongs to Γ_N ,
- \mathcal{F}_h^0 contains the remaining faces.

The figure 6.1(b) shows an example of such separation.

6.3 Discrete variationnal formulation

Definition 6.6. We introduce these finite element spaces :

- $\mathbf{V}_h = \prod_{K \in \mathcal{T}_h} \mathbf{V}(K)$,
- $W_h = \prod_{K \in \mathcal{T}_h} W(K)$,
- $\widetilde{M}_h = \left\{ \mu \in L^2(\mathcal{F}_h) \mid \mu|_F \in \mathbb{P}_k(F) \quad \forall F \in \mathcal{F}_h^0 \cup \mathcal{F}_h^{\Gamma_N}, \mu|_{\Gamma_D \cup \Gamma_I} = 0 \right\}$,
- $M_h^* = \left\{ \mu \in L^2(\mathcal{F}_h) \mid \mu|_{\Gamma_I} \in \mathbb{R}, \mu|_{\mathcal{F}_h \setminus \Gamma_I} = 0 \right\}$ (we have $\dim M_h^* = 1$),
- $M_h = M_h^* \oplus \widetilde{M}_h^1$.

with $\mathbf{V}_k(K) = (\mathbb{P}_k(K))^d$ et $W_k(K) = \mathbb{P}_k(K)$

We can notice that from these definitions, the functions in \mathbf{V}_h and W_h are in general discontinuous over faces of elements as shown on figure 6.2(a). As well as functions in M_h : they are single-valued over faces of elements, but they are discontinuous on the vertices of the skeleton (see figure 6.2(b)). These functions are « connectors » between elements.

Definition 6.7. We define the *numerical normal flux* on ∂K :

$$\hat{\mathbf{j}}_K^{\partial K} \cdot \mathbf{n}_{\partial K} = \mathbf{j}_h^K|_{\partial K} \cdot \mathbf{n}_{\partial K} + \tau_{\partial K} (p_h^K|_{\partial K} - \hat{p}_h|_{\partial K}) \quad (6.3.1)$$

$\tau_{\partial K} \geq 0$ is a stabilisation parameter, that can depend on the face $F \in \partial K$.

The discrete formulation of the problem (6.1.3) is : find $\mathbf{j}_h \in \mathbf{V}_h$, $p_h \in W_h$ and $\hat{p} \in M_h$ such that $\forall \mathbf{v}_h \in \mathbf{V}_h, \forall w_h \in W_h, \forall \mu_h \in M_h$:

$$\sum_{K \in \mathcal{T}_h} \left[\left(\underline{\mathcal{K}}^{-1} \mathbf{j}_h^K, \mathbf{v}_h^K \right)_K - (p_h^K, \nabla \cdot \mathbf{v}_h^K)_K + \langle \hat{p}_h^{\partial K}, \mathbf{v}_h^K \cdot \mathbf{n}_{\partial K} \rangle_{\partial K} \right] = 0 \quad (6.3.2a)$$

$$\sum_{K \in \mathcal{T}_h} \left[- \left(\mathbf{j}_h^K, \nabla w_h^K \right)_K + \langle \hat{\mathbf{j}}_h^{\partial K} \cdot \mathbf{n}_{\partial K}, w_h^K \rangle_{\partial K} \right] = \sum_{K \in \mathcal{T}_h} (f, w_h^K)_K \quad (6.3.2b)$$

$$\sum_{K \in \mathcal{T}_h} \langle \hat{\mathbf{j}}_h^{\partial K} \cdot \mathbf{n}_{\partial K}, \mu_h \rangle_{\partial K} = \langle g_N, \mu_h \rangle_{\Gamma_N} + I_{target} |\Gamma_I|^{-1} \langle \mu_h, 1 \rangle_{\Gamma_I} \quad (6.3.2c)$$

¹the direct sum can be easily checked : if $\mu \in \widetilde{M}_h \cap M_h^*$, then on the one hand $\mu = 0$ on Γ_I , and on the other hand $\mu = 0$ on $\mathcal{F}_h \setminus \Gamma_I$, so $\mu = 0$.

The quantities \mathbf{j}_h and p_h are the approximations of \mathbf{j} and p inside elements $K \in \mathcal{T}_h$, and \widehat{p}_h is the approximation of the trace of \widehat{p}_h on the faces of \mathcal{F}_h .

As we will see in section 6.4 the discrete equations stand inside each $K \in \mathcal{T}_h$ and can be solved on each K to eliminate \mathbf{j}_h^K and p_h^K in favor of $\widehat{p}_h^{\partial K}$. This is *static condensation*. Combinant this procedure to the definition of the numerical normal flux $\widehat{\mathbf{j}}_h^{\partial K}$, we can express $\widehat{\mathbf{j}}_h^{\partial K}$ as a function of $\widehat{p}_h^{\partial K}$ only.

From the remaining equation (equation (6.3.2c)), and using properties of μ , we have :

1. If $F \in \partial K \in \mathcal{F}_h^0$, then $\langle \widehat{\mathbf{j}}^{\partial K} \cdot \mathbf{n}, \mu_h \rangle_{F_{\text{left}}} = \langle \widehat{\mathbf{j}}^{\partial K} \cdot \mathbf{n}, \mu_h \rangle_{F_{\text{right}}}$, so

$$\langle [\widehat{\mathbf{j}}_h], \mu_h \rangle_F = 0 \quad \forall F \in \mathcal{F}_h^0, \forall \mu \in M_h \quad (6.3.3)$$

This equation ensures weakly the continuity through the inner faces.

2. Si $F \in \partial K \in \mathcal{F}_h^0$,

$$\langle \widehat{\mathbf{j}}_h^{\partial K} \cdot \mathbf{n}_{\partial K}, \mu_h \rangle_F = \langle g_N, \mu \rangle_F \quad (6.3.4)$$

weakly ensures Neumann conditions.

3. It remains one equation

$$\langle \widehat{\mathbf{j}}_h^{\partial K} \cdot \mathbf{n}_{\partial K}, \mu_h \rangle_F = I_{\text{target}} |\Gamma_I|^{-1} \langle \mu_h, 1 \rangle_{\Gamma_I} \quad (6.3.5)$$

weakly ensures the IBC condition.

Theorem 6.8. *The discrete problem (6.3.1 – 6.3.2) has a unique solution.*

The proof is given in [2, Theorem 4.2].

6.4 Static condensation

As said earlier, we will study how the system (6.3.1 – 6.3.2) can be *statically condensed*, which is recast in terms of a global linear system where only the trace of the solution on the boundaries of the elements of the mesh is used.

As $M_h = \widetilde{M}_h \oplus M_h^*$, we can split the trace :

$$\widehat{p}_h = \lambda_{1,h} + \lambda_{2,h} \quad (6.4.1)$$

with : $\lambda_{1,h} = \widehat{p}_h|_{\widetilde{M}_h}$ et $\lambda_{2,h} = \widehat{p}_h|M_h^*$.

The problem (6.3.1 – 6.3.2) can be rewritten :

$$\sum_{K \in \mathcal{T}_h} \left[\left(\underline{\underline{\mathcal{K}}}^{-1} \mathbf{j}_h^K, \mathbf{v}_h^K \right)_K - (p_h^K, \nabla \cdot \mathbf{v}_h^K)_K + \langle \lambda_{1,h}, \mathbf{v}_h^K \cdot \mathbf{n}_{\partial K} \rangle_{\partial K} + \langle \lambda_{2,h}, \mathbf{v}_h^K \cdot \mathbf{n}_{\partial K} \rangle_{\partial K} \right] = 0 \quad (6.4.2a)$$

$$\sum_{K \in \mathcal{T}_h} \left[\left(\nabla \cdot \mathbf{j}_h^K, w_h^K \right)_K + \langle \tau_{\partial K} p_h^K, w_h^K \rangle_{\partial K} - \langle \tau_{\partial K} \lambda_{1,h}, w_h^K \rangle_{\partial K} - \langle \tau_{\partial K} \lambda_{2,h}, w_h^K \rangle_{\partial K} \right]^{\dagger} = \sum_{K \in \mathcal{T}_h} (f, w_h^K)_K \quad (6.4.2b)$$

$$\sum_{K \in \mathcal{T}_h} \left[\langle \mathbf{j}_h^K \cdot \mathbf{n}_{\partial K}, \mu_{1,h} \rangle_{\partial K} + \langle \tau_{\partial K} p_h^K, \mu_{1,h} \rangle_{\partial K} - \langle \tau_{\partial K} \lambda_{1,h}, \mu_{1,h} \rangle_{\partial K} \right] = \langle g_N, \mu_{1,h} \rangle_{\Gamma_N} \quad (6.4.2c)$$

$$\sum_{K \in \mathcal{T}_h} \left[\langle \mathbf{j}_h^K \cdot \mathbf{n}_{\partial K}, \mu_{2,h} \rangle_{\partial K} + \langle \tau_{\partial K} p_h^K, \mu_{2,h} \rangle_{\partial K} - \langle \tau_{\partial K} \lambda_{2,h}, \mu_{2,h} \rangle_{\partial K} \right] = I_{\text{target}} |\Gamma_I|^{-1} \langle \mu_{2,h}, 1 \rangle_{\Gamma_I} \quad (6.4.2d)$$

for $(\mathbf{v}_h, w_h, \mu_{1,h}, \mu_{2,h}) \in \mathbf{V}_h \times W_h \times \widetilde{M}_h \times M_h^*$.

Remark 6.9 (†). To get this term, we have to make a partial integration. From equation (6.3.2b) :

$$\begin{aligned} & - \left(\mathbf{j}_h^K, \nabla w_h^K \right)_K + \langle \widehat{\mathbf{j}}_h^{\partial K} \cdot \mathbf{n}_{\partial K}, w_h^K \rangle_{\partial K} \\ &= \left(\mathbf{j}_h^K, \nabla w_h^K \right)_K - \left\langle w_h^K, \mathbf{j}_h^K|_{\partial K} \cdot \mathbf{n}_{\partial K} \right\rangle_{\partial K} + \left\langle \mathbf{j}_h^K|_{\partial K} \cdot \mathbf{n}_{\partial K} + \tau_{\partial K} (p_h^K|_{\partial K} - \widehat{p}_h|_{\partial K}), w_h^K \right\rangle_{\partial K} \\ &= \left(\mathbf{j}_h^K, \nabla w_h^K \right)_K - \left\langle \mathbf{j}_h^K|_{\partial K} \cdot \mathbf{n}_{\partial K}, w_h^K \right\rangle_{\partial K} + \left\langle \mathbf{j}_h^K|_{\partial K} \cdot \mathbf{n}_{\partial K}, w_h^K \right\rangle_{\partial K} \\ & \quad + \left\langle \tau_{\partial K} p_h^K, w_h^K \right\rangle_{\partial K} - \left\langle \tau_{\partial K} \lambda_{1,h}, w_h^K \right\rangle_{\partial K} - \left\langle \tau_{\partial K} \lambda_{2,h}, w_h^K \right\rangle_{\partial K} \end{aligned}$$

We can now build the matrix of the system :

$$\begin{bmatrix} \left(\underline{\mathcal{K}}^{-1} \mathbf{j}_h^K, \mathbf{v}_h^K \right)_K & -\left(p_h^K, \nabla \cdot \mathbf{v}_h^K \right)_K & \left\langle \lambda_{1,h}, \mathbf{v}_h^K \cdot \mathbf{n}_{\partial K} \right\rangle_{\partial K} & \left\langle \lambda_{2,h}, \mathbf{v}_h^K \cdot \mathbf{n}_{\partial K} \right\rangle_{\partial K} \\ \left(\nabla \cdot \mathbf{j}_h^K, w_h^K \right)_K & \left\langle \tau_{\partial K} p_h^K, w_h^K \right\rangle_{\partial K} & -\left\langle \tau_{\partial K} \lambda_{1,h}, w_h^K \right\rangle_{\partial K} & -\left\langle \tau_{\partial K} \lambda_{2,h}, w_h^K \right\rangle_{\partial K} \\ \left\langle \mathbf{j}_h^K \cdot \mathbf{n}_{\partial K}, \mu_{1,h} \right\rangle_{\partial K} & \left\langle \tau_{\partial K} p_h^K, \mu_{1,h} \right\rangle_{\partial K} & -\left\langle \tau_{\partial K} \lambda_{1,h}, \mu_{1,h} \right\rangle_{\partial K} & 0 \\ \left\langle \mathbf{j}_h^K \cdot \mathbf{n}_{\partial K}, \mu_{2,h} \right\rangle_{\partial K} & \left\langle \tau_{\partial K} p_h^K, \mu_{2,h} \right\rangle_{\partial K} & 0 & -\left\langle \tau_{\partial K} \lambda_{2,h}, \mu_{2,h} \right\rangle_{\partial K} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{J} \\ P \\ \Lambda_1 \\ \Lambda_2 \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ (f, w_h^K)_K \\ 0 \\ 0 \end{bmatrix} \quad (6.4.3)$$

There is an abuse of notation : in each cell of the matrix, this is the matrix representing the bilinear form, and not the bilinear form itself. The vectors \mathbf{J} , P , Λ_1 and Λ_2 contain the degrees of freedom of \mathbf{j}_h , p_h , $\lambda_{1,h}$ and $\lambda_{2,h}$ respectively.

Note that in the right-hand side, the values of (6.4.2c,6.4.2d) don't appear in this formulation. The corresponding terms are added during the global resolution, see section 6.7.

We define the matrices associated to the bilinear forms :

- $A_{11}^K \leftrightarrow \left(\underline{\mathcal{K}}^{-1} \mathbf{j}_h^K, \mathbf{v}_h^K \right)_K$
- $A_{12}^K \leftrightarrow \left(p_h^K, \nabla \cdot \mathbf{v}_h^K \right)_K$
- $A_{13}^K \leftrightarrow \left\langle \lambda_{1,h}, \mathbf{v}_h^K \cdot \mathbf{n}_{\partial K} \right\rangle_{\partial K}$
- $A_{14}^K \leftrightarrow \left\langle \lambda_{2,h}, \mathbf{v}_h^K \cdot \mathbf{n}_{\partial K} \right\rangle_{\partial K}$
- $A_{22}^K \leftrightarrow \left\langle \tau_{\partial K} p_h^K, w_h^K \right\rangle_{\partial K}$
- $A_{23}^K \leftrightarrow \left\langle \tau_{\partial K} \lambda_{1,h}, w_h^K \right\rangle_{\partial K}$
- $A_{24}^K \leftrightarrow \left\langle \tau_{\partial K} \lambda_{2,h}, w_h^K \right\rangle_{\partial K}$
- $A_{33}^K \leftrightarrow \left\langle \tau_{\partial K} \lambda_{1,h}, \mu_{1,h} \right\rangle_{\partial K}$
- $A_{44}^K \leftrightarrow \left\langle \tau_{\partial K} \lambda_{2,h}, \mu_{2,h} \right\rangle_{\partial K}$
- $A_f^K \leftrightarrow (f, w_h^K)_K$

We also notice that the matrix associated with the bilinear form $\mathbf{j}, w \mapsto \left(\nabla \cdot \mathbf{j}_h^K, w_h^K \right)_K$ is the transpose matrix of A_{12}^K because the training space of \mathbf{j} is the test space of \mathbf{v} . The same statement can be made for all the forms below the diagonal of the matrix. So equation (6.4.3) can be rewritten :

$$\begin{bmatrix} A_{11}^K & -A_{12}^K & A_{13}^K & A_{14}^K \\ (A_{12}^K)^T & A_{22}^K & -A_{23}^K & -A_{24}^K \\ (A_{13}^K)^T & (A_{23}^K)^T & -A_{33}^K & 0 \\ (A_{14}^K)^T & (A_{24}^K)^T & 0 & -A_{44}^K \end{bmatrix} \cdot \begin{bmatrix} \mathbf{J} \\ P \\ \Lambda_1 \\ \Lambda_2 \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ A_F^K \\ 0 \\ 0 \end{bmatrix} \quad (6.4.4)$$

If we are on an element with a side in Γ_I ($\partial K \cap \Gamma_I = \emptyset$), the last « column » and the last « row » of the matrix and Λ_2 are empty.

Lets focus on the dimension of those matrices. Let n_V , n_W , n_M the dimensions of $\mathbf{V}_k(K)$, $W_k(K)$ and $P_k(F)$ respectively, of $K \in \mathcal{T}_h$ and $F \in \partial K$. Let NF be the number of faces in $\partial K \cap (\mathcal{F}_h^0 \cup \mathcal{F}^{\Gamma_N})$. Then we have :

- $A_{11} \in \mathbb{R}^{n_V \times n_V}$,
- $A_{12} \in \mathbb{R}^{n_V \times n_W}$,
- $A_{11} \in \mathbb{R}^{n_V \times (NF)n_M}$,
- $A_{22} \in \mathbb{R}^{n_W \times n_W}$,
- $A_{23} \in \mathbb{R}^{n_W \times (NF)n_M}$,
- $A_{33} \in \mathbb{R}^{(NF)n_M \times (NF)n_M}$,
- $A_f^K \in \mathbb{R}^{n_W \times 1}$,

If $\partial K \cap \Gamma_I \neq \emptyset$, we also have

$$A_{14} \in \mathbb{R}^{n_V \times 1} \quad A_{24} \in \mathbb{R}^{n_W \times 1} \quad A_{44} \in \mathbb{R}^{1 \times 1}$$

otherwise, these three matrices are empty.

6.5 Local solver

We define those two matrices :

$$A^K = \begin{bmatrix} A_{11}^K & -A_{12}^K \\ (A_{12}^K)^T & A_{22}^K \end{bmatrix} \quad B^K = \begin{bmatrix} A_{13}^K & A_{14}^K \\ -A_{23}^K & -A_{24}^K \end{bmatrix} \quad \mathbf{F}^K = \begin{bmatrix} 0 \\ A_f^K \end{bmatrix} \quad (6.5.1)$$

From equations (6.3.2a) and (6.3.2b), which hold in the interior of every $K \in \mathcal{T}_h$, we can solve locally to eliminate \mathbf{j}_h^K and p_h^K in favor of $\lambda_{1,h}$ and $\lambda_{2,h}$:

$$A^K \begin{bmatrix} \mathbf{j}^K \\ \mathbf{p}^K \end{bmatrix} + B^K \begin{bmatrix} \boldsymbol{\lambda}_1^{\partial K} \\ \boldsymbol{\lambda}_2^{\partial K} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ A_f^K \end{bmatrix} \quad (6.5.2)$$

where \mathbf{j}^K , \mathbf{p}^K , $\boldsymbol{\lambda}_1^{\partial K}$ and $\boldsymbol{\lambda}_2^{\partial K}$ are the matrices of the local solutions The solution of this equation is :

$$\begin{bmatrix} \mathbf{j}^K \\ \mathbf{p}^K \end{bmatrix} = -(A^K)^{-1} B^K \begin{bmatrix} \boldsymbol{\lambda}_1^{\partial K} \\ \boldsymbol{\lambda}_2^{\partial K} \end{bmatrix} + (A^K)^{-1} \begin{bmatrix} \mathbf{0} \\ A_f^K \end{bmatrix} \quad (6.5.3)$$

We have expressed \mathbf{j} and p as a function of the trace.

6.6 Flux operator

Now we define the matrices :

$$C^K = \begin{bmatrix} (A_{13}^K)^T & (A_{23}^K)^T \\ (A_{14}^K)^T & A_{24}^K \end{bmatrix} \quad D^K = \begin{bmatrix} A_{33}^K & 0 \\ 0 & A_{44}^K \end{bmatrix} \quad (6.6.1)$$

Let \mathcal{B} the bilinear form induced by the normal flux (see definition 6.7). For $(\mu_{1,h}, \mu_{2,h}) \in \widetilde{M}_h \times M_h^*$, we have :

$$\begin{aligned} \mathcal{B}(\mu_{1,h}, \mu_{2,h}) &= \left\langle \mathbf{j}_h^K|_{\partial K} \cdot \mathbf{n}_{\partial K} + \tau_{\partial K} (p_h^K|_{\partial K} - \hat{p}_h|_{\partial K}), \mu_{1,h} + \mu_{2,h} \right\rangle_{\partial K} \\ &= \left\langle \mathbf{j}_h^K|_{\partial K} \cdot \mathbf{n}_{\partial K} + \tau_{\partial K} (p_h^K|_{\partial K} - \lambda_{1,h}|_{\partial K} - \lambda_{2,h}|_{\partial K}), \mu_{1,h} + \mu_{2,h} \right\rangle_{\partial K} \\ &= \left\langle \mathbf{j}_h^K|_{\partial K} \cdot \mathbf{n}_{\partial K} + \tau_{\partial K} p_h^K|_{\partial K}, \mu_{1,h} + \mu_{2,h} \right\rangle_{\partial K} - \langle \tau_{\partial K} \lambda_{1,h}|_{\partial K}, \mu_{1,h} \rangle_{\partial K} \\ &\quad - \underbrace{\langle \tau_{\partial K} \lambda_{1,h}|_{\partial K}, \mu_{2,h} \rangle_{\partial K} - \langle \tau_{\partial K} \lambda_{2,h}|_{\partial K}, \mu_{1,h} \rangle_{\partial K} - \langle \tau_{\partial K} \lambda_{2,h}|_{\partial K}, \mu_{2,h} \rangle_{\partial K}}_{0 \text{ because } \lambda_{1,h}, \mu_{1,h} \in \widetilde{M}_h \text{ and } \lambda_{2,h}, \mu_{2,h} \in M_h^*} \end{aligned}$$

The matrix representation of this bilinear form is, using equation (6.5.3)

$$\begin{aligned} C^K \begin{bmatrix} \mathbf{j}^K \\ \mathbf{p}^K \end{bmatrix} - D^K \begin{bmatrix} \boldsymbol{\lambda}_1^{\partial K} \\ \boldsymbol{\lambda}_2^{\partial K} \end{bmatrix} &= -C^K (A^K)^{-1} B^K \begin{bmatrix} \boldsymbol{\lambda}_1^{\partial K} \\ \boldsymbol{\lambda}_2^{\partial K} \end{bmatrix} + C^K (A^K)^{-1} \begin{bmatrix} \mathbf{0} \\ A_f^K \end{bmatrix} - D^K \begin{bmatrix} \boldsymbol{\lambda}_1^{\partial K} \\ \boldsymbol{\lambda}_2^{\partial K} \end{bmatrix} \\ &= \mathbf{E}_f^K - E^K \begin{bmatrix} \boldsymbol{\lambda}_1^{\partial K} \\ \boldsymbol{\lambda}_2^{\partial K} \end{bmatrix} \end{aligned} \quad (6.6.2)$$

with $\mathbf{E}_f^K = C^K (A^K)^{-1} \mathbf{F}^K$ and $E^K = C^K (A^K)^{-1} B^K + D^K$

6.7 Global resolution

We have expressed the numerical normal flux as a function of the trace (represented by $\lambda_{1,h}|_{\partial K}$ and $\lambda_{2,h}|_{\partial K}$) for all $K \in \mathcal{T}_h$. It remains to calculate $\lambda_{1,h}$ and $\lambda_{2,h}$.

To solve globally the problem, we form :

- all the E^K in a global matrix \mathbb{H} ,
- all the vectors \mathbf{E}_F^K in a global vector \mathbf{F} ,
- a vector \mathbf{G}_N which contains $\langle g_N, \mu_{1,h} \rangle_{\Gamma_N}$ for the degrees of freedom corresponding to Neumann faces, and 0 elsewhere,
- a vector \mathbf{G}_I which contains only 0, excepted for the degrees of freedom associated to the faces of $\mathcal{F}_h^{I_I}$, with this value $I_{\text{target}}|\Gamma_I|^{-1} \langle \mu_{2,h}, 1 \rangle_{\Gamma_I}$.

$$\mathbb{H}\widehat{\mathbf{p}} = \mathbf{F} + \mathbf{G}_N + \mathbf{G}_I \quad (6.7.1)$$

The Dirichlet boundary condition $p = 0$ on Γ_D is assured by the fact that $\widehat{p}_h = \lambda_{1,h} + \lambda_{2,h}$.

6.8 Implementation in Feel++

To simulate with Feel++, we need to split the mesh : one part corresponding to locations where we have IBC, and the other part containing the remaining. The code to get this is given in listing 3.

```

1 //  $\mathcal{T}_h(\Omega)$ 
2 auto mesh = loadMesh( _mesh = new Mesh<Simplex<d>> );
3 // filter to retrieve the complement of  $\Gamma_I$  in  $\mathcal{F}_h$ 
4 auto complement_integral_bdy = complement( faces( mesh ),
5 [& mesh ]( auto const & e ) {
6     if ( e.hasMarker() && e.marker().matches( mesh->markerName("Ibc*") ) )
7         return true;
8     return false;
9 });
10 //  $\mathcal{F}_h \setminus \mathcal{F}_h^{\Gamma_I}$ 
11 auto face_mesh = createSubmesh( mesh, complement_integral_bdy );
12 //  $\mathcal{F}_h^{\Gamma_I}$ 
13 auto ibc_mesh = createSubmesh( mesh, markedfaces(mesh , "Ibc*") );

```

Listing 3: Construction of \mathcal{F}_h and $\mathcal{F}_h \setminus \mathcal{F}_h^{\Gamma_I}$

Then we need to construct the approximation spaces \mathbf{V}_h , W_H , \widetilde{M}_h and M_h^* . It is possible in the code to use as many IBC as we want, so the product space is :

$$X_h = \mathbf{V}_h \times W_H \times \widetilde{M}_h \times (M_h^*)^n$$

According to what is given in definition 6.6, if we denote by `OrderP` the degree of the polynomials (named k previously), the three first spaces have an ordre `OrderP`, while each IBC sapce has an order of 0 (constant functions are equivalent to polynomials of degree 0).

```

1 Vh_ptr_t Vh = Pdhv<OrderP>( _mesh = mesh );
2 Wh_ptr_t Wh = Pdh<OrderP>( mesh );
3 Mh_ptr_t Mh = Pdh<OrderP>( face_mesh );
4 // only one degree of freedom
5 Ch_ptr_t Ch = Pch<0>( ibc_mesh );
6 // nb_ibc = number of IBC conditions
7 auto ibcSpaces = product( nb_ibc , Ch );
8 auto Xh = product( Vh, Wh, Mh, ibcSpaces );

```

Listing 4: Construction of approximation spaces

The objects used to solve the problem are the `blockform1` and `blockform2` :

```

1 auto a = blockform2( Xh );
2 auto rhs = blockform1( Xh );
3 ...
4 // Assembling the right hand side
5 rhs( 1_c ) += integrate( _range = elements( mesh ), _expr = -f*id(w) );
6 ...
7 // Assembling the main matrix
8 //  $(\lambda u, v)_K$ 
9 a( 0_c, 0_c ) += integrate( _range = elements( mesh ),
10                                _expr = ( trans( lambda*idt(u) ) * id(v) ) );
11 ...
12 //  $\langle \hat{p}_h, v_h^K \cdot n_{\partial K} \rangle$ 
13 a( 0_c, 2_c ) += integrate( _range = internalfaces( mesh ),
14                                _expr = ( idt(phat) * ( leftface(trans(id(v)) * N())
15                                         + rightface( trans(id(v)) * N() ) ) );

```

Listing 5: Construction of the matrix

The elements $0_c, 1_c \dots$ correspond to the index of the test and trial spaces in X_h . So the part of the `blockform2` corresponding to the matrix A_{11} is `a(0_c, 0_c)`.

Finally, the code to solve the system and get the solution is given in listing 6.

```

1 auto U = Xh.element();
2
3 // static condensation is done during the solve
4 a.solve( _solution = U, _rhs = rhs, _name = "hdg" );
5
6 // get back values on each component
7 auto up = U( 0_c ); // element of  $V_h$ 
8 auto pp = U( 1_c ); // element of  $W_h$ 
9 auto phat = U( 2_c ); // element of  $\tilde{M}_h$ 
10 auto ip = U( 3_c, 0 ); // element of  $M_h^*$ 

```

Listing 6: Solve the system and get solutions

Some tests have been implemented with Feel++, see the article [2]. Convergence tests with analytical solutions have been performed in section 5.3, giving the expected results. The tissues perfusion case (see section 4.1) has also been implemented, using the splitting algorithm (see next section), but we didn't manage to run this case in the actual state of the code of Feel++.

7. Time-splitting algorithm

We recall here the equations given by the system (4.1.1) for the model presented in figure 4.1.

$$\mathbf{j} + k_p \nabla p = 0 \quad \text{in } \Omega \times]0, T[\quad (7.0.1a)$$

$$\frac{\partial p}{\partial t} + \nabla \cdot \mathbf{j} = f \quad \text{in } \Omega \times]0, T[\quad (7.0.1b)$$

where p is the pressure, \mathbf{j} is the discharge velocity (blood perfusion velocity) and k_p is the permeability.

We denote by $\boldsymbol{\Pi} = [\Pi_1, \Pi_2, \Pi_3]^T$ the vector of unknown pressures at the circuit nodes. The dynamic of the 0D circuite is described by :

$$\frac{d\boldsymbol{\Pi}}{dt} = \underline{\underline{A}}\boldsymbol{\Pi} + \mathbf{s} + \mathbf{b} \quad (7.0.1c)$$

where $\underline{\underline{A}}$ is a matrix representing the vascular resistances and compliances. Moreover, we have the interfaces conditions :

$$\int_{\Sigma_{\text{lateral}}} \hat{\mathbf{j}} \cdot \mathbf{n} = Q_I \quad p \text{ is constant on } \Sigma_{\text{lateral}} \quad (7.0.1d)$$

$$U_I = p \text{ on } \Sigma_{\text{lateral}} \quad (7.0.1e)$$

and the boundary conditions :

$$p = p_{\text{hole}} \quad \text{on } \Sigma_{\text{hole}} \quad (7.0.1f)$$

$$\mathbf{j} \cdot \mathbf{n} = 0 \quad \text{on } \Sigma_{\text{top}} \cup \Sigma_{\text{bottom}} \quad (7.0.1g)$$

where p_{hole} is known. We also have those initial conditions :

$$p(\mathbf{x}, t = 0) = p_0(\mathbf{x}) \quad \text{in } \Omega \quad (7.0.1h)$$

$$\boldsymbol{\Pi}(t = 0) = \boldsymbol{\Pi}_0 \quad (7.0.1i)$$

7.1 Discretisation

To solve the 3D – 0D coupled system (7.0.1), we begin by performing a semi-discretization in time, in order to maintian the flexibility to apply the HDG method in this time-dependant problem. We introduce $t^n = n\Delta t$ for $n \geq 0$, and for the quantity φ , we denote by φ^n the value of $\varphi(t^n)$.

Given p^n and $\boldsymbol{\Pi}^n$ for $n \geq 0$, to advance from t^n to t^{n+1} , we solve those two steps :

Step 1. Find j, p and $\boldsymbol{\Pi}$ such that :

$$\mathbf{j} + k_p \nabla p = \mathbf{0} \quad \text{in } \Omega \times]t^n, t^{n+1}[\quad (7.1.1a)$$

$$\frac{\partial p}{\partial t} + \nabla \cdot \mathbf{j} = f \quad \text{in } \Omega \times]t^n, t^{n+1}[\quad (7.1.1b)$$

$$\frac{d\boldsymbol{\Pi}}{dt} = \mathbf{b} \quad \text{in } \Omega \times]t^n, t^{n+1}[\quad (7.1.1c)$$

with the boundary conditions

$$p = p_{\text{hole}} \text{ on } \Sigma_{\text{hole}} \quad \mathbf{j} \cdot \mathbf{n} = 0 \text{ on } \Sigma_{\text{top}} \cup \Sigma_{\text{bottom}}, \quad (7.1.2)$$

the interface conditions

$$\int_{\Sigma_{\text{lateral}}} \hat{\mathbf{j}} \cdot \mathbf{n} = Q_I \quad p \text{ is constant on } \Sigma_{\text{lateral}} \quad U_I = p \text{ on } \Sigma_{\text{lateral}}, \quad (7.1.3)$$

and the initial conditions $p(t^n) = p^n$, $\boldsymbol{\Pi}(t^n) = \boldsymbol{\Pi}^n$. Then we set $p^{n+\frac{1}{2}} = p(t^{n+1})$, $\boldsymbol{\Pi}^{n+\frac{1}{2}} = \boldsymbol{\Pi}(t^{n+1})$ and $\mathbf{j}^{n+\frac{1}{2}} = \mathbf{j}(t^{n+1})$.

Step 2. Find p and $\boldsymbol{\Pi}$ such that :

$$\begin{aligned} \frac{\partial p}{\partial t} &= 0 && \text{in } \Omega \times]t^n, t^{n+1}[\\ \frac{d\boldsymbol{\Pi}}{dt} &= \underline{A}\boldsymbol{\Pi} + \mathbf{s} && \text{in }]t, t^{n+1}[\end{aligned} \quad (7.1.4a)$$

with the initial conditions $p(t^n) = p^{n+\frac{1}{2}}$ and $\boldsymbol{\Pi}(t^n) = \boldsymbol{\Pi}^{n+\frac{1}{2}}$. Then we set $p^{n+1} = p(t^{n+1})$ (which equals to $p^{n+\frac{1}{2}}$), $\boldsymbol{\Pi}^{n+1} = \boldsymbol{\Pi}(t^{n+1})$ and $\mathbf{j}^{n+1} = \mathbf{j}(t^{n+1})$ (which is $\hat{\mathbf{j}}^{n+\frac{1}{2}}$).

We see that the problem (equation (7.1.1)) described in step 1 is a Darcy elliptic problem with IBC, as described in equation (6.1.1). The problem (7.1.4) is an ODEs system describing the nonlinear 0D circuit.

7.2 Variational problem for step 1

We recall here the variational formulation given previously in chapter 6.

$$(\underline{\mathcal{K}}^{-1}\mathbf{j}, \mathbf{v})_{\mathcal{T}_h} - (p, \nabla \cdot \mathbf{v})_{\mathcal{T}_h} + \langle \hat{p}, \mathbf{v} \cdot \mathbf{n} \rangle_{\partial\mathcal{T}_h} = 0 \quad (7.2.1a)$$

$$(\nabla \cdot \mathbf{j}, w)_{\mathcal{T}_h} + (\partial_t p, w)_{\mathcal{T}_h} + \langle \tau p, w \rangle_{\partial\mathcal{T}_h} - \langle \tau \hat{p}, w \rangle_{\partial\mathcal{T}_h} = (f, w)_{\mathcal{T}_h} \quad (7.2.1b)$$

$$\langle \hat{\mathbf{j}} \cdot \mathbf{n}, \mu \rangle_{\partial\mathcal{T}_h} = \langle g_N, \mu \rangle_{\Gamma_N} \quad (7.2.1c)$$

Then we split the space. We denote by Γ_{int} the interior faces of the triangulation. Because $p = U_I$ on Γ_I and $\hat{\mathbf{j}} = \mathbf{j} + \tau(p - \hat{p})$, it gives these equations :

$$(\underline{\mathcal{K}}^{-1}\mathbf{j}, \mathbf{v})_{\Omega} - (p, \nabla \cdot \mathbf{v})_{\Omega} + \langle \hat{p}, \mathbf{v} \cdot \mathbf{n} \rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} + \langle U_I, \mathbf{v} \cdot \mathbf{n} \rangle_{\Gamma_I} = 0 \quad (7.2.2a)$$

$$(\nabla \cdot \mathbf{j}, w)_{\Omega} + (\partial_t p, w)_{\Omega} + \langle \tau p, w \rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} - \langle \tau \hat{p}, w \rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} - \langle \tau U_I, w \rangle_{\Gamma_I} = (f, w)_{\Omega} \quad (7.2.2b)$$

$$\langle \hat{\mathbf{j}} \cdot \mathbf{n}, \mu_1 \rangle_{\Gamma_{\text{int}} \cup \Gamma_N} + \langle \tau p, \mu_1 \rangle_{\Gamma_{\text{int}} \cup \Gamma_N} - \langle \tau \hat{p}, \mu_1 \rangle_{\Gamma_{\text{int}} \cup \Gamma_N} = \langle g_N, \mu_1 \rangle_{\Gamma_N} \quad (7.2.2c)$$

Furthermore, the interface condition (4.1.1f) with the electrical formula given in remark 4.2 give :

$$\begin{aligned} \int_{\Gamma_I} \hat{\mathbf{j}} \cdot \mathbf{n} - \left(\frac{U_I - Y}{\Delta t} \right) &= 0 \\ \langle \hat{\mathbf{j}} \cdot \mathbf{n}, \mu_2 \rangle_{\Gamma_I} - \frac{1}{|\Gamma_I|} \left\langle \frac{1}{R_b} U_I, \mu_2 \right\rangle_{\Gamma_I} + \frac{1}{|\Gamma_I|} \left\langle \frac{1}{R_b} Y, \mu_2 \right\rangle_{\Gamma_I} &= 0 \end{aligned}$$

as $\hat{\mathbf{j}} = \mathbf{j} + \tau(p - \hat{p})$, this leads to this equation :

$$\langle \mathbf{j} \cdot \mathbf{n}, \mu_2 \rangle_{\Gamma_I} + \langle \tau p, \mu_2 \rangle_{\Gamma_I} - \langle \tau U_I, \mu_2 \rangle_{\Gamma_I} - \frac{1}{|\Gamma_I|} \left\langle \frac{1}{R_b} U_I, \mu_2 \right\rangle_{\Gamma_I} + \frac{1}{|\Gamma_I|} \left\langle \frac{1}{R_b} Y, \mu_2 \right\rangle_{\Gamma_I} = 0 \quad (7.2.2d)$$

Now, if we take the ODE from step 1 equation (7.1.1c) and apply the same reasoning. This gives the following equation :

$$\underbrace{\frac{1}{|\Gamma_I|} \left\langle C_b \frac{Y}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} - \left(\frac{1}{|\Gamma_I|} \left\langle \frac{1}{R_b} U_I, \mu_3 \right\rangle_{\Gamma_I} - \frac{1}{|\Gamma_I|} \left\langle \frac{1}{R_b} Y, \mu_3 \right\rangle_{\Gamma_I} \right)}_{\langle b, \mu_3 \rangle_{\Gamma_I} = \langle Q_I, \mu_3 \rangle_{\Gamma_I}} = \frac{1}{|\Gamma_I|} \left\langle C_b \frac{Y^{\text{old}}}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} \quad (7.2.2e/A)$$

We could also use the formula $Q_I = \int_{\Gamma_I} \hat{\mathbf{j}} \cdot \mathbf{n}$ in equation (7.1.1c). This leads to this equation :

$$\underbrace{\frac{1}{|\Gamma_I|} \left\langle C_b \frac{Y}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} - \left(\langle \mathbf{j} \cdot \mathbf{n}, \mu_3 \rangle_{\Gamma_I} + \langle \tau p, \mu_3 \rangle_{\Gamma_I} - \langle \tau U_I, \mu_3 \rangle_{\Gamma_I} \right)}_{\langle b, \mu_3 \rangle_{\Gamma_I} = \langle Q_I, \mu_3 \rangle_{\Gamma_I}} = \frac{1}{|\Gamma_I|} \left\langle C_b \frac{Y^{\text{old}}}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} \quad (7.2.2e/B)$$

The test function μ_1 lives in \widetilde{M}_h (see chapter 6), while μ_2 and μ_3 are functions of M_h^* (with isomorphic to \mathbb{R}).

Remark 7.1. Here, we used the first order time approximation $\frac{dY}{dt}(t) \approx \frac{Y - Y^{\text{old}}}{\Delta t}$. We could use another discretization, the code in Feel++ automatically deals with it, using *Backward differencing formula*.

The variational problem for step 1 is given by equation (7.2.2), where $(\mathbf{j}, p, \hat{p}, U_I, Y) \in \mathbf{V}_h \times W_h \times M_h \times \mathbb{R} \times \mathbb{R}$ are the trial functions, and $(\mathbf{v}, w, \mu_1, \mu_2, \mu_3) \in \mathbf{V}_h \times W_h \times M_h \times \mathbb{R} \times \mathbb{R}$ are the test functions.

In figure 7.1 is represented the finite elements matrix from the system of equations (7.2.2). The three first lines correspond to the 3D problem, which is implemented in Feel++ in the class `MixedPoisson`. Two matrices are given, depending on what formula for Q_I is used. The highlighted terms correspond to the contribution of the coupling. On the further tests, we will try to uncouple the system : those terms will land on the right-hand side of the equation, with their analytical values.

$$\left[\begin{array}{cccc} \left(\underline{\mathcal{K}}^{-1} \mathbf{j}, \mathbf{v} \right)_\Omega & \left(-p, \nabla \cdot \mathbf{v} \right)_\Omega & \left\langle \hat{p}, \mathbf{v} \cdot \mathbf{n} \right\rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} & \left\langle U_I, \mathbf{v} \cdot \mathbf{n} \right\rangle_{\Gamma_I} \\ \left(\nabla \cdot \mathbf{j}, w \right)_\Omega & \left(\partial_t p, w \right)_\Omega + \left\langle \tau p, w \right\rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} & - \left\langle \tau \hat{p}, w \right\rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} & - \left\langle \tau U_I, w \right\rangle_{\Gamma_I} \\ \left\langle \mathbf{j} \cdot \mathbf{n}, \mu_1 \right\rangle_{\Gamma_{\text{int}} \cup \Gamma_N} & \left\langle \tau p, \mu_1 \right\rangle_{\Gamma_{\text{int}} \cup \Gamma_N} & - \left\langle \tau \hat{p}, \mu_1 \right\rangle_{\Gamma_{\text{int}} \cup \Gamma_N} & - \left\langle \tau U_I, \mu_2 \right\rangle_{\Gamma_I} - \left\langle \frac{1}{|\Gamma_I| R_b} U_I, \mu_2 \right\rangle_{\Gamma_I} \\ \left\langle \mathbf{j} \cdot \mathbf{n}, \mu_2 \right\rangle_{\Gamma_I} & \left\langle \tau p, \mu_2 \right\rangle_{\Gamma_I} & & \left\langle \tau U_I, \mu_3 \right\rangle_{\Gamma_I} \\ - \left\langle \mathbf{j} \cdot \mathbf{n}, \mu_3 \right\rangle_{\Gamma_I} & - \left\langle \tau p, \mu_3 \right\rangle_{\Gamma_I} & & \left\langle \frac{C_b}{|\Gamma_I|} \frac{Y}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} \end{array} \right]$$

(a) Using the formula $Q_I = \int_{\Gamma_I} \hat{\mathbf{j}} \cdot \mathbf{n}$ (`coupling.mode=1`)

$$\left[\begin{array}{cccc} \left(\underline{\mathcal{K}}^{-1} \mathbf{j}, \mathbf{v} \right)_\Omega & \left(-p, \nabla \cdot \mathbf{v} \right)_\Omega & \left\langle \hat{p}, \mathbf{v} \cdot \mathbf{n} \right\rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} & \left\langle U_I, \mathbf{v} \cdot \mathbf{n} \right\rangle_{\Gamma_I} \\ \left(\nabla \cdot \mathbf{j}, w \right)_\Omega & \left(\partial_t p, w \right)_\Omega + \left\langle \tau p, w \right\rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} & - \left\langle \tau \hat{p}, w \right\rangle_{\Gamma_{\text{int}} \cup \Gamma \setminus \Gamma_I} & - \left\langle \tau U_I, w \right\rangle_{\Gamma_I} \\ \left\langle \mathbf{j} \cdot \mathbf{n}, \mu_1 \right\rangle_{\Gamma_{\text{int}} \cup \Gamma_N} & \left\langle \tau p, \mu_1 \right\rangle_{\Gamma_{\text{int}} \cup \Gamma_N} & - \left\langle \tau \hat{p}, \mu_1 \right\rangle_{\Gamma_{\text{int}} \cup \Gamma_N} & - \left\langle \tau U_I, \mu_2 \right\rangle_{\Gamma_I} - \left\langle \frac{1}{|\Gamma_I| R_b} U_I, \mu_2 \right\rangle_{\Gamma_I} \\ \left\langle \mathbf{j} \cdot \mathbf{n}, \mu_2 \right\rangle_{\Gamma_I} & \left\langle \tau p, \mu_2 \right\rangle_{\Gamma_I} & & \left\langle \frac{1}{|\Gamma_I| R_b} U_I, \mu_3 \right\rangle_{\Gamma_I} \\ - \left\langle \mathbf{j} \cdot \mathbf{n}, \mu_3 \right\rangle_{\Gamma_I} & - \left\langle \tau p, \mu_3 \right\rangle_{\Gamma_I} & & \left\langle \frac{C_b}{|\Gamma_I|} \frac{Y}{\Delta t} + \frac{1}{|\Gamma_I| R_b} Y, \mu_3 \right\rangle_{\Gamma_I} \end{array} \right]$$

(b) Using the formula $Q_I = \frac{U_I - Y}{R_b}$ (`coupling.mode=2`)

Figure 7.1: FE matrix for step 1

Now, let's look at the right-hand side. It is given in figure 7.2, with on the one hand the case where the coupling is enabled, and on the other hand when it is disabled.

$$\left[\begin{array}{c} F_j \\ F_p \\ F_{\hat{p}} \\ 0 \\ \frac{1}{|\Gamma_I|} \left\langle C_b \frac{Y^{\text{old}}}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} \end{array} \right] \quad \left[\begin{array}{c} F_j \\ F_p \\ F_{\hat{p}} \\ - \frac{1}{|\Gamma_I|} \left\langle \frac{Y^{\text{exact}}}{|\Gamma_I|}, \mu_2 \right\rangle_{\Gamma_I} \\ \frac{1}{|\Gamma_I|} \left\langle C_b \frac{Y^{\text{old}}}{\Delta t}, \mu_3 \right\rangle_{\Gamma_I} + \frac{1}{|\Gamma_I|} \left\langle Q_I^{\text{exact}}, \mu_3 \right\rangle_{\Gamma_I} \end{array} \right]$$

(a) with coupling (`coupling.mode={1,2}`)

(b) w/o coupling (`coupling.mode=0`)

Figure 7.2: Right-hand size for the FE problem

Part III

Reduced Order Methods

This part deals with the reduced basis method. The chapter 8 will present the theory and the methods, while chapter 9 will show the implementation of those methods. The purpose of this study is to set up an environment allowing to run sensibility analysis on the parameters of the eye, especially computing Sobol indices. The parameters can be physiological, such as the intraocular pressure, or geometrical such as the shape of the eye.

This chapter deals with model order reduction (*MOR*) and reduced basis method (*RB*). In the following, \mathcal{N} will denote the finite element problem size (also denoted `NN` in the programs), and N the reduced size. We have $N \ll \mathcal{N}$.

In this chapter, we won't deal with models related to the geometry of the eye, but with a heat problem of a thermal fin on a CPU. We study this model because we studied it in class during the RB class [12], so we will have a reference to compare the results. In the end, the code produced during the internship can work with several models.

8.1 Thermal-fin model

We will begin by summarizing the practical work done during the class on a reduced basis [12]. The codes and reports for the work produced during this class are gathered in this repository.

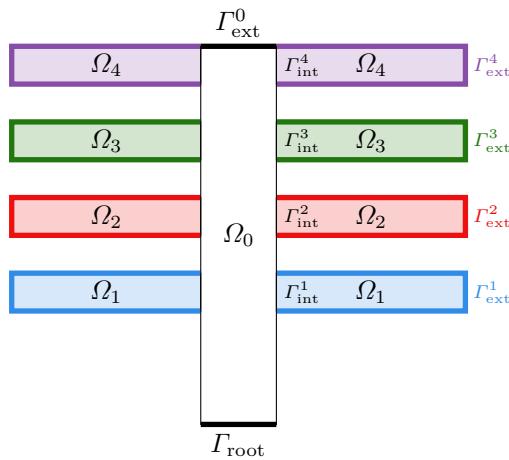


Figure 8.1: Geometry of the thermal fin

We consider the geometry given in figure 8.1, composed of 5 domains Ω_i . On domain Ω_i , the temperature u_i is governed by the elliptic PDE :

$$-k_i \Delta u_i = 0 \quad (8.1.1)$$

Furthermore, we add those boundary conditions :

$$u_0 = u_i \quad -(\nabla u_0 \cdot n_i) = -k_i (\nabla u_i \cdot n_i) \quad \text{on } \Gamma_i^{\text{int}} \text{ for } i \in \{1, 2, 3, 4\} \quad (8.1.2)$$

$$-(\nabla u_0 \cdot n_i) = -1 \quad \text{on } \Gamma_{\text{root}} \quad (8.1.3)$$

$$-k_i (\nabla u_i \cdot n_i) = \text{Bi} u_i \quad \text{on } \Gamma_i^{\text{ext}} \text{ for } i \in \{0, 1, 2, 3, 4\} \quad (8.1.4)$$

The parameters involved in this problem are the conductivities $k_i \in [0.1, 10]$ and the Biot number $\text{Bi} \in [0.01, 1]$.

1. The first problem is a theoretical study of the problem.
2. The second one is the implementation of the reduced basis method, for a time-independent problem.
3. The third problem focuses on *a posteriori* error estimation, error bound, and a greedy sampling procedure
4. Finally, problem 4 treats a time-dependent problem, with a POD-Greedy sampling procedure.

We took back the code produced during this project and grouped it in a Python module `ReducedBasis`. The classes produced are gathered in section 9.1.

Several geometries can be imagined from this one, depending on the parameters : we could the size of the fin, the number of them, or the number of dimensions of the model. We wrote a Python script, `case_generator` doing this. The principle of this script is detailed in section 9.2.

8.2 Reduced Basis and Empirical Interpolation Method

In the course *Calcul Scientifique 3* [12], we studied the reduced basis method (RB) to solve PDE. In this method, we have a parameter dependant problem to solve, such as : *Find $u(\boldsymbol{\mu}) \in V$ such that $\forall v \in V$*

$$a(u(\boldsymbol{\mu}), v) = f(v, \boldsymbol{\mu}) \quad (8.2.1a)$$

and evaluate

$$s(\boldsymbol{\mu}) = l(u(\boldsymbol{\mu}), \boldsymbol{\mu}) \quad (8.2.1b)$$

The parameter $\boldsymbol{\mu}$ lives in a set $D \subset \mathbb{R}^d$. To solve such a problem, we decompose the bilinear form a in a sum :

$$a(u, v; \boldsymbol{\mu}) = \sum_{q=0}^Q \theta^q(\boldsymbol{\mu}) a^q(u, v) \quad (8.2.2)$$

with θ^q a function of the parameter $\boldsymbol{\mu}$, and a^q independant of $\boldsymbol{\mu}$. In some case, this decomposition may not be possible. To do so, we can use the *Empirical Interpolation Method* (EIM). The principle of this method is to approximate a parametrized function by a sum of affine terms :

$$g(x, \boldsymbol{\mu}) \approx g_M(x, \boldsymbol{\mu}) = \sum_{m=1}^M \theta_{g,M}^m(\boldsymbol{\mu}) q_m(x) \quad (8.2.3)$$

The content of this section is taken from the thesis of Romain Hild [7, Chapter 3].

Let Ξ_{train} a subset of D . We denote by $g_{\boldsymbol{\mu}}$ the function $x \mapsto g(x, \boldsymbol{\mu})$ for a given $\boldsymbol{\mu}$, and $g_{M,\boldsymbol{\mu}}$ the function $x \mapsto g_M(x, \boldsymbol{\mu})$.

Algorithm 1: EIM approximation

Input: $\Xi_{\text{train}}, \varepsilon > 0$ a tolerance

Choose $\boldsymbol{\mu}_1 \in \Xi_{\text{train}}$ such that $g_{\boldsymbol{\mu}_1} \neq 0$

$$t_1 \leftarrow \arg \max_{x \in \Omega} |g_{\boldsymbol{\mu}_1}(x)|^\ddagger, q_1 \leftarrow \frac{g_{\boldsymbol{\mu}_1}(x)}{g_{\boldsymbol{\mu}_1}(t_1)}$$

$$e_1 \leftarrow \|g_{\boldsymbol{\mu}_1} - g_{1,\boldsymbol{\mu}_1}\|_{L^2(\Omega)}$$

while $e_{m-1} > \varepsilon$ **do**

$$\boldsymbol{\mu}_m \leftarrow \arg \max_{\boldsymbol{\mu} \in \Xi_{\text{train}}} \|g_{\boldsymbol{\mu}} - g_{m-1,\boldsymbol{\mu}}\|$$

Set $T^{m-1} \in \mathbb{R}^{(m-1) \times (m-1)}$ defined by $(T^{m-1})_{ij} = q_j(t_i)$ for $i, j \in \llbracket 1, m-1 \rrbracket$, and $g_{\boldsymbol{\mu}_m}^{m-1} \in \mathbb{R}^{m-1}$ defined by

$$(g_{\boldsymbol{\mu}_m}^{m-1})_i = g_{\boldsymbol{\mu}_m}(t_i) \text{ for } i \in \llbracket 1, m-1 \rrbracket$$

$$\text{Solve } T^{m-1} \theta_{g,m-1} = g_{\boldsymbol{\mu}_m}^{m-1}$$

$$r_m \leftarrow g_{\boldsymbol{\mu}_m} - g_{m,\boldsymbol{\mu}_m}, t_m \leftarrow \arg \sup_{x \in \Omega} |r_m(x)|^\ddagger, q_m(x) \leftarrow \frac{r_m(x)}{r_m(t_m)}$$

$$e_m \leftarrow \|r_m\|_{L^2(\Omega)}, m \leftarrow m + 1$$

end

Output: $[q_1, \dots, q_M], \{t_1, \dots, t_M\}$

This algorithm returns a basis q_1, \dots, q_M of linearly independant functions, interpolation points t_1, \dots, t_M such that the matrix $T_{ij} = q_j(t_i)$ is lower triangular of size $M \times M$, with unity diagonal.

To compute $g_{M,\boldsymbol{\mu}}$ for all $\boldsymbol{\mu} \in D$, we follow these steps :

1. Compute the vector $g_{\boldsymbol{\mu}}^M$ defined by $(g_{\boldsymbol{\mu}}^M)_i = g(t_i, \boldsymbol{\mu})$

2. Solve the equation of unknown $\theta_{g,M}$

$$T \theta_{g,M} = g_{\boldsymbol{\mu}}^M \quad (8.2.4)$$

3. The approximation of $g(x, \boldsymbol{\mu})$ is

$$g(x, \boldsymbol{\mu}) \approx \sum_{m=1}^M \theta_{g,M}^m q_m(x) \quad (8.2.5)$$

where $\theta_{g,M}^m$ is the m -th coordinate of the vector $\theta_{g,M}$

Example :

$$a(u, v; \mu) = \int_{\Omega} g(x; \mu) b(u, v; x) dx$$

Then using the EIM decomposition of $g(x, \mu) \approx \sum_{m=1}^M \theta_{g,M}^m$ we have the following approximation for a :

$$a(u, v, \mu) \approx \sum_{m=1}^M \theta_{g,M}^m(\mu) \int_{\Omega} q_m(x) b(u, v; x) dx = \sum_{m=1}^M \theta_{g,M}^m(\mu) a_q(u, v)$$

With complex operators, it can be difficult to have an analytical expression of the non-affine component. We can directly calculate the discrete operator. The steps given in algorithm 1 are the same, but instead of using functions spaces, we act directly on vectors or matrices. We want to have :

$$\mathbf{T}(x, \mu) \approx \mathbf{T}_M(x, \mu) = \sum_{m=1}^M \Theta_m(\mu) \Phi^m(x) \quad (8.2.6)$$

The main difference between the two methods is that instead of taking $t_m = \arg \max_{x \in \Omega} f(x)$ (see \ddagger in algorithm 1), we set an interpolation index

$$i_m = \arg \max_{j \in \mathbb{I}} |\mathbf{R}_m(x, \mu)_j| \quad (8.2.7)$$

The set \mathbb{I} contains tuples of integers defining the index of an element of a vector (then the tuple has only one value) or a matrix (the tuple has two values).

8.3 *A posteriori* error estimation

As we saw earlier, we have the following decompositions :

$$A(\mu) = \sum_{q=1}^{Q_A} \beta_A^q A^q \quad F(\mu) = \sum_{p=1}^{Q_F} \beta_F^p F^p \quad (8.3.1)$$

From now on we will use this convention : q [resp. p] is the index for the decomposition of A [resp. F]. In the following, we may forget to specify the bounds on sums, but the indices will be consistent with this convention (n will goes from 1 to the reduced dimension N).

The residual error satisfies this equation :

$$(\hat{e}(\mu), v)_X = \sum_p \beta_F^p(\mu) f^q(v) - \sum_q \sum_n \beta_A^q(\mu) u_N^n(\mu) a^q(\xi^n, v) \quad \forall v \in X \quad (8.3.2)$$

We recall that $(u, v)_X = v^T A_{\bar{\mu}} u$. This identity, using the superposition principle, leads to the following equation :

$$\hat{e}(\mu) = \sum_p \beta_F^p(\mu) \mathcal{S}^p + \sum_q \sum_n \beta_A^q(\mu) u_N^n(\mu) \mathcal{L}^{n,q} \quad (8.3.3a)$$

with :

$$\begin{aligned} (\mathcal{S}^p, v) &= f^p(v) & \forall v \in X, \forall p \in \llbracket 1, Q_F \rrbracket \\ (\mathcal{L}^{n,q}, v) &= -a^q(\xi^n, v) & \forall v \in X, \forall n \in \llbracket 1, N \rrbracket, \forall q \in \llbracket 1, Q_A \rrbracket \end{aligned} \quad (8.3.3b)$$

Therefore, we have

$$A_{\bar{\mu}} \mathcal{S}^p = F^p \quad A_{\bar{\mu}} \mathcal{L}^{n,q} = -A^q \xi^n \quad (8.3.4)$$

In the following, the parameter dependency is not written : the terms in blue correspond to parameter-dependent terms, while the one in red are independent of μ .

$$\begin{aligned} \|\hat{e}(\mu)\|_X^2 &= (\hat{e}(\mu), \hat{e}(\mu))_X \\ &= \left(\sum_p \beta_F^p \mathcal{S}^p + \sum_q \sum_n \beta_A^q u_N^n \mathcal{L}^{n,q}, \sum_p \beta_F^p \mathcal{S}^p + \sum_q \sum_n \beta_A^q u_N^n \mathcal{L}^{n,q} \right)_X \end{aligned}$$

As $A_{\bar{\mu}}$ is symmetric, we have $(u, v)_X = (v, u)_X$, so

$$\|\hat{e}(\mu)\|_X^2 = \sum_p \sum_{p'} \beta_F^p \beta_F^{p'} (\mathcal{S}^p, \mathcal{S}^{p'})_X + 2 \sum_p \sum_q \sum_n \beta_F^p \beta_A^q u_N^n (\mathcal{S}^p, \mathcal{L}^{n,q})_X + \sum_q \sum_n \sum_{q'} \sum_{n'} \beta_A^q \beta_A^{q'} u_N^n u_N^{n'} (\mathcal{L}^{n',q'}, \mathcal{L}^{n,q})_X \quad (8.3.5)$$

All the red terms in the previous equation can be calculated once and stored, to use them afterwards. Moreover, these calculs are costly because of the vectors of size \mathcal{N} involved. This is the *offline* stage. The data to be stored are

- a matrix of shape (Q_F, Q_F) , defined by $\mathbf{SS}_{p,p'} = (\mathcal{S}^p, \mathcal{S}^{p'})_X$,
- a tensor of shape (Q_A, Q_F, N) , defined by $\mathbf{SL}_{q,p,n} = (\mathcal{S}^p, \mathcal{L}^{n,q})_X$
- a tensor of shape (Q_A, N, Q_A, N) , defined by $\mathbf{LL}_{q,n,q',n'} = (\mathcal{L}^{n,q}, \mathcal{L}^{n',q'})_X$

On the other hand, the blue terms are easily computed from a parameter μ . The final computation using the stored results and these terms is the *online* stage.

8.4 Greedy algorithm

Definition 8.1. The *relative output error bound* is defined by the formula

$$\Delta_N(\mu) = \frac{\|\hat{e}(\mu)\|_X}{\alpha_{lb}(\mu)} \quad (8.4.1)$$

Where

- $\|\hat{e}(\mu)\|_X$ is the norm of the residual error (see section 8.3)
- $\alpha_{lb}(\mu)$ is the coercivity constant of the bilinear form a , defined by $\alpha_{lb}(\mu) := \inf_{v \in X} \frac{a(v, v; \mu)}{\|v\|_X}$.

The algorithm 2 presents the Greedy algorithm. The main principle of this algorithm is to test which parameters maximize the energy Δ_N and add it to the sample.

Algorithm 2: Greedy algorithm

```

Input:  $\mu_0 \in D$  and  $\Xi_{\text{train}} \subset D$ 
 $S \leftarrow [\mu_0]$ 
while  $\Delta_N^{\max} > \varepsilon$  do
     $u(\mu^*) \leftarrow \text{FE solution, using } S \text{ as generating sample}^a$ 
     $W_N \leftarrow \{\xi = u(m^*)\} \cup W_{N-1}$ 
     $\mu^* \leftarrow \arg \max_{\mu \in \Xi_{\text{train}}} \Delta_N(\mu) \text{ (and } \Delta_N^{\max} \leftarrow \max_{\mu \in \Xi_{\text{train}}} \Delta_N(\mu))$ 
    Append  $\mu^*$  to  $S$ 
end
Output: sample  $S$ , reduced basis  $W$ 

```

^aas new elements will be added to S , there is no need to compute the full basis at each step, but we can increase it.

In the test made during the internship (see section 9.9), the sampling Ξ_{train} contains only 100 parameters.

If the problem studied is time-dependant, we use the POD(t)-Greedy(μ) algorithm, described in algorithm 3. POD, for *Proper Orthogonal Decomposition*, means that we optimally capture causality of time variation by keeping the m POD modes at each step. Those modes are computed from the eigenvectors of the matrix $\frac{1}{K}(e_K, e_K)_X$, where K is

the number of time steps used in the simulation.

Algorithm 3: POD-Greedy algorithm

Input: $\mu_0 \in D$ and $\Xi_{\text{train}} \subset D$, $m \in \mathbb{N}^{+*}$
 $S \leftarrow [\mu_0]$
while $\Delta_N^{\max} > \varepsilon$ **do**
 for $k \in 1, K$ **do**
 $u^k(\mu^*) \leftarrow$ FE solution at time t^k ; $u_N^k(\mu^*) \leftarrow$ RB solution at time t^k
 $e_K^k \leftarrow u^k(\mu^*) - \Pi_N u_N^k(\mu^*)$
 end
 $\mathbf{V} \leftarrow$ eigenvectors of $\frac{1}{K}(e_K, e_K)_X$, associated to the m greater eigenvalues
 $\Psi^{\text{POD},i} = \sum_{k=1}^K \mathbf{V}_i^k u^k(\mu^*)$, added to the reduced basis, for $i \in \llbracket 1, m \rrbracket$
 $\mu^* \leftarrow \arg \max_{\mu \in \Xi_{\text{train}}} \Delta_N^K(\mu)$ (and $\Delta_N^{\max} \leftarrow \max_{\mu \in \Xi_{\text{train}}} \Delta_N^K(\mu)$)
 Append μ^* to S
end
Output: S , reduced basis

9.1 Implementation using only NumPy

This section gathers the documentation for the first implementation of the problem. This corresponds to the code produced during the S3 class *Calcul Scientifique 3* [12], see this repository.

Remark 9.1. This documentation is up to date with the commit `ReducedBasisNumPy` of the repository Eye2brain.

9.1.1 ReducedBasisCst

`ReducedBasisCst` is a class corresponding to what is developed for problem 2, without time dependency.

- `__init__(Aq, Fh, paramFunc)` : initiates the object. The arguments are explained in this table :

Name	Type	Description
Aq	list of matrices	list of the matrices of the forms a_q (see report 1)
Fh	NumPy array	Vector of the right-hand side of the equation
paramFunc	python function	function which returns the list of parameters [$k_1, k_2, k_3, k_4, k_0, Bi$] from given parameters

- `gramSchmidt(Z)` : runs the Gram-Schmidt algorithm on the matrix Z . The result of the call is a matrix orthonormal according to the scalar product $(u, v)_{\bar{A}} = v^T \cdot \bar{A} \cdot u$, with $\bar{A} = A(k_i = 1, Bi = 0.1)$.
- `assembleA(mu)` : assemble the FE matrix for a given list of parameters $\mu = \{k_1, k_2, k_3, k_4, k_0, Bi\}$
- `generateOffline(mu)` : generate the approximate basis from a given set of parameters (this set contains arguments of the `paramFunc`), as well as the approximates matrices A_N^q of the linear decomposition, and the right-hand side.
- `generateOfflineFromZ(Z)` : computes the appriximates matrices A_N^q and the right-hand side, from a reduced basis matrix given.
- `computeTNroot(mu)` : returns the value of $T_{N,\text{root}}$, for a givne parameter μ .
- `getSolutions(mu)` : returns the tuple $(u, TNroot)$, where u is the vector solution of the reduced problem and $TNroot$ as in the previous method.

9.1.2 ReducedBasis

The class `ReducedBasis` heritages from the class `ReducedBasisCst` described in the previous section. It deals with the time-dependent problem. The parameter-dependant problem to be solved is

$$m \left(\frac{u^k(\mu) - u^{k-1}(\mu)}{\Delta t}, v \right) + a(u^k(\mu), v; \mu) = f(v)g(t^k) \quad (9.1.1)$$

with g a time-dependant function.

- `__init__(Aq, Fh, paramFunc, alphaLB, M, tf, K)` : initiates the object. The arguments are the one described in the class `ReducedBasisCst`, to which we added these :

Name	Type	Description
alphaLB	python function	function which returns the theorical value of alpha_LB from the list of parameters
M	matrix	matrix of mass of the function m
tf	float	final time
K	integer	number of time intervals, so $\Delta t = \frac{tf}{K}$

- `computeOffline` and `computeOfflineFromZ` are overloaded functions from the mother class, because of the presence of `M`.

- `generateApproxBasis(musk)` : generates the reduced basis matrix from different parameters and different instants. The argument `musk` is a `dict` containing the desired values of μ associated to every instant where we chose. For the example given in the subject, we take `musk = 0.01:[1,5,10,20,30], 0.1:[5,10,20], 1:[5,10]`.
- `solveTime(mus, g)` and `solveTimeForStudy(mus, g)` : solves the time-dependant equation for a given parameter and time dependant function `g`. The argument `mus` is a list of all parameters of the problem, and `g` is a vector containing the value of the funcitons `g` at times t^k for $k \in [1, K]$. The first function just returns the solution $u^K(\mu)$, while the return more detailed results :

Returned value	Description
<code>t</code>	time axis
<code>sN [s]</code>	output of RB method [FE]
<code>sDiff</code>	difference between the two methods
<code>normN [norm]</code>	energy norm of RB method [FE]
<code>normDiff</code>	difference between the two methods

These results are all NumPy arrays.

- `computeErrorDirect(mu, g, computeEnergyNorm=False)` : compute the error with a direct method.
- `computeOfflineError(mu, g)` : stores the offline data for error bound computation. More more details, report to problem 4, part 2.
- `computeOnlineError(mu, g, computeEnergyNorm=False)` : computes the online error bound. A call to `computeOfflineError` must be done before. The optional argument `computeEnergyNorm` is used in the POD-Greedy function.
- `podGreedy(mu0, mu_train, g, eps_tol)` : runs the POD-Greedy algorithm. `mu0` is the initial list of parameters, `mu_train` is a set of parameters wehre the algorithm will run his tests, `g` is the time-dependant function and `eps_tol` is the limit tolerance for the maximal of the error Δ_N . The result of this funciton is a tuple composed of :

Name	Type	Description
<code>Z</code>	NumPy matrix	matrix of the reduced basis (<code>self.Z</code>)
<code>SN</code>	list	list of all the parameters corresponding to the reduced basis
<code>df_en</code>	pandas DataFrame	values of the error obund at each iteration of the algorithm
<code>maxs</code>	list	values of Δ_N^{\max} at each iteration

the two last arguments allow us to make a study of the behavior of the algorithm.

Remark 9.2. In the following, an implementation of the POD-Greedy algorithm with many POD-many was made, but at the end of the internship, this was not giving proper results.

9.2 Case generator

Many geometrical parameters are involved in the geometry described earlier : we could set the size of the fins, the number of them for example, or even chose to have a three-dimensional model. The purpose of the script python `generate_cases` is to generate from given parameters the files used by Feel++ : cfg, JSON and geo.

To do it, we use the python module `liquidpy` which takes templates of files (see this link), where « blanks » are left. For instance, in the cfg template, the line to set the dimension of the case is

```
case.dimension={{ dim }}
```

In the Python script, the lines to generate the rendered file are the following. In the end, we just have to save the `str` generated in a file.

```
env = Environment(loader=FileSystemLoader( "templates/" ))
templateCfg = env.get_template( "thermal-fin.cfg" )
renderCfg = templateCfg.render( dim = args.dim )
```

To set the parameter `N` for example, we simply have to call the script with the option `--N 3`. The parameters that can be set are :

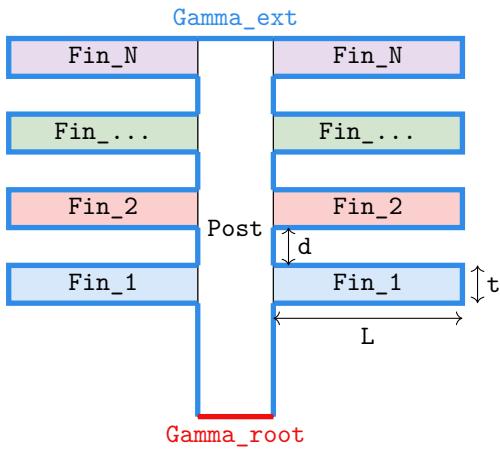


Figure 9.1: Geometry with different parameters

Cylinder	0	1	2
Figure			
Nb of elt	1 773 437	1 014 216	999 087

Table 9.1: 3D geometries

Name	Description	Default value
N	number of fins	4
L	width of a fin	2.5
d	distance between two fins	0.5
t	thickness of a fin	0.25
dim	dimension of the case (2 or 3)	2
cylinder	shape of fin and post (0=boxes, 1=box/cylinders, 2=cylinders)	0

The figure 9.1 shows some of those parameters. In three dimension, three cases are possible, depending on the parameter `cylinder` :

- `cylinder = 0` : the geometry is composed only of boxes.
- `cylinder = 1` : the central part of the thermal fin (Post) is a box and the fins are cylinders.
- `cylinder = 2` : both central part and fins are cylinders.

A figure of these geometries, with its number of elements, if given in table 9.1.

The more tricky part of the templates is because of the parameter `N` : in the JSON, we have to list all the markers associated with the fins. But this can easily be handled by liquid. For example, in `N = 5`, the code

```
"init":
{
  "markers": [% for item in fins %]"Fin_{% item %}", {% endfor %}"Post"] ,
  "expr": "0"
}
```

will render this, by setting `fins = list(range(1,N+1))` in the renderer :

```
"init":
{
  "markers": ["Fin_1", "Fin_2", "Fin_3", "Fin_4", "Fin_5", "Post"],
```

```
"expr": "0"
}
```

9.3 Pyfeelpp-mor

The programs of Feel++ dealing with reduced basis can be used in a Python module `feelpp.mor`. To get the module available in a Python script, we have to do the following step, after the compilation is configured with CMake as explained previously in appendix A :

```
cd mor
make install [-j 12]
```

To have all the libraries required by Python installed, the targets that we have to construct are :

- `feelpp/pyfeelpp`
- `toolboxes/pyfeelpp-toolboxes`
- `feelpp/contrib`
- `feelpp/feel`

The `install` command will create a directory inside `build` were all the applications generated and Python modules will be stored. To make thoses modules accesible to a Python script, we have to add the path to this folder in then environment variable `PYTHONPATH` :

```
export PYTHONPATH=<build dir>/install/lib/python3.8/site-packages:$PYTHONPATH
export LD_LIBRARY_PATH=<build dir>/install/lib/:<build dir>/install/lib/python3.8/site-packages/
```

The tool used to « convert » C++ code to Python is `pybind` [15]. Here is an example of code to create a Python module from C++ code.

```
#include <pybind11/pybind11.h>

int add(int i, int j) { return i + j; }

PYBIND11_MODULE(python_example, m)
{
    m.def("add", &add, "Add two numbers");
    m.attr("__version__") = "1.0";
}
```

With the method `def` we define methods of the generated module, and with the method `attr` we define attributes of the method. Then, with pip or CMake, we can install the module to use it in Python. Such an operation will create a dynamic library (a `so` file) in the `site-packages` folder.

Then we can access this library in a Python script, either by being in the right folder, either by updating the `PYTHONPATH`, as described earlier.

```
>>> import python_example
>>> python_example.add(1,2)
3
```

9.4 The script `toolboxmor.py`

In this section, I will present the Python script `toolboxmor.py`¹.

Remark 9.3. Usually, we are interested in running directly this script, with `python3` :

¹The state of the file at this time can be found here

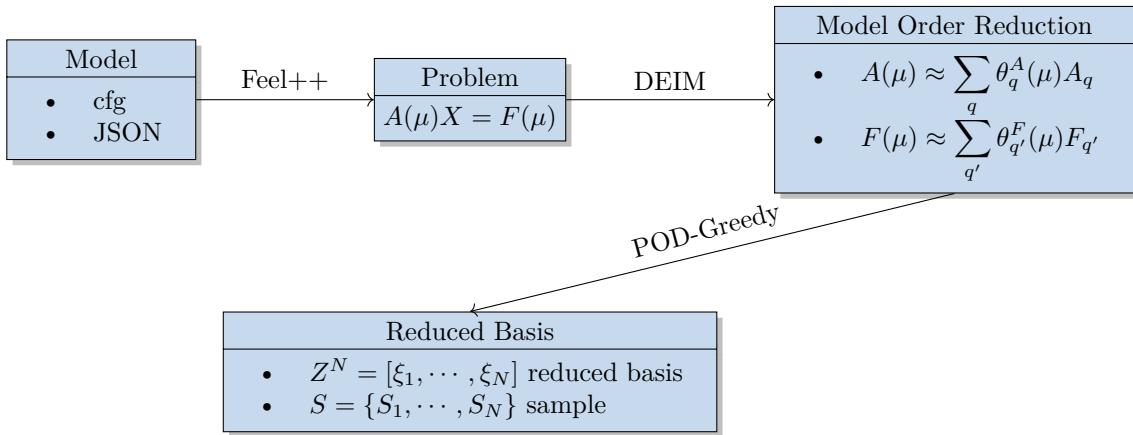


Figure 9.2: Pipeline for Model Order Reduction

```
python3 toolboxmor.py --config-file opusheat/opusheat-heat.cfg
```

Doing that way, the interpreter will give the options to the environment using the variable `sys.argv`. If we don't set them, we get a segmentation fault. If we want to run the script step by step to study the behavior of the object and methos called, on can be tricky and change by hand the content on `sys.argv`, and then run line by line the content of the file (or using a notebook).

```
import sys
sys.argv += ['--config-file', '/ssd/saigre/feelpp/mor/pyfeelpp-mor/feelpp/mor/opusheat/
→opusheat-heat.cfg']
```

On figure 9.2 is presented the different steps of the pipeline. The last arrow corresponds to what is done in my program `ReducedBasis`.

Here are some functions used in the script :

- `model.initModel()`. This function runs the construction of affine decomposition using the DEIM algorithm, on the matrix A , and on the right-hand side F . Two different decompositions are made. In `pyfeelpp-mor` the functions `<...>DEIM` deal with the right-hand side F , and `<...>MDEIM` deal with the matrix of the bilinear form A . One main advantage of this function is that the generated decomposition is saved. So if the offline generation has already been made, the function just loads the decomposition from files.
- `model.getAffineDecomposition` : returns a tuple `[Aq, Fq]` containing the matrices of the affine decomposition for A , and the vectors for F . The data are stored in a Petsc object, that can be manipulated afterward.

To get an instance of a parameter, we can use these lines

```
Dmu = model.parameterSpace()
mu = Dmu.element(True, False)
```

The first argument (`True`) tells that the parameter is shared with all the processors when the program is run in parallel, and the second (`False`) tells whether or not the parameter is chosen log-randomly in the set of values.

In appendix B is presented the execution of the script step by step.

9.5 Module PETSc4py

`PETSc` [1] (for *Portable, Extensible Toolkit for Scientific Computation*) is a collection of classes, such as vectors or matrices. It provides effective code for the various phases of solving PDEs and can be run in parallel. The library `petsc4py` allows to use the functions of PETSc in a Python script.

Remark 9.4. If we want to run the content of some scripts *by hand* in a python shell, we first have to run those lines, to set the environment of `Feel++` and PETSc and avoid errors.

```
import sys
import feelpp
sys.argv = ['test_feelpp']
e = feelpp.Environment(sys.argv)
```

Here is an example of code solving a linear system using the class KSP. The system to be solved is $Ax = b$. The object pc is a preconditioner on the system.

```
from petsc4py import PETSc
KSP_TYPE = PETSc.KSP.Type.GMRES
PC_TYPE = PETSc.PC.Type.LU

ksp = PETSc.KSP()
ksp.create(PETSc.COMM_SELF)
ksp.setType(KSP_TYPE)
reshist = {}

def monitor(ksp, its, rnorm):
    reshist[its] = rnorm
    print("[petsc4py] Iteration {} Residual {}".format(its,rnorm))

ksp.setMonitor(monitor)
pc = ksp.getPC()
pc.setType(PC_TYPE)
ksp.setOperators(A)
ksp.setConvergenceHistory()
x = b.duplicate()
x.set(0)
ksp.solve(b, x)      # the solution is present in the vector x
```

The function `monitor` allows to display information when the system is solved. On the following example, the system solved is

$$\begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 & 0 \\ 0 & 2 & 1 & 4 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} X = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad (9.5.1)$$

whose solution is $[5 \ -2 \ 1 \ 1 \ 1]^T$. As we use the LU decomposition as preconditioner, the convergence is immediate, but other PC take more steps to converge.

[1]: `ksp.solve(b, x)`

```
[petsc4py] Iteration 0 Residual norm: 5.656854249492381
[petsc4py] Iteration 1 Residual norm: 9.420554752102651e-16
```

[2]: `x[:, :]`

```
[2]: array([ 5., -2.,  1.,  1.,  1.])
```

9.6 Adaptation to PETSc

We presented in section 8.1 the program I developed using NumPy. Now we will present how we adapted it using PETSc vectors and matrices. The main difference is that instead of storing the reduced basis in a matrix of shape (NN, N) , we store each vector PETSc of size NN in a list of length N. The main consequence of this is that we cannot make a simple matrix multiplication.

For instance, listing 7 shows the code used to create the reduced matrices A_N^q for $q \in [1, Q_a]$. All the loops are made only on « small » ranges : the reduced size N or the number of terms in the affine decomposition Q_a .

```

1 self.ANq = []
2 for q in range(self.Qa):
3     self.ANq.append(np.zeros((self.N, self.N)))
4 for i,u in enumerate(self.Z)
5     for j,v in enumerate(self.Z):
6         for q in range(self.Qa):
7             self.ANq[q][i,j] = v.dot( self.Aq[q] * u )

```

Listing 7: Computation of reduced matrices A_N^q

One way to check that the construction of the reduced basis is to skip the orthonormalization of the basis, then for each μ used in the construction we check that the following values are equal.

Indeed, if we say that $N = 1$, for a fixed μ_i from the generation sample, we have $u_N(\mu_i) = u_0 \xi_i$, where ξ_i is the i -th basis function, calculated by the FE method by μ_i , so $u_0 = 1$. More generally, with any N we have $u_N(\mu_i) = [\delta_{ij}]_{1 \leq j \leq N}$

9.7 Description of the code produced

In this chapter, I will present the class `ReducedBasisPETSc` created to solve to work on the problem described in The source code is available in the repository `eye2brain`.

The main principle of this class is that calculs with with vectors (of size \mathcal{N}) are done with PETSc, while online calculs are made with NumPy.

9.7.1 Element of the class

This section lists the elements of the class. They will be used in other sections to compute the reduced basis, the greedy algorithm...

- `self.Aq`, `self.Fq` are the matrices [resp vectors] of the decomposition given by DEIM. Theye are lists of `PETSc.Mat` or `PETSc.Vec` with a size of `self.Qa` or `self.Qf` respectively.
- `self.NN` size of the FE problem, `self.N` size of the reduced basis (quantity updated during the geretation).
- `self.model` of type `ToolboxMor_2D` (or 3D) : model used to generate the DEIM decomposition.
- `self.mubar` parameter used for the energy norm ($\|u\|_X = u^T A_{\bar{\mu}} u$).
- `self.alphaLB` function which returns the coefficient of coercivity fro the bilinear form a , depending on μ .
- `self.Z` is a list repserenting the matrix of the reduced basis. Each element of the list is a `PETSc.Vec` of size `NN`, and the list has a size `N`, see figure 9.3.

$$\text{self.Z} = \left[\begin{bmatrix} \xi_0^0 \\ \xi_1^0 \\ \vdots \\ \xi_N^0 \end{bmatrix}, \begin{bmatrix} \xi_0^1 \\ \xi_1^1 \\ \vdots \\ \xi_N^1 \end{bmatrix}, \dots, \begin{bmatrix} \xi_0^N \\ \xi_1^N \\ \vdots \\ \xi_N^N \end{bmatrix} \right] \xrightarrow[N]{\quad} \text{NN}$$

Figure 9.3: Structure of `self.Z`

- `self.ANQ`, `self.FNp`, lists of sizes `Qa` and `self.Qf` respectively, containing the reduced matrices (of shape (N, N)) and vectors (of size `N`).
- `self.Sp` a list containing `PETSc.Vec` of size `NN`, corresponding to vectors \mathcal{S}^p (see section 8.3), and `self.Lnq` a dictionnary containing the vectors $\mathcal{L}^{n,q}$.
- the three following NumPy matrices contain the offline values for the *a posteriori* error

name	shape	definition
self.SS	(Qf, Qf)	$SS[p, p'] = (\mathcal{S}^p, \mathcal{S}^{p'})_X$
self.SL	(Qf, Qa, N)	$SL[p, n, q] = (\mathcal{S}^p, L^{n,q})_X$
self.LL	(Qa, N, Qa, N)	$LL[n, q, n', q'] = (\mathcal{L}^{n,q}, \mathcal{L}^{n',q'})_X$

- Other elements of the class are used by the library PETSc4py for solving the systems of equations (`self.Abar`, `self.ksp...`)

9.7.2 Methods of the class

This section lists and describes the methods of the class. The methods with a blue bullet are the fonctions that the user of the class will mostly use.

- `__init__(self, Aq, Fq, model, mubar, alphaLB) -> None`, initiates the object. The meaning of the argument is described in the previous section.
- `scalarA(self, u, v) -> float`, computes the scalar product $(u, v)_X = u^T A_\mu v$.
- `normA(self, u) -> float` compute the energy norm of the vector u
- `orthonormalizeZ(self, nb=0) -> None`, use Gram-Schmidt algorithm to orthonormalize the reduced basis `self.Z` (the optionnal argument is not needed, it is a barrier in case the matrix is hard to orthonormalize)
- `assembleA(self, betaA) -> PETSc.Mat` and `assembleF(self, betaF) -> PETSc.Vec`. Assemble the matrice $A = \sum_q^{Q_a} \beta_A^q A^q$ and the vector $F = \sum_p^{Q_f} \beta_F^p F^p$ from a given list of parameters. From a given `ParameterSpaceElement mu`, the lines to get both $A(\mu)$ and $F(\mu)$ are

```

1 [betaA, betaF] = model.computeBetaQm(mu) # rb.model.computeBetaQm(mu)
2 A = rb.assembleA(betaA[0])
3 F = rb.assembleF(betaF[0][0])

```

- `assembleAN(self, beta, size=None)` [resp. `assembleFN(self, beta, size=None)`], assembles the reduced matrix [resp. right-hand side] from a given vector of parameters. The parameter `size` corresponds to size of the sub-basis wanted, the default value is `None`, meaning the whole basis is used.
- `generateZ(self, mus, orth=True) -> None`, generates the base matrix `self.Z`. The argument are the same as in the function `computeOfflineReducedBasis`.
- `test_orth(self) -> bool`, tests is the matrix `slef.Z` is orthonormal.
- `generateANq(self)`, `generateLNp(self)` and `generateFNp(self)`, those three functions computes reduced matrices or vectors $((A_N^q)_{i,j} = \xi_j A^q \xi_i, (L_N^p)_i = (F_N^p)_i = \xi_i \cdot F^p)$.
- `computeOfflineReducedBasis(self, mus, orth=True) -> None`, computes the reduced basis and reduces matrices from a set of parameters.

Argument	Description
<code>mus</code>	list of N <code>ParameterSpaceElement</code> to evalutate offline
<code>orth</code> (default to <code>True</code>)	set the orthonormalization of the basis

- `getSolutionsFE(self, mu)`, computes the finite element solution (problem of size \mathcal{N}). This function retrus the tuple $(u_{\mathcal{N}}, s_{\mathcal{N}})$
- `getSolutions(self, mu, size=None)`, computes the online solution and output (u_N, s_N) . The parameter `size` is the size of the sub-basis used (defaults is `None`, meaning the whole basis is used)
- `computeOfflineErrorRhs(self)`, `computeOfflineError(self)`, compute offline errors. The first function is the error about the right-hand side (\mathcal{S}^p) independant of N , and the second about the basis ($\mathcal{L}^{n,q}$)
- `expandOffline(self)`, add errors to values computed in previous steps. Before this function is called, the last column of Z must be computed.
- `computeOnlineError(self, mu, precalc=None)`, computes the squared *a posteriori* online error $\|\hat{e}(\mu)\|_X^2$, from a parameter μ

Argument	Type	Description
mu	ParameterSpaceElement	parameter μ
precalc = None	dict (optional)	Dict containing the values of <code>betaA</code> , <code>betaF</code> and <code>uN</code> if these values have already been calculated. Defaults to None. If <code>None</code> is given, the quantities are calculated in the function

- `computeDirectError(self, mu, precalc=None)`, computes a *posteriori* error $\|\hat{e}(\mu)\|_X$ using a direct method (costly), from a parameter μ . The parameters of this functions are the same as the previous one.
- `computeEnergyBound(self, mu, precalc=None)`, computes $\frac{\|\hat{e}(\mu)\|_X}{\sqrt{\alpha_{LB}(\mu)}}$
- `compareSols(self, mu)`, compares solutions between reduced basis and finite element method. The function returns the relative error $\frac{\|u_N - u_N\|}{\|u_N\|}$.
- `projFE(self, uN)`, computes the projection of the RB solution (of size N) to the FE space (of size \mathcal{N}).

Remark 9.5. The convention described in figure 9.3 raises an issue : when we want to project the reduced basis solution u_N onto the FE space, we can't apply the formula $W = Z_N \cdot u_N$. This problem is illustrated on figure 9.4. One way to compute this multiplication is to make a loop on \mathcal{N} and sum the terms *by hand*, but this method can be costly (it means makin of loop in Python of size \mathcal{N}). Unfortunately, I have not found yet another way to do it.

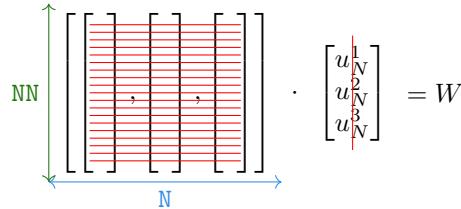


Figure 9.4: Illustration of the issue raised by the storage of the matrix

- `greedy(self, mu_0, Dmu, eps_tol, Nmax)`, runs the Greedy algorithm 2. The functions returns the list of the parameters used to generate the basis.

Argument	Type	Default value	Description
mu_0	ParameterSpaceElement		First parameter to generate the basis
Dmu	list of parameters		Set Ξ_{train} for the algorithm
eps_tol	float	1e-6	Tolerance of the algo
Nmax	int	40	maximal size of the algorithm

The two following functions `saveReducedBasis` and `loadReducedBasis` allow to save the basis once it is generated and load it.

- `saveReducedBasis(self, path, force=False)` save the reduced basis using Pickle. Args: path (str): path of the directory whre data are to be saved force (bool, optional): Force saving, even if files are already present. Defaults to False. """
- `loadReducedBasis(path, model)`, loads the basis from the files saved in the directory path. The argument `model` is the object `ToolboxMor_2D` (or 3D), which is loaded in the global script.

9.7.3 Next steps

There are still some features that can be added to the program. First of all, the time-dependant problem and POD greedy algorithm (3) have not yet been implemented using PETSc objects. Then a bridge with the fmu must be implemented, using FMPy which is a library allowing to simulate FMU (see section 5.2). We began to handle this tool during the internship, on the geometry of the thermal fin problem. The main problem of this library is that the documentation is not extensive, but it provides a Graphical User Interface.

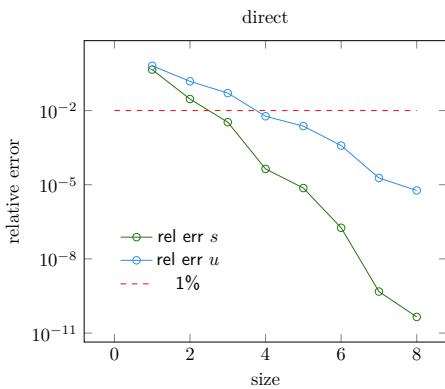


Figure 9.5: Relative error for a fixed parameter

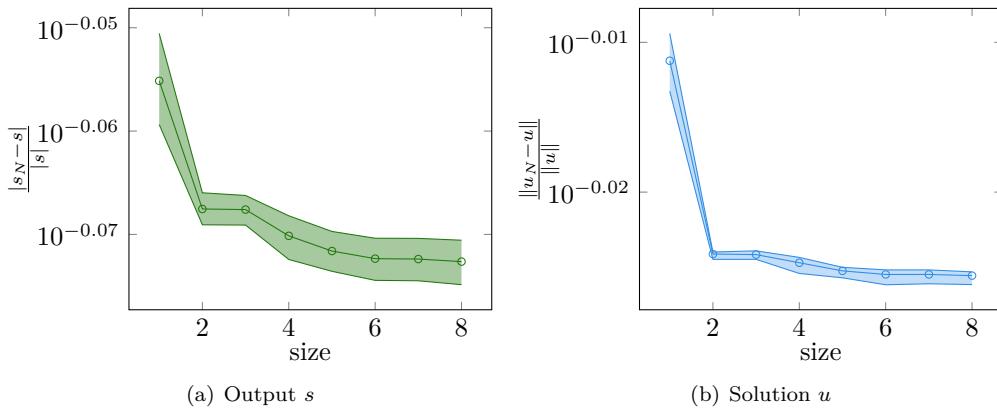


Figure 9.6: Minimum, maximum and mean for relative error

9.8 First results

Let's run the script to see if the code returns right results. We construct a reduced basis of size 8, using the parameters given in the problem² We take the parameter μ defined with those values :

```
mu.setParameters({"Bi":0.1, "k_0":1, "k_1":1.5, "k_2":1.5, "k_3":1.5, "k_4":1.5})
```

The first thing to check is that the generated reduced basis gives results approximately equal to the FE solution. With $\mu = 1.5$, we have :

```
Solution big problem : 1.4889615506729705
Solution reduced problem : 1.4889615506061944
Relative Error 4.484745770818489e-11
```

We can also check the evolution of the error while the reduced basis grows. This result is presented on figure 9.5. We see that the relative error is under 1 % after 4 functions in the basis. But if we try to make the same calculation with many parameters (here 50) taken in the space, we find an odd behavior : the relative error stays stuck above 10 %, as we can see on figure 9.6.

In the quest of what could cause erroneous results, we can check that the matrix generated by the toolbox for the bilinear form is correct, by testing it with simple functions. For the thermal-fin problem, we have

$$a(w, v; \boldsymbol{\mu}) = \sum_{i=0}^4 k^i \int_{\Omega^i} \nabla w \cdot \nabla v + \text{Bi} \int_{\Gamma \setminus \Gamma_{\text{root}}} wv$$

Taking $w = v = 1$, we should find $a(w, u, \boldsymbol{\mu}) = \text{Bi} \text{ meas}(\Gamma \setminus \Gamma_{\text{root}})$. With Feel++, we can measure parts of the geometry. More precisely, $\Gamma \setminus \Gamma_{\text{root}}$ corresponds to the face Γ_{ext} .

²These parameters are {"Bi":0.1, "k_0":1, "k_1":k, "k_2":k, "k_3":k, "k_4":k} for k in [0.1, 10., 0.19179103, 3.94420606, 0.12618569, 0.38535286, 2.15443469, 0.10974988]

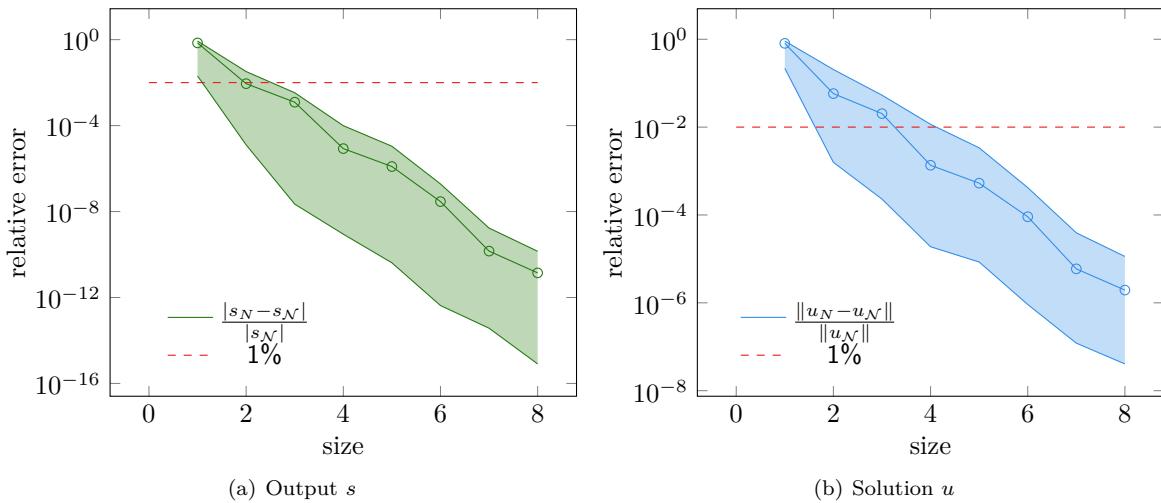


Figure 9.7: Minimum, maximum and mean for relative error

```
[1]: mesh = heatBox.mesh()
      measure(range=feelpp.markedfaces(mesh, "Gamma_ext"))
```

[1]: 47.50000000000003

```
[2]: A = heatBox.assembleMatrix().mat()
      A.assemble()
      F = heatBox.assembleRhs().vec()
      F.set(1)          #  $F = [1, 1, \dots]$ 
      F.assemble()
      F.dot(A * u)    #  $F.T @ A @ F$ 
```

[2]: 47.50000000000037

Actually, the problem was an issue of configuration / construction : in the JSON, the parameter space was defined to have 6 dimensions (k_i for $i \in \llbracket 0, 4 \rrbracket$, and B_i). But on the construction, the parameters used are all on a same 1-dimensional subspace (see footnote 2). This is the reason why the error was high with a parameter outside the subspace and small with a parameter within. To solve this we have two choices : we can change the JSON to make appear only 1 parameter k and set all the other values according to this parameter, or we can construct the reduced basis with parameters in the whole space.

On figure 9.7 the result using the first solution is plotted³. Using the 6 dimensions would give fewer good errors because the parameters are chosen randomly in the space. In the following section, we will focus on a greedy algorithm to chose these parameters.

9.9 Results of the greedy algorithm

In this section, we will use the model described in section 8.1, with coupled parameters : we have $k_0 = 1$, $k_1 = k_2 = k_3 = k_4$ and $\text{Bi} = 0.1$. Hence, the space of parameters has a dimension of 1 (moreover, it would be difficult to plot figures in 6D !). We will also be constant in time.

On figure 9.8, the energy for all parameters of the train set Ξ_{train} is plotted, on a different step of the algorithm. An interactive figure can be found on the repository here. We can figure that the energy globally decreases after each iteration.

The following block of code shows how to call the function running the greedy algorithm. In the log displayed during the execution, the integer shows the current size of the basis, and the float shows the maximum of the energy obtained during the previous step. This maximum has to reach the tolerance to stop the algorithm. In a case where this tolerance is never reached, a maximal size for the basis has been implemented.

³The function called in the script to get such results is `cvgRelError`

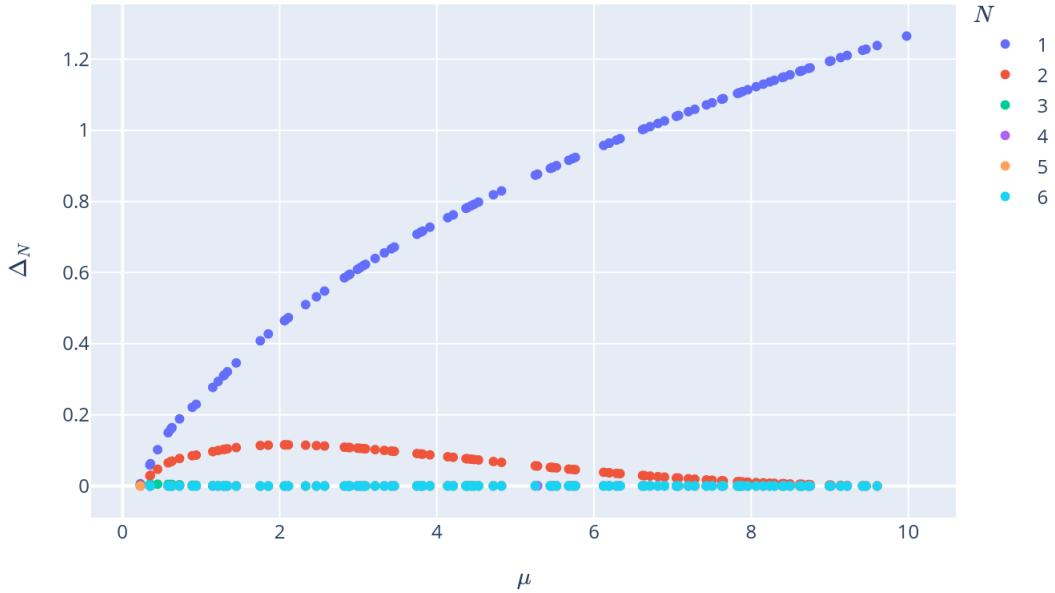


Figure 9.8: Energy on different steps of greedy algorithm

```
[1]: Xi_test = listOfParams(100)
mu0 = Dmu.element(True, False)
S = rb.greedy(mu0, Xi_test)
S
```

```
[reduced basis] Greedy alg. 1 1.2646922390981183
[reduced basis] Greedy alg. 2 0.11555684305275059
[reduced basis] Greedy alg. 3 0.005074361519291918
[reduced basis] Greedy alg. 4 0.000252748470349559
[reduced basis] Greedy alg. 5 2.4362701123146925e-06
[reduced basis] Greedy alg. 6 9.63811004486168e-08
```

```
[2]: [0.19, 9.98, 2.08, 0.447, 5.28, 0.227]
```

Let Ξ_{test} be a set of elements of D . As we did earlier, we measure the relative error on the output $s_N(\mu)$ and the solution $u_N(\mu)$ for all $\mu \in \Xi_{\text{test}}$, and plot the minimum, maximum and mean on the set, for different values of N . Doing this, we can compare the log-random generated basis to the basis computed by the greedy algorithm.

The result is the plot on figure 9.9. We see that there is no such difference between the two methods. We could expect that because the space is « reduced ».

Let's look at the problem where the parameters are in a space of 5 dimensions (all the parameters k_i except k_0 and B_i can vary). This time the greedy-algorithm take more step to give an energy error under the tolerance $\varepsilon = 10^{-6}$: the reduced basis has 42 functions. The figure 9.10 shown on the one hand the evolution of the maximum of the energy as the algorithm runs and on the other hand, the time used by the python function to test all the 1 000 elements of Ξ_{train} to get the best one. We can figure that the more the basis grows, the longer time it takes.

The figure 9.11 gives a comparison of the relative error obtained with this reduced basis (drawn in purple), and a basis of different sizes, generated from random parameters. The computation of the error is made on the same sample Ξ_{test} of size 50. We see that the greedy sampling gives better results than when the basis is chosen randomly. Of course, if we take a random basis of a big size (such as 40 in the figure) we will have better results, but the goal is to keep a small basis.

The computation of Δ_N^{\max} is done on the output. This is why the difference between the two curves is larger for the relative error on output.

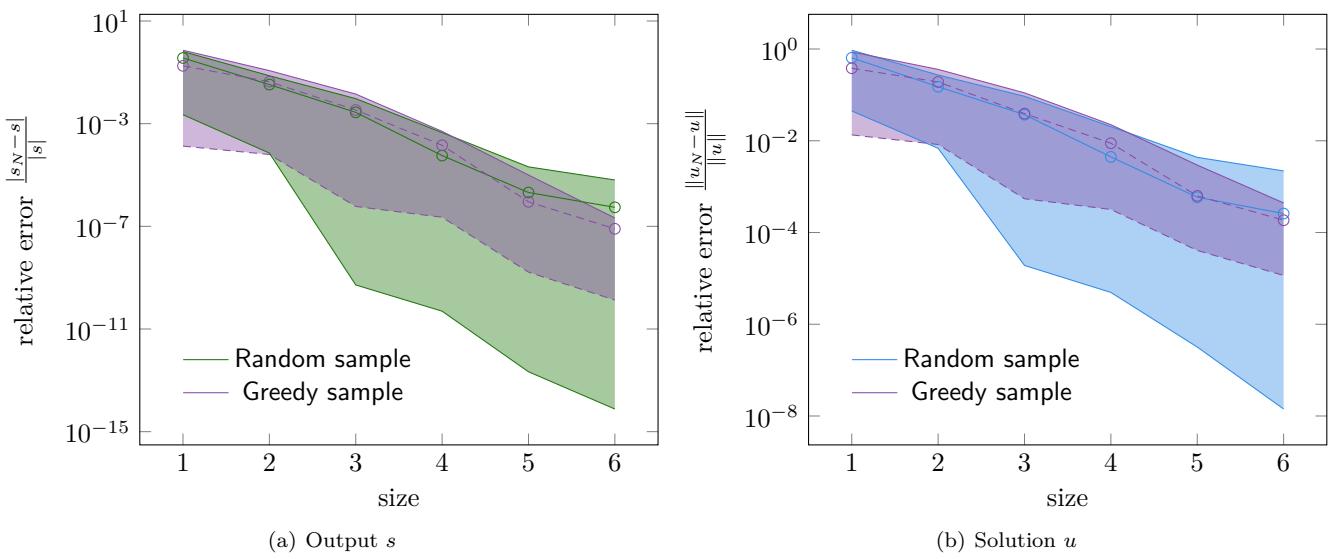
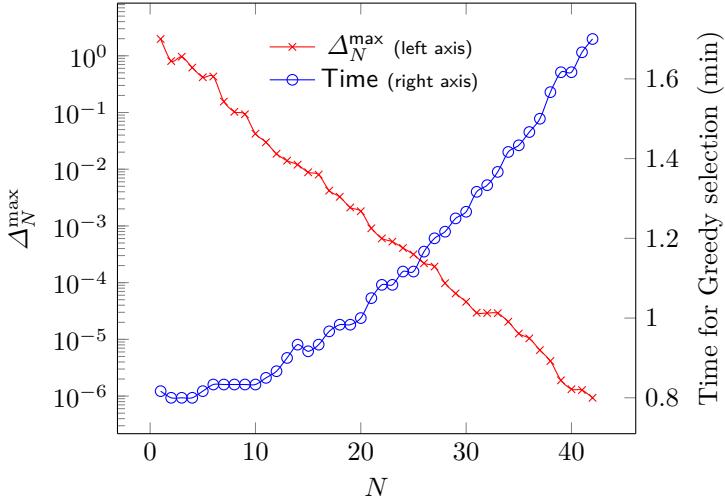
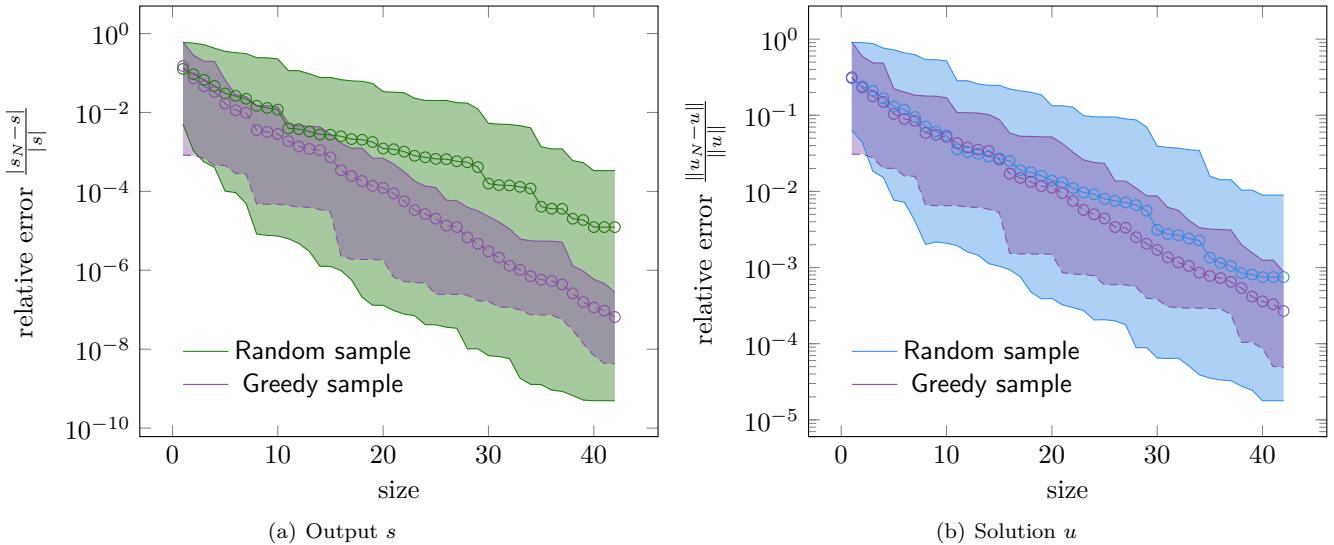
Figure 9.9: Comparaison of relative error when D has a dimension 1

Figure 9.10: Execution of greedy algorithm

Figure 9.11: Comparaison of relative error when D has a dimension 5

10.

Applications

During the internship, the code described in the previous sections was not used on a practical case for Eye2brain. The idea would be in terms to run the physical model described in chapter 4 and more complex model of the eye, and run sensibility analysis to compute Sobol indices. Such a study has already been made on the OMVS [13] using meta-models to compute those indices.

The variables of interest for the computation of those indices can be physiological such as the intraocular pressure (IOP). They can also be geometrical : a patient could have an eye with an elongated shape. The issue raised by such a variable is that the geometry and the mesh of the eye are no longer the same.

Part IV

Appendix

Feel++ is a powerful, expressive, and scalable finite element embedded library in C++. It allows solving partial differential equations with generalized Galerkin methods and export results to visualize them with Paraview. It is developed by Cemosis. *Toolboxes* allow expanding conventional models into multi-physics to solve coupled equations.

Feel++ is available on a Github repository [5].

A.1 Building Feel++

For this internship, we won't use all the libraries that are implemented with Feel++, so there is no need to compile them all. In this section, I will present how to compile just what is needed¹

On Atlas, the compilation is faster if we are on a node of the machine, with many cores. To access one of them, we use the command `salloc` :

```
salloc -t "02:00:00" -p public -J "feelpp" --exclusive -N 1 -n 12 srun --pty ${SHELL}
```

We also need to load a module on Atlas, to have the right libraries to build Feel++ :

```
module load feelpp.profile_gcc830_openmpi402
```

We have to create a `build` directory, where the compiled programs will be, among all the files handled by CMake.

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang -
       -DCMAKE_INSTALL_PREFIX=. -DFEELPP_ENABLE_TOOLBOXES=ON -DFEELPP_ENABLE_MOR=ON -DFEELPP_ENABLE_
       -TESTS=ON -DFEELPP_PARMMG_DOWNLOAD_METIS=ON <path to feelpp cloned repo>
```

The files are organized so that exactly what is needed is compiled. For instance, if we want to compile all the program associated to the toolbox `hdg`, we enter these commands :

```
cd toolboxes/hdg
make [install] -j 12
```

- The `-j 12` option tells the number of cores used to make the compilation.
- The keyword `install` is optional. It tells to CMake to put links to generated application in a directory `install`. This will also be useful to install Python modules, cf section 9.3.

A.2 Handling the software

During the course in S2 and the project-internship made last year, I already needed to handle and test Feel++. In this section, I will recall the principle, and run some examples with the toolbox `hdg`.

To run a simulation with Feel++, we need three files :

- A geometry file, already meshed or not (respectively with the format `msh` / `geo`). The domain shown on figure A.1 is described on listing 8.
- A JSON file, where the configuration of the equation is made : the initial condition and the boundary conditions are given. See listing 9 for an example.
- A configuration file in `cfg` format. This file contains all the global parameter for the simulation : the time-steps, the path to the geometry and the json file... The listing 10 shows such an example.

¹See the installation page for more information

```

1   h=0.1;
2   Point(1) = {0,0,0,h};
3   Point(2) = {0,2,0,h};
4   Point(3) = {2,2,0,h};
5   Point(4) = {2,0,0,h};
6   Line(1) = {1,2};
7   Line(2) = {2,3};
8   Line(3) = {3,4};
9   Line(4) = {4,1};
10  Line Loop(5) = {1,2,3,4};
11  Plane Surface(1) = {5};
12  Physical Surface("omega") = {1};
13  Physical Line("left") = {1};
14  Physical Line("top") = {2};
15  Physical Line("right") = {3};
16  Physical Line("bottom") = {4};

```

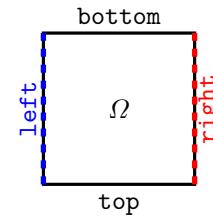


Figure A.1: Geometry described in listing 8

Listing 8: Example of a geo file, see figure A.1

```

1 {
2     "Name": "HDG-Mixed-Poisson",
3     "ShortName": "MP",
4     "Models": {"equations": "hdg"},
5     "Materials":
6     {
7         "omega": { "name": "copper", "cond": "-1" }
8     },
9     "BoundaryConditions":
10    {
11        "potential":
12        {
13            "SourceTerm":
14            {
15                "omega": { "expr": "-sin(Pi*x)*sin(Pi*y):x:y" },
16            },
17            "Neumann":
18            {
19                "top": { "expr": "sin(Pi*x)/(2*Pi):x" },
20                "bottom": { "expr": "-sin(Pi*x)/(2*Pi):x" },
21                "left": { "expr": "-sin(Pi*y)/(2*Pi):y" },
22                "right": { "expr": "sin(Pi*y)/(2*Pi):y" }
23            }
24        }
25    },
26    "Functions":
27    {
28        "u": { "expr": "{0,0}" },
29        "p": { "expr": "1/(2*Pi*Pi)*sin(Pi*x)*sin(Pi*y):x:y" }
30    },
31    "PostProcess":
32    {
33        "Exports": { "fields": ["potential", "flux"] }
34    }
35 }

```

Listing 9: Example of a json file

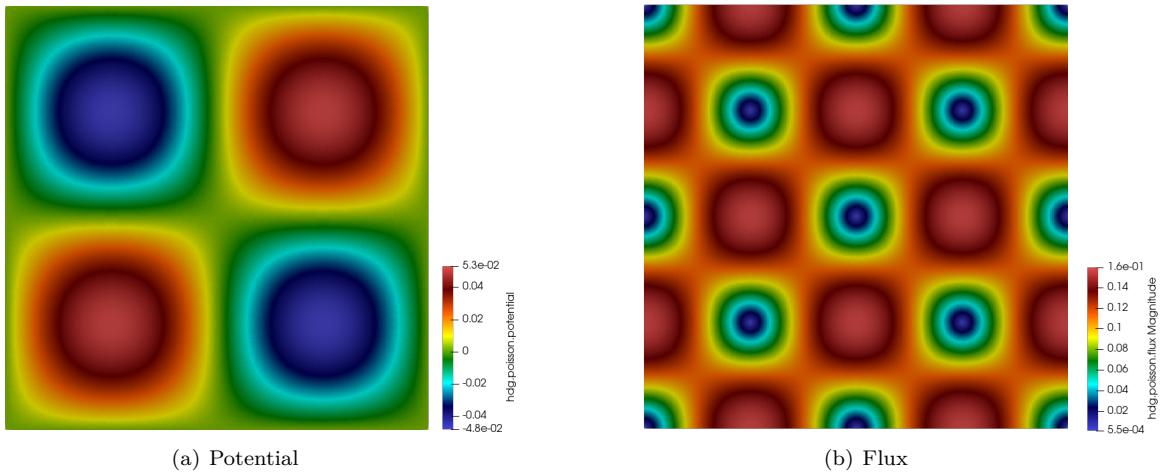


Figure A.2: Result of test HDG Darcy 2D

```

1 directory=toolbox/hdg/testDarcy2D/testDarcy2D
2 case.dimension=2
3 case.discretization=P1
4
5 [hdg.poisson]
6 filename=$cfgdir/testDarcy2D.json
7 [picard]
8 itol=1e-15
9 itmax=5
10
11 [exporter]
12 element-spaces=P0
13 [gmsh]
14 filename=$cfgdir/testDarcy2D.geo
15 hsize=0.01

```

Listing 10: Example of a cfg file

The command to run the simulation is the following :

```
mpirun -np 4 feelpp_toolbox_hdg_poisson --config-file <path to cfg file>
```

`feelpp_toolbox_hdg_poisson` is the name of the toolbox used for the simulation, this application depends on the model we are studying. There are many toolboxes available in Feel++, see the documentation.

We can run Feel++ in parallel, using `mpirun`.

In the `PostProcess` section of the JSON, we can set many fields to export, depending on the toolbox used. In the example above, we choose to export `potential` and `flux`². The application generates files that can be read by Paraview. Those results are given in figure A.2.

A.3 Open the geometry with Paraview

The Salome script returns a mesh in `med` format, this is a binary file that can't be read by ParaView or Feel++. We have to use Gmsh to convert it to a `msh` file. Unfortunately, this file format cannot be opened by Paraview either, but we can use a Feel++ program to convert it : `feelpp_mesh_exporter`³.

```
feelpp_mesh_exporter --gmsh.filename eye-mesh.msh
```

²refer to chapter 6 to see what those quantities represent

³see the manual page.

The result of this operation is a `case` file that can be opened with Paraview. Many pieces of information are present, here we will focus on the one named `marker`, which is simply the number of volume markers in the geometry. The advantage of Paraview is that we can set a camera to follow a trajectory to « record » the geometry. We can also change to opacity or the position of the objects displayed.

Such a video, showing the geometry of the eye with all markers can be found on this link.

A.4 feelpp_mesh_partitioner

When we use the generated mesh in a Feel++ application, such as `feelpp_toolbox_coefficientformpdes` (see section 3.3) on many processors, the application doesn't split the mesh on those cores, all the calculations are made on a single one. To avoid it, we have to partition the mesh before launching the toolbox. We can make it with the application `feelpp_mesh_partitionner`⁴.

For example, with the following command

```
feelpp_mesh_partitioner --part 12 --ifile $HOME/mesh/eye-mesh.msh --odir $HOME/mesh/
→partitionned/
```

the mesh `/home/u2/saigre/mesh/eye-mesh.msh` will be partitionned in 12 submeshes. The corresponding files will be present in the directory `partitionned`. On the command line, we have to give the absolute path for files because the folder where the program is executed is different from the one it is called.

This action results in two files : a binary file (with the `h5` extension), and a JSON file. This last file is the one that we give as an option to the field `mesh.filename` in the config file. To have an idea of the time saved with the partition : without it, the application took more than 2 minutes to run, while the partitioned one took only 25 seconds.

In the next section, we will solve a mathematical model with our geometry to test the mesh : if we get some strange results, this may be the consequence of a problem in the geometry. In chapter 3, simple tests are made on the geometry.

A.5 Pyfeelpp

This feature of Feel++ is still in development. It provides python packages to solve PDEs and uses the toolbox framework, see the documentation. More details on it are given in this report, in section 9.3.

⁴see the manual page.

B.

Script toolboxmor.py

This example runs the opus-heat case.

B.1 Initialisation of the environment

We begin by loading all the modules necessary.

```
[1]: import sys
sys.argv += ['--config-file', '/ssd/saigre/feelpp/mor/pyfeelpp-mor/feelpp/mor/opusheat/
→opusheat-heat.cfg']

[2]: import feelpp
from feelpp.mor import *
from feelpp.toolboxs.heat import *
```

Welcome to the Feel++ Toolboxes

Now we set the Feel++ environment, then we initialize the toolbox and the MOR model.

```
[3]: o = toolboxes_options("heat")
o.add(makeToolboxMorOptions())
e = feelpp.Environment(sys.argv,opts=o)
```

```
[ Starting Feel++ ] application ipython3 version 0.1 date 2021-Jun-20
[feelpp] create Feel++ repository "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb" ...
[feelpp] create Feel++ geo repository "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb/geo" ...
[feelpp] create Feel++ exprs directory "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb./exprs" ...
. ipython3 files are stored in /home/u2/saigre/feel/pyfeelpp-mor/feelppdb./np_1
.. logfiles :/ssd/saigre/build/install/lib/feelppdb./np_1/logs
```

B.2 Compute the offline decomposition

```
[4]: heatBox = heat(dim=2,order=1)
heatBox.init()
```

```
heat(2,1)
[modelProperties] Loading Model Properties : "/ssd/saigre/feelpp/mor/pyfeelpp-mor/feelpp/mor/
→opusheat/opusheat-heat.json"
```

```
[5]: model = toolboxmor_2d()
model.setFunctionSpaces( Vh=heatBox.spaceTemperature() )
```

```
Model repository: /home/u2/saigre/feel/pyfeelpp-mor/feelppdb/crbdb/toolboxmor/
→f49a4d9e-103f-4167-adf3-cd723779c3f3
[modelProperties] Loading Model Properties : "/ssd/saigre/feelpp/mor/pyfeelpp-mor/feelpp/mor/
→opusheat/opusheat-heat.json"
```

In this cell, we set the functions which will be used by the toolbox to assemble the right-hand side (`assembleDEIM`), and the matrix (`assembleMDEIM`).

```
[6]: def assembleDEIM(mu):
    for i in range(0,mu.size()):
        heatBox.addParameterInModelProperties(mu.parameterName(i),mu(i))
```

```

heatBox.updateParameterValues()
return heatBox.assembleRhs()

def assembleMDEIM(mu):
    for i in range(0,mu.size()):
        heatBox.addParameterInModelProperties(mu.parameterName(i),mu(i))
    heatBox.updateParameterValues()
    return heatBox.assembleMatrix()

model.setAssembleDEIM(fct=assembleDEIM)
model.setAssembleMDEIM(fct=assembleMDEIM)

```

Then we initialize the model. The DEIM using the greedy algorithm is run, and the results are saved. In the output, we can see the different steps of the algorithm, with the parameter chosen using a greedy method. As the maximal size is given (40), but in this case, we reach the tolerance after 4 steps for the right-hand side and 5 for the matrix.

[7] : model.initModel()

```

Number of local dof 8441
Number of dof 8441
Model repository: /home/u2/saigre/feel/pyfeelpp-mor/feelppdb/crbdb/toolboxmor/
↳ f49a4d9e-103f-4167-adf3-cd723779c3f3
[modelProperties] Loading Model Properties : "/ssd/saigre/feelpp/mor/pyfeelpp-mor/feelpp/mor/
↳ opusheat/opusheat-heat.json"
DEIMvec0 : No Database loaded : start greedy algorithm from beginning
DEIMvec0 Offline sampling size = 40
DEIMvec0 : Sampling size (=40) smaller than deim.dimension-max (=100), dimension max is now 40
=====
DEIMvec0 : Start algorithm with mu=[0.651348,1.66918,1.42955,987879,901133,5.44408]
    [DEIMvec0 : Add new vector] Time : 0.0675207s
    [DEIMvec0 : compute best fit] Time : 1.20456s
DEIMvec0 : Current max error=0.00424217, Atol=1e-16, relative max error=0.00424217, Rtol=1e-14, u
↳ for mu=[4.89352,1.11482,1.92775,626272,592027,28.0212]
=====
DEIMvec0 : Construction of basis 2/40, with mu=[4.89352,1.11482,1.92775,626272,592027,28.0212]
    [DEIMvec0 : Add new vector] Time : 0.000278964s
    [DEIMvec0 : compute best fit] Time : 0.00842828s
DEIMvec0 : Current max error=0.0029714, Atol=1e-16, relative max error=0.0029714, Rtol=1e-14, u
↳ for mu=[1.46602,2.69517,2.69063,11430.3,354775,5.21863]
=====
DEIMvec0 : Construction of basis 3/40, with mu=[1.46602,2.69517,2.69063,11430.3,354775,5.21863]
    [DEIMvec0 : Add new vector] Time : 0.000267437s
    [DEIMvec0 : compute best fit] Time : 0.00849803s
DEIMvec0 : Current max error=0.00247159, Atol=1e-16, relative max error=0.00247159, Rtol=1e-14, u
↳ for mu=[2.18272,2.76524,2.98617,872604,19092.2,10.6341]
=====
DEIMvec0 : Construction of basis 4/40, with mu=[2.18272,2.76524,2.98617,872604,19092.2,10.6341]
    [DEIMvec0 : Add new vector] Time : 0.000319684s
    [DEIMvec0 : compute best fit] Time : 0.00867022s
DEIMvec0 : Current max error=2.15622e-18, Atol=1e-16, relative max error=2.15622e-18, u
↳ Rtol=1e-14, for mu=[4.07809,2.80587,1.20672,197663,457850,8.9885]
=====
DEIMvec0 : Tolerance reached !
[loadMesh] Loading Gmsh compatible mesh: "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb/np_1/
↳ deimvec0-submesh.msh"
[loadMesh] Loading Gmsh compatible mesh: "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb/np_1/
↳ deimvec0-submesh.msh" done
=====
DEIMvec0 : Stopping greedy algorithm. Number of basis function : 4
    [DEIMvec0 : Offline Total Time] Time : 0.012811s

```

```

Electric DEIM construction finished!!
Model repository: /home/u2/saigre/feel/pyfeelpp-mor/feelppdb/crbdb/toolboxmor/
↳ f49a4d9e-103f-4167-adf3-cd723779c3f3
[modelProperties] Loading Model Properties : "/ssd/saigre/feelpp/mor/pyfeelpp-mor/feelpp/mor/
↳ opusheat/opusheat-heat.json"
MDEIMmat0 : No Database loaded : start greedy algorithm from beginning
MDEIMmat0 Offline sampling size = 40
=====
MDEIMmat0 : Start algorithm with mu=[1.13625,2.57564,1.64121,773238,813727,5.32909]
  [MDEIMmat0 : Add new vector] Time : 0.216194s
  [MDEIMmat0 : compute best fit] Time : 9.37655s
MDEIMmat0 : Current max error=0.749989, Atol=1e-16, relative max error=0.749989, Rtol=1e-12, for
↳ mu=[2.9341,1.01357,2.58331,843025,559995,15.6723]
=====
MDEIMmat0 : Construction of basis 2/20, with mu=[2.9341,1.01357,2.58331,843025,559995,15.6723]
  [MDEIMmat0 : Add new vector] Time : 0.219805s
  [MDEIMmat0 : compute best fit] Time : 9.20379s
MDEIMmat0 : Current max error=0.0891174, Atol=1e-16, relative max error=0.0891174, Rtol=1e-12, for
↳ mu=[0.719976,1.237,1.03514,603668,534815,9.64773]
=====
MDEIMmat0 : Construction of basis 3/20, with mu=[0.719976,1.237,1.03514,603668,534815,9.64773]
  [MDEIMmat0 : Add new vector] Time : 0.230843s
  [MDEIMmat0 : compute best fit] Time : 9.26244s
MDEIMmat0 : Current max error=0.00051939, Atol=1e-16, relative max error=0.00051939, Rtol=1e-12, for
↳ mu=[4.75596,1.60303,1.08847,313124,16233.2,25.3365]
=====
MDEIMmat0 : Construction of basis 4/20, with mu=[4.75596,1.60303,1.08847,313124,16233.2,25.3365]
  [MDEIMmat0 : Add new vector] Time : 0.243057s
  [MDEIMmat0 : compute best fit] Time : 9.29087s
MDEIMmat0 : Current max error=0.000414145, Atol=1e-16, relative max error=0.000414145, Rtol=1e-12, for
↳ mu=[0.269938,1.26076,1.69417,237421,143259,23.809]
=====
MDEIMmat0 : Construction of basis 5/20, with mu=[0.269938,1.26076,1.69417,237421,143259,23.809]
  [MDEIMmat0 : Add new vector] Time : 0.226725s
  [MDEIMmat0 : compute best fit] Time : 9.19958s
MDEIMmat0 : Current max error=5.50548e-16, Atol=1e-16, relative max error=5.50548e-16, Rtol=1e-12, for
↳ mu=[3.6409,2.26667,2.2522,822318,385227,1.32961]
=====
MDEIMmat0 : Tolerance reached !
[loadMesh] Loading Gmsh compatible mesh: "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb/np_1/
↳ mdeimmat0-submesh.msh"
[loadMesh] Loading Gmsh compatible mesh: "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb/np_1/
↳ mdeimmat0-submesh.msh" done
=====
MDEIMmat0 : Stopping greedy algorithm. Number of basis function : 5
  [MDEIMmat0 : Offline Total Time] Time : 0.0128268s
Electric MDEIM construction finished!

```

If we run again the previous cell, as the algorithm has already been fulfilled, the model is loaded from the files saved.

```

Number of local dof 8441
Number of dof 8441
Model repository: /home/u2/saigre/feel/pyfeelpp-mor/feelppdb/crbdb/toolboxmor/
↳ f49a4d9e-103f-4167-adf3-cd723779c3f3
[modelProperties] Loading Model Properties : "/ssd/saigre/feelpp/mor/pyfeelpp-mor/feelpp/mor/
↳ opusheat/opusheat-heat.json"
[loadMesh] Loading Gmsh compatible mesh: "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb/crbdb/
↳ toolboxmor/f49a4d9e-103f-4167-adf3-cd723779c3f3/deimvec/deimvec0-submesh.msh"
[loadMesh] Loading Gmsh compatible mesh: "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb/crbdb/
↳ toolboxmor/f49a4d9e-103f-4167-adf3-cd723779c3f3/deimvec/deimvec0-submesh.msh" done
DEIMvec0 : Database loaded with 4 basis functions

```

```

DEIMvec0 : Start reassambling 4 basis vectors
DEIMvec0 : reassemble for mu=[0.651348,1.66918,1.42955,987879,901133,5.44408]
DEIMvec0 : reassemble for mu=[4.89352,1.11482,1.92775,626272,592027,28.0212]
DEIMvec0 : reassemble for mu=[1.46602,2.69517,2.69063,11430.3,354775,5.21863]
DEIMvec0 : reassemble for mu=[2.18272,2.76524,2.98617,872604,19092.2,10.6341]
=====
DEIMvec0 : Stopping greedy algorithm. Number of basis function : 4
    [DEIMvec0 : Reassemble 4 basis] Time : 0.199402s
Electric DEIM construction finished!!
Model repository: /home/u2/saigre/feel/pyfeelpp-mor/feelppdb/crbdb/toolboxmor/
↳ f49a4d9e-103f-4167-adf3-cd723779c3f3
[modelProperties] Loading Model Properties : "/ssd/saigre/feelpp/mor/pyfeelpp-mor/feelpp/mor/
↳ opusheat/opusheat-heat.json"
[loadMesh] Loading Gmsh compatible mesh: "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb/crbdb/
↳ toolboxmor/f49a4d9e-103f-4167-adf3-cd723779c3f3/deimmat/mdeimmat0-submesh.msh"
[loadMesh] Loading Gmsh compatible mesh: "/home/u2/saigre/feel/pyfeelpp-mor/feelppdb/crbdb/
↳ toolboxmor/f49a4d9e-103f-4167-adf3-cd723779c3f3/deimmat/mdeimmat0-submesh.msh" done
MDEIMmat0 : Database loaded with 5 basis functions
MDEIMmat0 : Start reassambling 5 basis vectors
MDEIMmat0 : reassemble for mu=[1.13625,2.57564,1.64121,773238,813727,5.32909]
MDEIMmat0 : reassemble for mu=[2.9341,1.01357,2.58331,843025,559995,15.6723]
MDEIMmat0 : reassemble for mu=[0.719976,1.237,1.03514,603668,534815,9.64773]
MDEIMmat0 : reassemble for mu=[4.75596,1.60303,1.08847,313124,16233.2,25.3365]
MDEIMmat0 : reassemble for mu=[0.269938,1.26076,1.69417,237421,143259,23.809]
=====
MDEIMmat0 : Stopping greedy algorithm. Number of basis function : 5
    [MDEIMmat0 : Reassemble 5 basis] Time : 1.27214s
Electric MDEIM construction finished!!

```

```
[8]: heatBoxDEIM = heat(dim=2,order=1)
meshDEIM = model.getDEIMReducedMesh()
heatBoxDEIM.setMesh(meshDEIM)
heatBoxDEIM.init()
```

Now we create a heat toolbox, using the reduced basis created previously.

```
heat(2,1)
[modelProperties] Loading Model Properties : "/ssd/saigre/feelpp/mor/pyfeelpp-mor/feelpp/mor/
↳ opusheat/opusheat-heat.json"
```

Then, as earlier, we set the function to assemble from a parameter. This time the function will make online computations. This cell is for the right-hand side.

```
[9]: def assembleOnlineDEIM(mu):
    for i in range(0,mu.size()):
        heatBoxDEIM.addParameterInModelProperties(mu.parameterName(i),mu(i))
    heatBoxDEIM.updateParameterValues()
    return heatBoxDEIM.assembleRhs()

model.setOnlineAssembleDEIM(assembleOnlineDEIM)
```

And we make the same step for the matrix.

```
[10]: heatBoxMDEIM=heat(dim=2,order=1)
meshMDEIM = model.getMDEIMReducedMesh()
heatBoxMDEIM.setMesh(meshMDEIM)
heatBoxMDEIM.init()

def assembleOnlineMDEIM(mu):
    for i in range(0,mu.size()):
        heatBoxMDEIM.addParameterInModelProperties(mu.parameterName(i),mu(i))
    heatBoxMDEIM.updateParameterValues()
```

```

    return heatBoxMDEIM.assembleMatrix()

model.setOnlineAssembleMDEIM(assembleOnlineMDEIM)

```

Finally, we post-initialize the model.

```
[11]: model.postInitModel()
model.setInitialized(True)
```

B.3 Online computations

Here we get the matrixes and right-hand sides from the decomposition.

```
[12]: [Aq, Fq] = model.getAffineDecomposition()
```

Remark B.1. This feature has been added later in the module (see commit 94dc18f0) : if the model is time-dependant, we can get the mass matrix with the same function :

```
[12]: [Mq, Aq, Fq] = model.getAffineDecomposition()
```

```
[13]: Aq
```

```
[13]: [[<feelpp._alg.MatrixPetscDouble object at 0x7f54186cdc70>,
<feelpp._alg.MatrixPetscDouble object at 0x7f5418727270>,
<feelpp._alg.MatrixPetscDouble object at 0x7f54187329f0>,
<feelpp._alg.MatrixPetscDouble object at 0x7f54186bc070>,
<feelpp._alg.MatrixPetscDouble object at 0x7f54186f3730>]]
```

```
[14]: Fq
```

```
[14]: [[[<feelpp._alg.VectorPetscDouble object at 0x7f541879af30>,
<feelpp._alg.VectorPetscDouble object at 0x7f541870e8f0>,
<feelpp._alg.VectorPetscDouble object at 0x7f541870ecb0>,
<feelpp._alg.VectorPetscDouble object at 0x7f541870ec70>]]]
```

These matrices and vectors can be converted to `petsc4py.Mat` and `petsc4py.Vec` to be used by PETSc functions.
In this cell, we initialize a parameter from the model and print its values.

```
[15]: Dmu = model.parameterSpace()
mu = Dmu.element(True, False)
print("mu =", mu)
```

```
mu = [4.37e+00, 2.92e+00, 1.17e+00, 8.75e+05, 5.28e+05, 2.83e+01]
```

Now we get the decomposition of $A(\mu)$ and $F(\mu)$: $A(\mu) = \sum_{q=1}^{Q_A} \beta_q^A(\mu) A_q$, $F(\mu) = \sum_{q'=1}^{Q_F} \beta_{q'}^F(\mu) F_{q'}$

```
[16]: [betaA, betaF] = model.computeBetaQm(mu)
print("betaA =", betaA)
print("betaF =", betaF)
```

```
betaA = [[11.719017613745706, -2.7795543208276716, 0.0827590509751109, 0.005061081396828813, 0.
         ↵0014525863914981914]]
betaF = [[[300.0, 1.1159318846864676, -0.20075874676301833, 0.20146295707845197]]]
```


Bibliography

- [1] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. McInnes, R. Mills, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, and H. Zhang. Petsc users manual: Revision 3.11. Technical Report ANL-95/11 Rev 3.11., Argonne National Laboratory, Mar 2019.
- [2] Silvia Bertoluzza, Giovanna Guidoboni, Romain Hild, Daniele Prada, Christophe Prud'homme, Riccardo Sacco, Lorenzo Sala, and Marcela Szopos. A HDG method for elliptic problems with integral boundary condition: Theory and Applications. , 2021.
- [3] Lucia Carichino, Giovanna Guidoboni, and Marcela Szopos. Energy-based operator splitting approach for the time discretization of coupled systems of partial and ordinary differential equations for fluid flows: The Stokes case. *Journal of Computational Physics*, 364:235–256, 2018.
- [4] Lilian Cheraifi. Private Communication, 2021.
- [5] Feel++ Consortium. Finite Element Embedded Library in C++. <https://github.com/feelpp/feelpp>, 2011 (in development).
- [6] Giovanna Guidoboni, Alon Harris, and Riccardo Sacco. *Ocular Fluid Dynamics Anatomy, Physiology, Imaging Techniques, and Mathematical Modeling: Anatomy, Physiology, Imaging Techniques, and Mathematical Modeling*. Birkhäuser, 01 2019.
- [7] Romain Hild. *Optimization and control of high fields magnets*. Theses, Université de Strasbourg, November 2020.
- [8] Romain Hild, Christophe Prud'homme, Lorenzo Sala, and Marcela Szopos. Private Communication, 2021.
- [9] Sarra Madani and Hanane Ouachour. Eye2brain, 2021. 3rd semester project report, Université de Strasbourg.
- [10] Ean-Hin Ooi and Eddie Yin-Kwee Ng. Simulation of aqueous humor hydrodynamics in human eye heat transfer. *Computers in Biology and Medicine*, 38(2):252–262, 2008.
- [11] Daniele Prada, Alon Harris, Giovanna Guidoboni, Brent Siesky, Amelia M. Huang, and Julia Arciero. Autoregulation and neurovascular coupling in the optic nerve head. *Survey of Ophthalmology*, 61(2):164–186, 2016.
- [12] Christophe Prud'homme. Calcul Scientifique 3, Fall Semester 2020 – 2021.
- [13] Christophe Prud'homme, Lorenzo Sala, and Marcela Szopos. Uncertainty propagation and sensitivity analysis: results from the Ocular Mathematical Virtual Simulator. *Mathematical Biosciences and Engineering*, 18(3):2010–2032, 2021.
- [14] Christophe Prud'homme and Marcela Szopos. Private Communication, 2021.
- [15] Pybind. pybind11. <https://pybind11.readthedocs.io/en/stable/>, 2017.
- [16] Lorenzo Sala. Eye2brain. <http://www.cemosis.fr/projects/eye2brain/>, 2016.
- [17] Lorenzo Sala. *Mathematical modelling and simulation of ocular blood flows and their interactions*. Theses, Université de Strasbourg, September 2019.
- [18] Christophe Trophime. Private Communication, 2021.