A generic way to solve partial differential equations by the $\mathbb{P}^1$-Lagrange finite element method in vector languages

François Cuvelier Gilles Scarella

2015/11/16

Contents

1 Introduction

We present a simple and efficient method to write short codes in vector languages for solving boundary value problem (BVP) by $\mathbb{P}^1$-Lagrange finite element methods in any space dimension. The objective is not to produce the most efficient code but to obtain some short, simple, readable, efficient and generic code to use. Each algorithm of the report may be solved using the toolbox. The following codes are written in a vector language and they are easily transposable to a high number of languages. Full codes including the techniques presented in this paper have been made available in Matlab/Octave and Python.

In Section 2 the general problems are described and notations are given. In Section 3 some examples on how to use the toolbox are given, in several vector languages (Matlab, Octave, Python). In Section 4 the data structures are presented. In Sections 6 and 7, the discretisations to solve scalar and vector BVP's are explained. In Section 8 the performance of the classical assembly method is given. In Section 9, vectorization is explained. Section 10 shows some performance results using the vectorization.

2 Description of the generic problems

The notations of [12] are employed in this section and extended to the vector case.

2.1 Scalar boundary value problem

Let Ω be a bounded open subset of $\mathbb{R}^d$, $d \geqslant 1$. The boundary of Ω is denoted by Γ.

We denote by $\mathcal{L}_{\mathbb{A},\boldsymbol{b},\boldsymbol{c},a_0} = \mathcal{L} : \mathrm{H}^2(\Omega) \longrightarrow L^2(\Omega)$ the second order linear differential operator acting on *scalar fields* defined, $\forall u \in \mathrm{H}^2(\Omega)$, by

$$\mathcal{L}_{\mathbb{A},\boldsymbol{b},\boldsymbol{c},a_0}(u) \stackrel{\text{def}}{=} -\operatorname{div}(\mathbb{A} \nabla u) + \operatorname{div}(\boldsymbol{b}u) + \langle \nabla u, \boldsymbol{c} \rangle + a_0 u \tag{2.1}$$

where $\mathbb{A} \in (L^\infty(\Omega))^{d\times d}$, $\boldsymbol{b} \in (L^\infty(\Omega))^d$, $\boldsymbol{c} \in (L^\infty(\Omega))^d$ and $a_0 \in L^\infty(\Omega)$ are given functions and $\langle \cdot, \cdot \rangle$ is the usual scalar product in $\mathbb{R}^d$. We use the same notations as in the chapter 6 of [12] and we note that we can omit either $\operatorname{div}(\boldsymbol{b}u)$ or $\langle \nabla u, \boldsymbol{c} \rangle$ if $\boldsymbol{b}$ and $\boldsymbol{c}$ are sufficiently regular functions. We keep both terms with $\boldsymbol{b}$ and $\boldsymbol{c}$ to deal with more boundary conditions. It should be also noted that it is important to preserve the two terms $\boldsymbol{b}$ and $\boldsymbol{c}$ in the generic formulation to enable a greater flexibility in the choice of the boundary conditions.

Let Γ^D, Γ^R be open subsets of Γ, possibly empty and $f \in L^2(\Omega)$, $g^D \in \mathrm{H}^{1/2}(\Gamma^D)$, $g^R \in L^2(\Gamma^R)$, $a^R \in L^\infty(\Gamma^R)$ be given data.

A *scalar* boundary value problem is given by

Scalar BVP 1 : generic problem

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$\mathcal{L}(u) = f \quad \text{in } \Omega, \tag{2.2}$$

$$u = g^D \quad \text{on } \Gamma^D, \tag{2.3}$$

$$\frac{\partial u}{\partial n_{\mathcal{L}}} + a^R u = g^R \quad \text{on } \Gamma^R. \tag{2.4}$$

The **conormal derivative** of u is defined by

$$\frac{\partial u}{\partial n_{\mathcal{L}}} \stackrel{\text{def}}{=} \langle \mathbb{A} \nabla u, \boldsymbol{n} \rangle - \langle \boldsymbol{b}u, \boldsymbol{n} \rangle \tag{2.5}$$

The boundary conditions (2.3) and (2.4) are respectively **Dirichlet** and **Robin** boundary conditions. **Neumann** boundary conditions are particular Robin boundary conditions with $a^R \equiv 0$.

2.2 Vector boundary value problem

Let $m \geqslant 1$ and $\mathcal{H}$ be the m-by-m matrix of second order linear differential operators defined by

$$\left\{ \begin{array}{rcl} \mathcal{H} : \left(\mathrm{H}^2(\Omega)\right)^m & \longrightarrow & \left(L^2(\Omega)\right)^m \\ \boldsymbol{u} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m) & \longmapsto & \boldsymbol{f} = (\boldsymbol{f}_1, \ldots, \boldsymbol{f}_m) \stackrel{\text{def}}{=} \mathcal{H}(\boldsymbol{u}) \end{array} \right. \tag{2.6}$$

where

$$\boldsymbol{f}_\alpha = \sum_{\beta=1}^{m} \mathcal{H}_{\alpha,\beta}(\boldsymbol{u}_\beta), \quad \forall \alpha \in [\![1, m]\!], \tag{2.7}$$

with, for all $(\alpha, \beta) \in [\![1, m]\!]^2$,

$$\mathcal{H}_{\alpha,\beta} \stackrel{\text{def}}{=} \mathcal{L}_{\mathbb{A}^{\alpha,\beta},\boldsymbol{b}^{\alpha,\beta},\boldsymbol{c}^{\alpha,\beta},a_0^{\alpha,\beta}} \tag{2.8}$$

and $\mathbb{A}^{\alpha,\beta} \in (L^\infty(\Omega))^{d\times d}$, $\boldsymbol{b}^{\alpha,\beta} \in (L^\infty(\Omega))^d$, $\boldsymbol{c}^{\alpha,\beta} \in (L^\infty(\Omega))^d$ and $a_0^{\alpha,\beta} \in L^\infty(\Omega)$ are given functions. We can also write in matrix form

$$\mathcal{H}(\boldsymbol{u}) = \begin{pmatrix} \mathcal{L}_{\mathbb{A}^{1,1},\boldsymbol{b}^{1,1},\boldsymbol{c}^{1,1},a_0^{1,1}} & \dots & \mathcal{L}_{\mathbb{A}^{1,m},\boldsymbol{b}^{1,m},\boldsymbol{c}^{1,m},a_0^{1,m}} \\ \vdots & \ddots & \vdots \\ \mathcal{L}_{\mathbb{A}^{m,1},\boldsymbol{b}^{m,1},\boldsymbol{c}^{m,1},a_0^{m,1}} & \dots & \mathcal{L}_{\mathbb{A}^{m,m},\boldsymbol{b}^{m,m},\boldsymbol{c}^{m,m},a_0^{m,m}} \end{pmatrix} \begin{pmatrix} \boldsymbol{u}_1 \\ \vdots \\ \boldsymbol{u}_m \end{pmatrix}. \tag{2.9}$$

We remark that the $\mathcal{H}$ operator for $m = 1$ is equivalent to the $\mathcal{L}$ operator.

For $\alpha \in [\![1, m]\!]$, we define Γ_α^D and Γ_α^R as open subsets of Γ, possibly empty, such that $\Gamma_\alpha^D \cap \Gamma_\alpha^R = \varnothing$. Let $\boldsymbol{f} \in (L^2(\Omega))^m$, $g_\alpha^D \in \mathrm{H}^{1/2}(\Gamma_\alpha^D)$, $g_\alpha^R \in L^2(\Gamma_\alpha^R)$, $a_\alpha^R \in L^\infty(\Gamma_\alpha^R)$ be given data.

A *vector* boundary value problem is given by

Vector BVP 1 : generic problem

Find $\boldsymbol{u} = (\boldsymbol{u}_1, \dots, \boldsymbol{u}_m) \in (\mathrm{H}^2(\Omega))^m$ such that

$$\mathcal{H}(\boldsymbol{u}) = \boldsymbol{f} \qquad \text{in } \Omega, \tag{2.10}$$

$$\boldsymbol{u}_\alpha = g_\alpha^D \qquad \text{on } \Gamma_\alpha^D,\ \forall \alpha \in [\![1, m]\!], \tag{2.11}$$

$$\frac{\partial \boldsymbol{u}}{\partial n_{\mathcal{H}_\alpha}} + a_\alpha^R \boldsymbol{u}_\alpha = g_\alpha^R \qquad \text{on } \Gamma_\alpha^R,\ \forall \alpha \in [\![1, m]\!], \tag{2.12}$$

where the α-th component of the **conormal derivative** of $\boldsymbol{u}$ is defined by

$$\frac{\partial \boldsymbol{u}}{\partial n_{\mathcal{H}_\alpha}} \overset{\text{def}}{=} \sum_{\beta=1}^m \frac{\partial \boldsymbol{u}_\beta}{\partial n_{\mathcal{H}_{\alpha,\beta}}} = \sum_{\beta=1}^m \left(\langle \mathbb{A}^{\alpha,\beta} \nabla \boldsymbol{u}_\beta, \boldsymbol{n} \rangle - \langle \boldsymbol{b}^{\alpha,\beta} \boldsymbol{u}_\beta, \boldsymbol{n} \rangle \right). \tag{2.13}$$

The boundary conditions (2.12) are the **Robin** boundary conditions and (2.11) is the **Dirichlet** boundary condition. The **Neumann** boundary conditions are particular Robin boundary conditions with $a_\alpha^R \equiv 0$.

In this problem, we may consider on a given boundary some conditions which can vary depending on the component. For example we may have a Robin boundary condition satisfying $\frac{\partial \boldsymbol{u}}{\partial n_{\mathcal{H}_1}} + a_1^R \boldsymbol{u}_1 = g_1^R$ and a Dirichlet one with $\boldsymbol{u}_2 = g_2^D$.In the following of the report we will solve by a $\mathbb{P}^1$-Lagrange finite element method *scalar* BVP (2.2) to (2.4) and *vector* BVP (2.10) to (2.12) without additional restrictive assumption.

3 Examples

In this section, most commonly used functions are presented. They will be explained in details in Sections 6 and 7. Then we solve some PDE's starting from Poisson problem with mixed boundary conditions up to elasticity problems.

We suppose that the mesh generation software used (either FreeFEM++ or gmsh) enables to label the parts of the boundary of Ω. Each part of the boundary is denoted by Γ_{label}.

3.1 First level functions or commonly used functions

We briefly describe the main functions that will be used in the sequel.

$\mathcal{T}_h \leftarrow$ **getMesh**(`FileName`) : to define the mesh $\mathcal{T}_h$ by reading a 2d or 3d mesh from the file `FileName`.

$\mathcal{T}_h \leftarrow$ **HyperCube**(d, N) : to define the mesh $\mathcal{T}_h$ as the unit hypercube $[0,1]^d$. There are N points in each direction and the mesh of the hypercube contains N^d points.

`Lop` $\leftarrow$ **Loperator**$(d, \mathbb{A}, \boldsymbol{b}, \boldsymbol{c}, a_0)$: to initialize the operator $\mathcal{L}$ in dimension d given by (2.1) : `Lop` $\leftarrow \mathcal{L}_{\mathbb{A},\boldsymbol{b},\boldsymbol{c},a_0}$.

`Hop` $\leftarrow$ **Hopoperator**(d, m) : to initialize the operator $\mathcal{H}$ given by (2.6) verifying $\mathcal{H}_{\alpha,\beta} = 0,\ \forall(\alpha,\beta) \in [\![1,m]\!]^2$. Each operator $\mathcal{H}_{\alpha,\beta}$ corresponds to `Hop.H`(α,β) and can be initialized by the function Loperator

PDE $\leftarrow$ **initPDE**(`Op`, $\mathcal{T}_h$) : to initialize a PDE structure fron an operator (either $\mathcal{L}$-operator or $\mathcal{H}$-operator) and a mesh. Default boundary conditions are homogeneous generalized Neumann.

PDE $\leftarrow$ **setBC_PDE**(PDE, `label`, `comps`, `type`, `g`, `ar`) : to define or modify the boundary conditions on the boundary $\Gamma_{\texttt{label}}$ on the mesh PDE.$\mathcal{T}_h$ for components of index `comps` (in the scalar case `comps` $\equiv 1$). For a scalar PDE, we have for example

- Dirichlet condition : $\boldsymbol{u}_2 = g \ \ on\ \Gamma_{11}$, then
  PDE $\leftarrow$ setBC_PDE(PDE, 11, 2, `'Dirichlet'`, $g, \varnothing$)
- generalized Neumann condition : $\frac{\partial \boldsymbol{u}}{\partial n_{\mathcal{H}_3}} = g \ \ on\ \Gamma_{12}$, then
  PDE $\leftarrow$ setBC_PDE(PDE, 12, 3, `'Neumann'`, $g, \varnothing$)
- generalized Robin condition : $\frac{\partial \boldsymbol{u}}{\partial n_{\mathcal{H}_2}} + a_2^R \boldsymbol{u}_2 = g \ \ on\ \Gamma_{13}$, then
  PDE $\leftarrow$ setBC_PDE(PDE, 13, 2, `'Robin'`, g, a_2^R)

$\boldsymbol{x} \leftarrow$ **SolvePDE**(PDE) : to solve by $\mathbb{P}^1$ Lagrange finite element method the partial differential equation defined by the structure PDE. This function returns the solution $\boldsymbol{x}$

3.2 Scalar case

3.2.1 Poisson PDE with mixed boundary conditions in 2D

We first consider the classical 2D Poisson problem with various boundary conditions. The problem to solve is the following

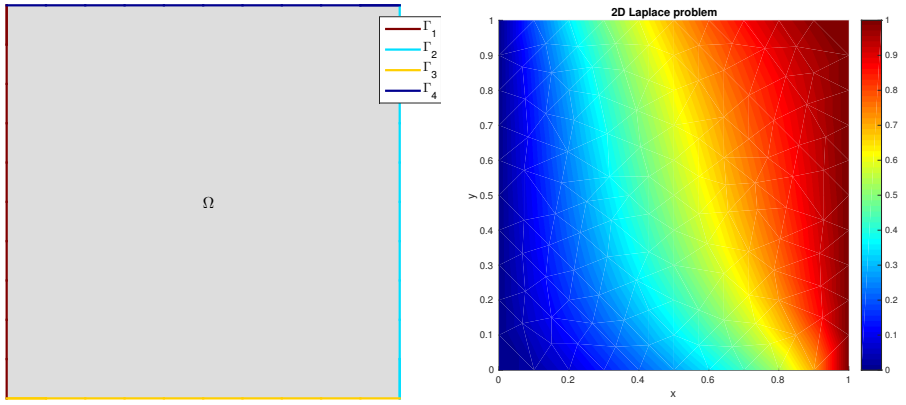
2D Poisson problem

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$
\begin{aligned}
-\Delta u &= f \quad \text{in } \Omega \subset \mathbb{R}^2, && (3.1)\\
u &= 0 \quad \text{on } \Gamma_1, && (3.2)\\
u &= 1 \quad \text{on } \Gamma_2, && (3.3)\\
\frac{\partial u}{\partial n} + a_R\, u &= -0.5 \quad \text{on } \Gamma_3, && (3.4)\\
\frac{\partial u}{\partial n} &= 0.5 \quad \text{on } \Gamma_4 && (3.5)
\end{aligned}
$$

where $\Omega = [0,1]^2$ and its boundaries are given in Figure 1a. f and a_R satisfy:

$$
\begin{aligned}
f(\boldsymbol{x}) &= \cos(\boldsymbol{x}_1 + \boldsymbol{x}_2) \quad \forall \boldsymbol{x} \in \Omega\\
a_R(\boldsymbol{x}) &= 1 + \boldsymbol{x}_1^2 + \boldsymbol{x}_2^2 \quad \forall \boldsymbol{x} \in \Omega
\end{aligned}
$$



(a) Domain and boundaries (b) Result

Figure 1: 2D Poisson problem

The operator in (3.1) is the *Stiffness* operator : $\mathcal{L}_{\mathbb{I},\mathbf{0},\mathbf{0},0}$.
The conormal derivative $\frac{\partial u}{\partial n_{\mathcal{L}}}$ is

$$\frac{\partial u}{\partial n_{\mathcal{L}}} := \langle \mathbb{A} \nabla u, \boldsymbol{n} \rangle - \langle \boldsymbol{b} u, \boldsymbol{n} \rangle = \frac{\partial u}{\partial n}.$$

The algorithm using the toolbox for solving (3.1)-(3.5) is given in Algorithm 3.1. The corresponding Matlab/Octave and Python codes are given in Listing 1.

Algorithm 3.1 2D Poisson problem

1: $\mathcal{T}_h \leftarrow$ HyperCube$(2, 50)$ ▷ Build unit square mesh
2: Dop $\leftarrow$ Loperator$(2, \mathbb{I}, \mathbf{0}, \mathbf{0}, 0)$ ▷ Stiffness operator
3: PDE $\leftarrow$ initPDE$(\texttt{Dop}, \mathcal{T}_h)$
4: PDE $\leftarrow$ setBC_PDE$(\text{PDE}, 1, 1, \texttt{'Dirichlet'}, 0., \varnothing)$ ▷ $u = 0$ on Γ_1
5: PDE $\leftarrow$ setBC_PDE$(\text{PDE}, 2, 1, \texttt{'Dirichlet'}, 1., \varnothing)$ ▷ $u = 1$ on Γ_2
6: PDE $\leftarrow$ setBC_PDE$(\text{PDE}, 3, 1, \texttt{'Robin'}, -0.5, \boldsymbol{x} \rightarrow 1 + \boldsymbol{x}_1^2 + \boldsymbol{x}_2^2)$ ▷ $\frac{\partial u}{\partial n} + a_R\, u = -0.5$ on Γ_3
7: PDE $\leftarrow$ setBC_PDE$(\text{PDE}, 4, 1, \texttt{'Neumann'}, 0.5, \varnothing)$ ▷ $\frac{\partial u}{\partial n} = 0.5$ on Γ_4
8: PDE.$f \leftarrow (\boldsymbol{x}_1, \boldsymbol{x}_2) \mapsto \cos(\boldsymbol{x}_1 + \boldsymbol{x}_2)$
9: $\boldsymbol{u}_h \leftarrow$ SolvePDE(PDE)

A numerical solution for a given mesh is shown on Figure 1b

```
fprintf('1. Creating the mesh\n');
d=2;Th=HyperCube(d,50);
fprintf('2. Definition of the BVP\n');
LOp=Loperator(d,{1,0;0,1},[],[],[]);
PDE=initPDE(LOp,Th);
PDE=setBC_PDE(PDE,1,1,'Dirichlet',0);
PDE=setBC_PDE(PDE,2,1,'Dirichlet',1);
PDE=setBC_PDE(PDE,3,1,'Robin',-0.5 , ...
              @(x1,x2) 1+x1.^2+x2.^2 );
PDE=setBC_PDE(PDE,4,1,'Neumann',0.5);
PDE.f=@(x,y) cos(x+y) ;
fprintf('3. Solving BVP\n');
uh=solvePDE(PDE);
```

(a) Matlab/Octave

```
print('1. Creating the mesh')
d=2;Th=HyperCube(d,50)
print('2. Definition of the BVP')
LOp=Loperator(d=d,A=[[1,0],[0,1]])
pde=initPDE(LOp,Th)
pde=setBC_PDE(pde,1,0,'Dirichlet',0.,None)
pde=setBC_PDE(pde,2,0,'Dirichlet',1.,None)
pde=setBC_PDE(pde,3,0,'Robin',-0.5,
              lambda x,y: 1+x**2+y**2)
pde=setBC_PDE(pde,4,0,'Neumann',0.5,None)
pde.f=lambda x,y: cos(x+y)
print('3. Solving BVP');
uh=solvePDE(pde)
```

(b) Python (`Poisson2D01.py`)

Listing 1: 2D Poisson codes

3.2.2 Poisson PDE with mixed boundary conditions in a 2D distorted domain

We first consider the classical Poisson problem with various boundary conditions in a 2D distorted domain. The problem to solve is the following

2D Poisson problem

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$
\begin{aligned}
-\Delta u &= f \quad \text{in } \Omega \subset \mathbb{R}^2, && (3.6)\\
u &= 0 \quad \text{on } \Gamma_1, && (3.7)\\
u &= 1 \quad \text{on } \Gamma_2, && (3.8)\\
\frac{\partial u}{\partial n} + a_R\, u &= -0.5 \quad \text{on } \Gamma_3, && (3.9)\\
\frac{\partial u}{\partial n} &= 0.5 \quad \text{on } \Gamma_4 && (3.10)
\end{aligned}
$$

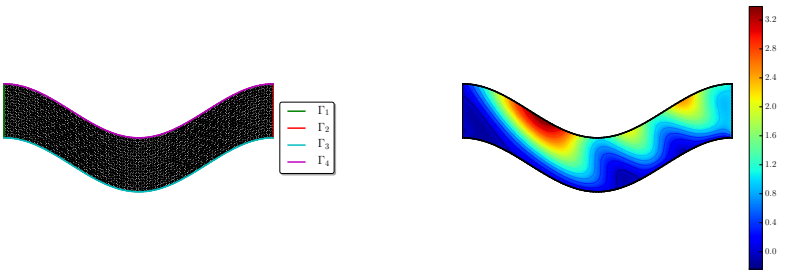
where Ω is the unit hypercube transformed by the function

$$\boldsymbol{\Phi}(x, y) = (20x,\ 2(2y - 1 + \cos(2\pi x))).$$

The boundaries are given in Figure 2a.

f and a_R satisfy:

$$\begin{aligned} f(\boldsymbol{x}) &= \cos(\boldsymbol{x}_1 + \boldsymbol{x}_2) \quad \forall \boldsymbol{x} \in \Omega \\ a_R(\boldsymbol{x}) &= 1 + \boldsymbol{x}_1^2 + \boldsymbol{x}_2^2 \quad \forall \boldsymbol{x} \in \Omega \end{aligned}$$



(a) Domain and boundaries (b) Result

Figure 2: 2D Poisson problem with distorted mesh

The operator in (3.6) is the *Stiffness* operator : $\mathcal{L}_{\mathbb{I},\mathbf{0},\mathbf{0},0}$.
The conormal derivative $\frac{\partial u}{\partial n_{\mathcal{L}}}$ is

$$\frac{\partial u}{\partial n_{\mathcal{L}}} := \langle \mathbb{A} \nabla u, \boldsymbol{n} \rangle - \langle \boldsymbol{b} u, \boldsymbol{n} \rangle = \frac{\partial u}{\partial n}.$$

The algorithm using the toolbox for solving (3.6)-(3.10) is given in Algorithm 3.2. The corresponding Matlab/Octave and Python codes are given in Listing 2.

Algorithm 3.2 2D Poisson problem in a distorted domain

1: $\mathcal{T}_h \leftarrow$ HyperCube$(2, [100, 20], trans = (x, y) \mapsto (20x,\ 2(2y - 1 + \cos(2\pi x)))$
2: Dop $\leftarrow$ Loperator$(2, \mathbb{I}, \mathbf{0}, \mathbf{0}, 0)$ ▷ Stiffness operator
3: PDE $\leftarrow$ initPDE$(\texttt{Dop}, \mathcal{T}_h)$
4: PDE $\leftarrow$ setBC_PDE$(\text{PDE}, 1, 1, \texttt{'Dirichlet'}, 0., \varnothing)$ ▷ $u = 0$ on Γ_1
5: PDE $\leftarrow$ setBC_PDE$(\text{PDE}, 2, 1, \texttt{'Dirichlet'}, 1., \varnothing)$ ▷ $u = 1$ on Γ_2
6: PDE $\leftarrow$ setBC_PDE$(\text{PDE}, 3, 1, \texttt{'Robin'}, -0.5, \boldsymbol{x} \rightarrow 1 + \boldsymbol{x}_1^2 + \boldsymbol{x}_2^2)$ ▷ $\frac{\partial u}{\partial n} + a_R\, u = -0.5$ on Γ_3
7: PDE $\leftarrow$ setBC_PDE$(\text{PDE}, 4, 1, \texttt{'Neumann'}, 0.5, \varnothing)$ ▷ $\frac{\partial u}{\partial n} = 0.5$ on Γ_4
8: PDE.$f \leftarrow (\boldsymbol{x}_1, \boldsymbol{x}_2) \mapsto \cos(\boldsymbol{x}_1 + \boldsymbol{x}_2)$
9: $\boldsymbol{u}_h \leftarrow$ SolvePDE(PDE)

A numerical solution for a given mesh is shown on Figure 2b

3.2.3 2D condenser problem

The problem to solve is the Laplace problem for a condenser.

```
fprintf('1. Creating the mesh\n');
trans=@(q) [20*q(1,:);  ...
           2*(2*q(2,:)-1+cos(2*pi*q(1,:)))];
Th=HyperCube(2,[100,20],trans);
fprintf('2. Definition of the BVP\n');
Lop=Loperator(2,{1,0;0,1},[],[],[]);
pde=initPDE(Lop,Th);
pde=setBC_PDE(pde,1,1,'Dirichlet', 0 );
pde=setBC_PDE(pde,2,1,'Dirichlet', 1 );
pde=setBC_PDE(pde,3,1,'Robin',-0.5 , ...
              @(x1,x2) 1+x1.^2+x2.^2 );
pde=setBC_PDE(pde,4,1,'Neumann', 0.5 );
pde.f=@(x,y) cos(x+y) ;
fprintf('3. Solving BVP\n');
uh=solvePDE(pde);
```

(a) Matlab/Octave

```
print('1. Creating the mesh')
trans=lambda q: np.c_[20*q[:,0],
      2*(2*q[:,1]-1+np.cos(2*pi*q[:,0]))]
Th=HyperCube(2,[100,20],trans=trans)
print('2. Definition of the BVP')
LOp=Loperator(d=2,A=[[1,0],[0,1]])
pde=initPDE(LOp,Th)
pde=setBC_PDE(pde,1,0,'Dirichlet',0.,None)
pde=setBC_PDE(pde,2,0,'Dirichlet',1.,None)
pde=setBC_PDE(pde,3,0,'Robin',-0.5,
              lambda x,y: 1+x**2+y**2)
pde=setBC_PDE(pde,4,0,'Neumann',0.5,None)
pde.f=lambda x,y: cos(x+y)
print('3. Solving BVP');
uh=solvePDE(pde)
```

(b) Python (`Poisson2DTrans.py`)

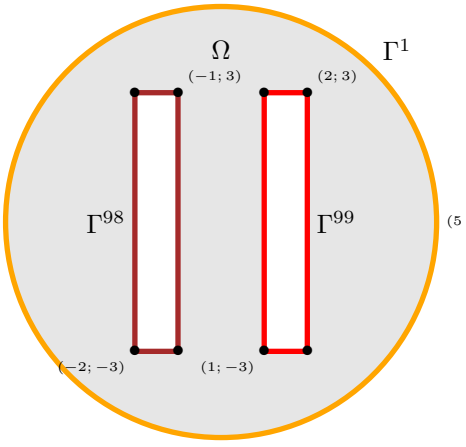
Listing 2: 2D Poisson codes with distorted mesh

2D condenser problem

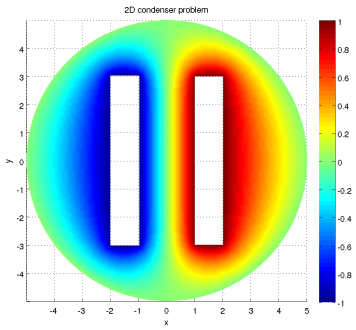
Find $u \in \mathrm{H}^2(\Omega)$ such that

$$-\Delta u = 0 \text{ in } \Omega \subset \mathbb{R}^2, \tag{3.11}$$
$$u = 0 \text{ on } \Gamma_1, \tag{3.12}$$
$$u = -1 \text{ on } \Gamma_{98}, \tag{3.13}$$
$$u = 1 \text{ on } \Gamma_{99}, \tag{3.14}$$

where Ω and its boundaries are given in Figure 3a.



(a) Domain for the condenser problem



(b) Result for the 2D condenser problem

Figure 3: Condenser problem

The problem (3.11)-(3.14) can be equivalently expressed as the scalar BVP (2.2)-(2.4) :

2D condenser problem as a *scalar* BVP

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$\begin{aligned} \mathcal{L}(u) &= f && \text{in } \Omega, \\ u &= g^D && \text{on } \Gamma^D. \end{aligned}$$

where $\Gamma^R = \varnothing$ (no Robin boundary condition) and

- $\mathcal{L} := \mathcal{L}_{\mathbb{I},\mathbf{0},\mathbf{0},0}$, and then the conormal derivative of u is given by

$$\frac{\partial u}{\partial n_{\mathcal{L}}} := \langle \mathbb{A} \nabla u, \boldsymbol{n} \rangle - \langle \boldsymbol{b} u, \boldsymbol{n} \rangle = \frac{\partial u}{\partial n}.$$

- $f(\boldsymbol{x}) := 0$
- $\Gamma^D = \Gamma_1 \cup \Gamma_{98} \cup \Gamma_{99}$
- $g^D := 0$ on Γ_1, and $g^D := -1$ on Γ_{98} and $g^D := +1$ on Γ_{99}

The algorithm using the toolbox for solving (3.11)-(3.14) is the following:

Algorithm 3.3 2D condenser

1: $\mathcal{T}_h \leftarrow$ getMesh(...) ▷ Load FreeFEM++ mesh
2: Dop $\leftarrow$ Loperator$(\mathbb{I}, \mathbf{0}, \mathbf{0}, 0)$ ▷ Stiffness operator
3: PDE $\leftarrow$ initPDE(Dop, $\mathcal{T}_h$)
4: PDE $\leftarrow$ setBCLabel(PDE, 'Dirichlet', 1, 1, 0.) ▷ $u = 0$ on Γ_1
5: PDE $\leftarrow$ setBCLabel(PDE, 'Dirichlet', 99, 1, 1.) ▷ $u = 1$ on Γ_{99}
6: PDE $\leftarrow$ setBCLabel(PDE, 'Dirichlet', 98, 1, −1.) ▷ $u = -1$ on Γ_{98}
7: $\boldsymbol{x} \leftarrow$ SolvePDE(PDE)

We give respectively in Listing 3a and 3b the corresponding Matlab/Octave and Python codes.

```
fprintf('1. Reading of the condenser mesh\n')
Th=GetMesh2DOpt('condenser2D-10.msh');
fprintf('2. Definition of the BVP : 2D condenser\n')
Lop=Loperator(Th.d,{1,0;0,1},[],[],[]);
pde=initPDE(Lop,Th);
pde=setBC_PDE(pde, 1 ,1,'Dirichlet',0);
pde=setBC_PDE(pde,99 ,1,'Dirichlet',1);
pde=setBC_PDE(pde,98 ,1,'Dirichlet',-1);
fprintf('3. Solving BVP\n')
x=solvePDE(pde);
```

(a) Matlab/Octave

```
print("1. Reading of the condenser mesh")
Th=readFreeFEM('condenser2D-10.msh')
print("2. Definition of the BVP : 2D condenser")
Lop=Loperator(d=Th.d,A=[[1,None],[None,1]])
pde=initPDE(Lop,Th)
pde=setBC_PDE(pde,1,0,'Dirichlet',0.,None)
pde=setBC_PDE(pde,99,0,'Dirichlet',1.,None)
pde=setBC_PDE(pde,98,0,'Dirichlet',-1.,None)
print("3. Solving BVP")
x=solvePDE(pde)
```

(b) Python (`condenser2D.py`)

Listing 3: 2D condenser codes

The solution for a given mesh is shown on Figure 3b

3.2.4 Stationary convection-diffusion problem in 2D

The 2D problem to solve is the following

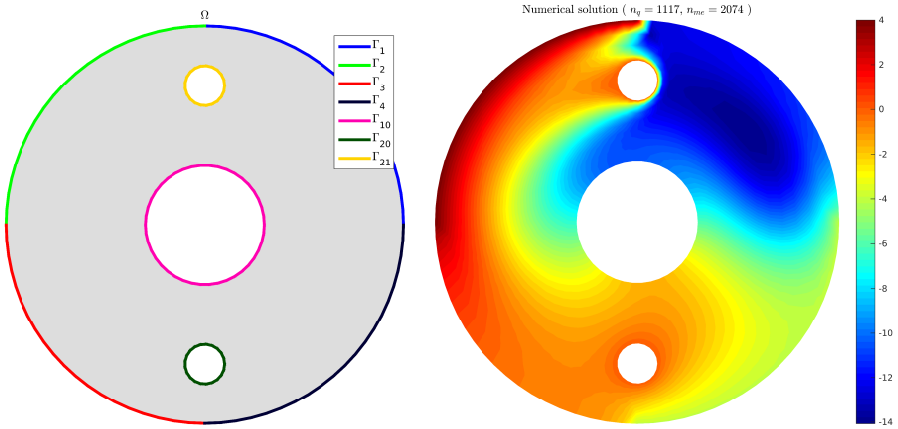
2D stationary convection-diffusion problem

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$
\begin{aligned}
-\operatorname{div}(\alpha \nabla u) + \langle \boldsymbol{V}, \nabla u \rangle + \beta u &= f \quad \text{in } \Omega \subset \mathbb{R}^2, && (3.15)\\
u &= 4 \quad \text{on } \Gamma_2, && (3.16)\\
u &= -4 \quad \text{on } \Gamma_4, && (3.17)\\
u &= 0 \quad \text{on } \Gamma_{20} \cup \Gamma_{21}, && (3.18)\\
\frac{\partial u}{\partial n} &= 0 \quad \text{on } \Gamma_1 \cup \Gamma_3 \cup \Gamma_{10} && (3.19)
\end{aligned}
$$

where Ω and its boundaries are given in Figure 4a. This problem is well posed if $\alpha(\boldsymbol{x}) > 0$ and $\beta(\boldsymbol{x}) \geqslant 0$.
We choose α, $\boldsymbol{V}$, β and f in Ω as :

$$
\begin{aligned}
\alpha(\boldsymbol{x}) &= 0.1 + (x_1 - 0.5)^2,\\
\boldsymbol{V}(\boldsymbol{x}) &= (-10x_2, 10x_1)^t,\\
\beta(\boldsymbol{x}) &= 0.01,\\
f(\boldsymbol{x}) &= -200\exp(-10((x_1 - 0.75)^2 + x_2^2)).
\end{aligned}
$$



(a) Domain and boundaries (b) Result

Figure 4: 2D stationary convection-diffusion problem

The problem (3.15)-(3.19) can be equivalently expressed as the scalar BVP (2.2)-(2.4) :

2D stationary convection-diffusion problem as a *scalar* BVP

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$\begin{aligned} \mathcal{L}(u) &= f && \text{in } \Omega, \\ u &= g^D && \text{on } \Gamma^D, \\ \frac{\partial u}{\partial n_{\mathcal{L}}} + a^R u &= g^R && \text{on } \Gamma^R. \end{aligned}$$

where

- $\mathcal{L} := \mathcal{L}_{\alpha\mathbb{I},\mathbf{0},\boldsymbol{V},\beta}$, and then the conormal derivative of u is given by

$$\frac{\partial u}{\partial n_{\mathcal{L}}} := \langle \mathbb{A} \nabla u, \boldsymbol{n} \rangle - \langle \boldsymbol{b} u, \boldsymbol{n} \rangle = \alpha \frac{\partial u}{\partial n}.$$

- $\Gamma^D = \Gamma_2 \cup \Gamma_4 \cup \Gamma_{20} \cup \Gamma_{21}$ and $\Gamma^R = \Gamma_1 \cup \Gamma_3 \cup \Gamma_{10}$
- $g^D := 4$ on Γ_2, and $g^D := -4$ on Γ_4 and $g^D := 0$ on $\Gamma_{20} \cup \Gamma_{21}$
- $a^R = g^R := 0$ on Γ^R.

The algorithm using the toolbox for solving (3.15)-(3.19) is the following:

Algorithm 3.4 Stationary convection-diffusion problem in 2D

$\mathcal{T}_h \leftarrow$ getMesh(...) ▷ Load FreeFEM++ mesh
$\alpha \leftarrow (x, y) \longmapsto 0.1 + (y - 0.5)(y - 0.5)$
$\beta \leftarrow 0.01$
$\mathcal{L} \leftarrow$ Loperator$(2, \begin{pmatrix} \alpha & 0 \\ 0 & \alpha \end{pmatrix}, \mathbf{0}, \begin{pmatrix} -10y \\ 10x \end{pmatrix}, \beta)$
pde $\leftarrow$ initPDE$(\mathcal{L}, \mathcal{T}_h)$ ▷ Set homogeneous `'Neumann'` condition on all boundaries
pde $\leftarrow$ setBC_PDE(pde, 2, 1, `'Dirichlet'`, 4., $\varnothing$) ▷ $u = 4$ on Γ_2
pde $\leftarrow$ setBC_PDE(pde, 4, 1, `'Dirichlet'`, -4, $\varnothing$) ▷ $u = -4$ on Γ_4
pde $\leftarrow$ setBC_PDE(pde, 20, 1, `'Dirichlet'`, 0, $\varnothing$ ▷ $u = 0$ on Γ_{20}
pde $\leftarrow$ setBC_PDE(pde, 21, 1, `'Dirichlet'`, 0, $\varnothing$) ▷ $u = 0$ on Γ_{21}
pde.$f \leftarrow (x, y) \mapsto -200 \exp(-10(x - 0.75)^2 + y^2)$
$\boldsymbol{x} \leftarrow$ SolvePDE(pde)

We give respectively in Listing 4a and 4b the corresponding Matlab/Octave and Python codes.

The numerical solution for a given mesh is shown on Figure 4b

3.2.5 Stationary convection-diffusion problem in 3D

Let $A = (x_A, y_A) \in \mathbb{R}^2$ and $\mathcal{C}_A^r([z_{min}, z_{max}])$ be the right circular cylinder along $z-$axis ($z \in [z_{min}, z_{max}]$) with bases the circles of radius r and center (x_A, y_A, z_{min}) and (x_A, y_A, z_{max}).

Let Ω be the cylinder defined by

$$\Omega = \mathcal{C}^1_{(0,0)}([0,3]) \backslash \{\mathcal{C}^{0.3}_{(0,0)}([0,3]) \cup \mathcal{C}^{0.1}_{(0,-0.7)}([0,3]) \cup \mathcal{C}^{0.1}_{(0,0.7)}([0,3])\}.$$

```
fprintf('1. Reading of the mesh\n');
Th=GetMesh2DOpt('sampleD2d01-20.msh');
fprintf('2. Definition of the BVP\n')
af=@(x,y) 0.1+(y-0.5).*(y-0.5);
Vx=@(x,y) -10*y;Vy=@(x,y) 10*x;
b=0.01;g2=4;g4=-4;
f=@(x,y) -200.0*exp(-((x-0.75).^2+y.^2)/(0.1));
L=Loperator(Th.d,{af,[];[],af},[],{Vx,Vy},b);
pde=initPDE(L,Th);
pde=setBC_PDE(pde,2,1,'Dirichlet',g2);
pde=setBC_PDE(pde,4,1,'Dirichlet',g4);
pde=setBC_PDE(pde,20,1,'Dirichlet',0);
pde=setBC_PDE(pde,21,1,'Dirichlet',0);
pde.f=f;
fprintf('3. Solving BVP\n')
x=solvePDE(pde);
```

(a) Matlab/Octave

```
print("1. Reading of the mesh")
Th=readFreeFEM("sampleD2d01-20.msh")
print("2. Definition of the BVP")
af=lambda x,y: 0.1+(y-0.5)*(y-0.5)
Vx=lambda x,y: -10*y;Vy=lambda x,y: 10*x
b=0.01;g2=4;g4=-4;
f=lambda x,y: -200*exp(-((x-0.75)**2+y**2)\
                       /(0.1));
Lop=Loperator(d=Th.d,A=[[af,None],[None,af]],
              c=[Vx,Vy],a0=b)
pde=initPDE(Lop,Th)
pde=setBC_PDE(pde,2,0, 'Dirichlet',g2)
pde=setBC_PDE(pde,4,0, 'Dirichlet',g4)
pde=setBC_PDE(pde,20,0,'Dirichlet',0)
pde=setBC_PDE(pde,21,0,'Dirichlet',0)
pde.f=f
print("3. Solving BVP")
x=solvePDE(pde)
```

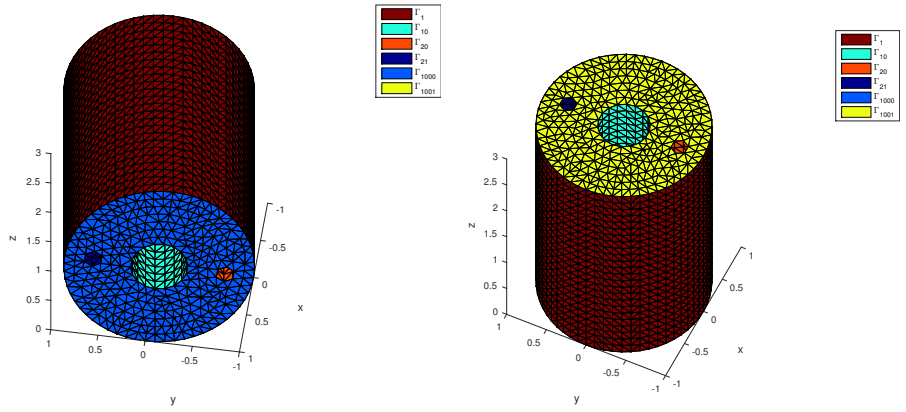
(b) Python (`StationaryConvectionDiffusion2D.py`)

Listing 4: 2D Poisson codes

We respectively denote by Γ_{1000} and Γ_{1001} the $z = 0$ and $z = 3$ bases of Ω.

$\Gamma_1, \Gamma_{10}, \Gamma_{20}$ and Γ_{21} are respectively the curved surfaces of cylinders $\mathcal{C}^1_{(0,0)}([0,3])$, $\mathcal{C}^{0.3}_{(0,0)}([0,3])$, $\mathcal{C}^{0.1}_{(0,-0.7)}([0,3])$ and $\mathcal{C}^{0.1}_{(0,0.7)}([0,3])$.

The domain Ω and its boundaries are represented in Figure 5.



(a) Bottom view (b) Top view

Figure 5: Mesh for the stationary convection-diffusion problem in 3D

The 3D problem to solve is the following

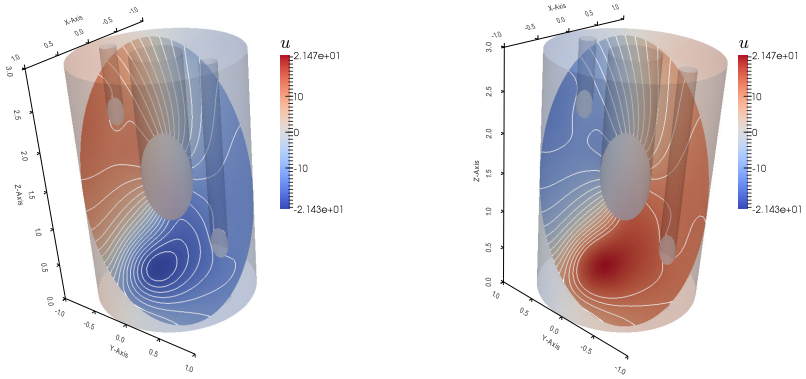
3D problem : Stationary convection-diffusion

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$
\begin{aligned}
-\operatorname{div}(\alpha \nabla u) + \langle \boldsymbol{V}, \nabla u \rangle + \beta u &= f \quad \text{in } \Omega \subset \mathbb{R}^3, && (3.20)\\
\alpha \frac{\partial u}{\partial n} + a_{20} u &= g_{20} \quad \text{on } \Gamma_{20}, && (3.21)\\
\alpha \frac{\partial u}{\partial n} + a_{21} u &= g_{21} \quad \text{on } \Gamma_{21}, && (3.22)\\
\frac{\partial u}{\partial n} &= 0 \quad \text{on } \Gamma^N && (3.23)
\end{aligned}
$$

where $\Gamma^N = \Gamma_1 \cup \Gamma_{10} \cup \Gamma_{1000} \cup \Gamma_{1001}$. This problem is well posed if $\alpha(\boldsymbol{x}) > 0$ and $\beta(\boldsymbol{x}) \geqslant 0$.
We choose $a_{20} = a_{21} = 1$, $g_{21} = -g_{20} = 0.05$ $\beta = 0.01$ and :

$$
\begin{aligned}
\alpha(\boldsymbol{x}) &= 0.7 + \boldsymbol{x}_3/10,\\
\boldsymbol{V}(\boldsymbol{x}) &= (-10x_2, 10x_1, 10x_3)^t,\\
f(\boldsymbol{x}) &= -800\exp(-10((x_1 - 0.65)^2 + x_2^2 + (x_3 - 0.5)^2))\\
&\quad +800\exp(-10((x_1 + 0.65)^2 + x_2^2 + (x_3 - 0.5)^2)).
\end{aligned}
$$



(a) first view (b) second view

Figure 6: Stationary convection-diffusion problem in 3D : numerical solution

The problem (3.20)-(3.23) can be equivalently expressed as the scalar BVP (2.2)-(2.4) :

3D stationary convection-diffusion problem as a *scalar* BVP

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$
\begin{aligned}
\mathcal{L}(u) &= f && \text{in } \Omega,\\
\frac{\partial u}{\partial n_{\mathcal{L}}} + a^R u &= g^R && \text{on } \Gamma^R.
\end{aligned}
$$

where

- $\mathcal{L} := \mathcal{L}_{\alpha\mathbb{I},\mathbf{0},\boldsymbol{V},\beta}$, and then the conormal derivative of u is given by

$$\frac{\partial u}{\partial n_{\mathcal{L}}} := \langle \mathbb{A} \nabla u, \boldsymbol{n} \rangle - \langle \boldsymbol{b} u, \boldsymbol{n} \rangle = \alpha \frac{\partial u}{\partial n}.$$

- $\Gamma^R = \Gamma_1 \cup \Gamma_{10} \cup \Gamma_{20} \cup \Gamma_{21} \cup \Gamma_{1000} \cup \Gamma_{1001}$ (and $\Gamma^D = \varnothing$)
-

$$a^R = \begin{cases} 0 & \text{on } \Gamma_1 \cup \Gamma_{10} \cup \Gamma_{1000} \cup \Gamma_{1001} \\ 1 & \text{on } \Gamma_{20} \cup \Gamma_{21} \end{cases}$$

$$g^R = \begin{cases} 0 & \text{on } \Gamma_1 \cup \Gamma_{10} \cup \Gamma_{1000} \cup \Gamma_{1001} \\ 0.05 & \text{on } \Gamma_{21}, \\ -0.05 & \text{on } \Gamma_{20} \end{cases}$$

The algorithm using the toolbox for solving (3.20)-(3.23) is the following:

Algorithm 3.5 sampleD3d01 problem

$\mathcal{T}_h \leftarrow$ getMesh(...) ▷ Load FreeFEM++ 3D-mesh
$\alpha \leftarrow (x, y, z) \longmapsto 0.7 + z/10$
Dop $\leftarrow$ Loperator$(3, \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \alpha & 0 \\ 0 & 0 & \alpha \end{pmatrix}, \mathbf{0}, \begin{pmatrix} -10y \\ 10x \\ 10z \end{pmatrix}, \beta)$
PDE $\leftarrow$ initPDE(Dop, $\mathcal{T}_h$) ▷ Set homogeneous 'Neumann' condition on all boundaries
PDE $\leftarrow$ setBC_PDE(PDE, 20, 1, 'Robin', -0.05, 1.)
PDE $\leftarrow$ setBC_PDE(PDE, 21, 1, 'Robin', 0.05, 1.)
PDE.$f \leftarrow (x, y, z) \mapsto -800 \exp(-10(x - 0.65)^2 + y^2 + (z - 0.5)^2)$
$\quad +800 \exp(-10(x + 0.65)^2 + y^2 + (z - 0.5)^2)$
$\boldsymbol{x} \leftarrow$ SolvePDE(PDE)

We give respectively in Listing 1 and 2 the corresponding Matlab/Octave and Python codes.

Listing 1: **3D stationary convection-diffusion, Matlab/Octave code**

```
1
2  fprintf('1. Reading of the mesh\n');
3  Th=GetMesh3DOpt('sampleD3d01-6.mesh','format','medit');
4  fprintf('2. Definition of the BVP\n')
5  af=@(x,y,z) 0.7+z/10;%0.1+(y-0.5).*(y-0.5);
6  Vx=@(x,y,z) -10*y;Vy=@(x,y,z) 10*x;Vz=@(x,y,z) 10*z;
7  b=0.01;%0.01;g2=4;g4=-4;
8  f=@(x,y,z) -800.0*exp(-10*((x-0.65).*(x-0.65)+y.*y+(z-0.5).^2)) ...
9             +800.0*exp(-10*((x+0.65).*(x+0.65)+y.*y+(z-0.5).^2));
10 Lop=Loperator(Th.d,{af,[],[];[],af,[];[],[],af},[],{Vx,Vy,Vz},b);
11 pde=initPDE(Lop,Th);
12 pde=setBC_PDE(pde,20,1,'Robin', -0.05 , 1 );
13 pde=setBC_PDE(pde,21,1,'Robin', 0.05 , 1 );
14 pde.f=f;
15 fprintf('3. Solving BVP\n')
16 x=solvePDE(pde);
```

Listing 2: **3D stationary convection-diffusion, Python code**

```
print("1. Reading of the mesh")
Th=readFreeFEM3D("sampleD3d01-6.mesh")
print("2. Definition of the BVP")
af=lambda x,y,z: 0.7+z/10
Vx=lambda x,y,z: -10*y;Vy=lambda x,y,z: 10*x;Vz=lambda x,y,z: 10*z
f=lambda x,y,z: -800.0*exp(-10*((x-0.65)**2+y*y+(z-0.5)**2))+\
                 800.0*exp(-10*((x+0.65)**2+y*y+(z-0.5)**2))
Lop=Loperator(d=3,A=[[af,None,None],[None,af,None],[None,None,af]],\
              c=[Vx,Vy,Vz], a0=0.01)
pde=PDE(Lop,Th)
pde=setBC_PDE(pde,20,0,'Robin',-0.05,1)
pde=setBC_PDE(pde,21,0,'Robin',0.05,1)
pde.f=f
print("3. Solving BVP")
x=solvePDE(pde)
```

The numerical solution for a more refined mesh is shown on Figure 6

3.2.6 Laplace problem in $[0,1]^d$

The Laplace problem in any d-dimensional domain is considered here, with various boundary conditions.

Let $\Omega = [0,1]^d$ be the hypercube in $\mathbb{R}^d$. The 2^d faces of this hypercube have a unique label : $\forall i \in [\![1,d]\!]$, faces $x_i \equiv 0$ and $x_i \equiv 1$ are respectively of label $(2i-1)$ and $2i$.

Laplace problem in $[0,1]^d$

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$-\Delta u = 0 \quad \text{in } \Omega \subset \mathbb{R}^d, \tag{3.24}$$

$$u = 1 \quad \text{on } \Gamma_1 \cup \Gamma_2, \tag{3.25}$$

$$\frac{\partial u}{\partial n} + 5u = 1 \quad \text{on } \Gamma_3 \cup \Gamma_4, \tag{3.26}$$

$$\frac{\partial u}{\partial n} = 0 \quad \text{on } \bigcup_{i=3}^{d} \Gamma_{2i-1} \cup \Gamma_{2i}, \tag{3.27}$$

The problem (3.24)-(3.27) can be equivalently expressed as the scalar BVP (2.2)-(2.4) :

Laplace problem in $[0,1]^d$ as a *scalar* BVP

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$\begin{aligned} \mathcal{L}(u) &= f && \text{in } \Omega, \\ u &= g^D && \text{on } \Gamma^D, \\ \frac{\partial u}{\partial n_{\mathcal{L}}} + a^R u &= g^R && \text{on } \Gamma^R. \end{aligned}$$

where

- $\mathcal{L} := \mathcal{L}_{\mathbb{I},\mathbf{0},\mathbf{0},0}$ and then the conormal derivative of u is given by

$$\frac{\partial u}{\partial n_{\mathcal{L}}} := \langle \mathbb{A} \nabla u, \boldsymbol{n} \rangle - \langle \boldsymbol{b} u, \boldsymbol{n} \rangle = \frac{\partial u}{\partial n}.$$

- $f := 0$

- $\Gamma^D = \Gamma_1 \cup \Gamma_2$ and $\Gamma^R = \bigcup_{i=2}^{d} \Gamma_{2i-1} \cup \Gamma_{2i}$
- $g^D := 1$ on $\Gamma_1 \cup \Gamma_2$
- $g^R := 1$ on $\Gamma_3 \cup \Gamma_4$ and $g^R := 0$ on $\bigcup_{i=3}^{d} \Gamma_{2i-1} \cup \Gamma_{2i}$
- $a^R(\boldsymbol{x}) := 1$ on $\Gamma_3 \cup \Gamma_4$ and $a^R := 0$ on $\bigcup_{i=3}^{d} \Gamma_{2i-1} \cup \Gamma_{2i}$

The algorithm using the toolbox for solving (3.24)-(3.27) is the following:

Algorithm 3.6 Laplace equation in $[0,1]^d$

$\mathcal{T}_h \leftarrow$ HyperCube(d, N)
$\mathcal{L} \leftarrow$ Loperator$(d, \mathbb{I}_d, \mathbf{0}_d, \mathbf{0}_d, 0)$ ▷ Stiffness operator
pde $\leftarrow$ initPDE$(\mathcal{L}, \mathcal{T}_h)$
pde $\leftarrow$ setBC_PDE(pde, 1, 1, 'Dirichlet', 1, $\varnothing$)
pde $\leftarrow$ setBC_PDE(pde, 2, 1, 'Dirichlet', 1, $\varnothing$)
pde $\leftarrow$ setBC_PDE(pde, 3, 1, 'Robin', 1, 1)
pde $\leftarrow$ setBC_PDE(pde, 4, 1, 'Robin', 1, 1)
$\boldsymbol{x} \leftarrow$ SolvePDE(pde)

The corresponding Matlab/Octave and Python codes are given in Listing 5.

```
fprintf('1. Set hypercube
    mesh\n');
d=input('d=');
N=input('N=');
Th=HyperCube(d,N);
fprintf('2. Definition of
    the BVP : %dD
    Laplace\n',d)
A=cell(d,d);
for i=1:d, A{i,i}=1;end
Lop=Loperator(d,A,[],[],[]);
pde=initPDE(Lop,Th);
pde=setBC_PDE(pde,1,1,'Dirichlet',1);
pde=setBC_PDE(pde,2,1,'Dirichlet',-1);
pde=setBC_PDE(pde,3,1,'Robin',5,1);
pde=setBC_PDE(pde,4,1,'Robin',5,1);
fprintf('3. Solving BVP\n')
x=solvePDE(pde);
```

(a) Matlab/Octave

```
print("1. Creating the mesh
    of the hypercube [0,1]^d")
d=int(input("d="))
N=int(input("N="))
Th=HyperCube(d,N)
print("2. Definition of the
    BVP : %dD Laplace"%d)
A=NoneMatrix(d,d)
for i in range(d):
  A[i][i]=1
Lop=Loperator(d=d,A=A)
pde=initPDE(Lop,Th)
pde=setBC_PDE(pde,1,0,'Dirichlet',0)
pde=setBC_PDE(pde,2,0,'Dirichlet',1)
pde=setBC_PDE(pde,3,0,'Robin',5,1)
pde=setBC_PDE(pde,4,0,'Robin',5,1)
print("3. Solving BVP")
x=solvePDE(pde)
```

(b) Python

Listing 5: Codes for Laplace problem in dimension d

It should be noted that in dimension d, the mesh of the hypercube $[0,1]^d$ obtained by the call to the function $\mathcal{T}_h \leftarrow$ HyperCube(d, N) contains $\mathrm{n_{me}} = d!(N-1)^d$ d-simplices , N being the number of points in each direction. This number can be very huge (see Table 1). So one does not need to be too ambitious in dimension $d > 3$ and choose a reasonable N.

$N \ddots d$	2	3	4	5	6
5	32	384	6144	122880	2949120
10	162	4374	157464	7085880	382637520
15	392	16464	921984	64538880	2147483647

Table 1: Number of d-simplices in hypercube $[0,1]^d$ meshes

3.2.7 Grad-Shafranov problem

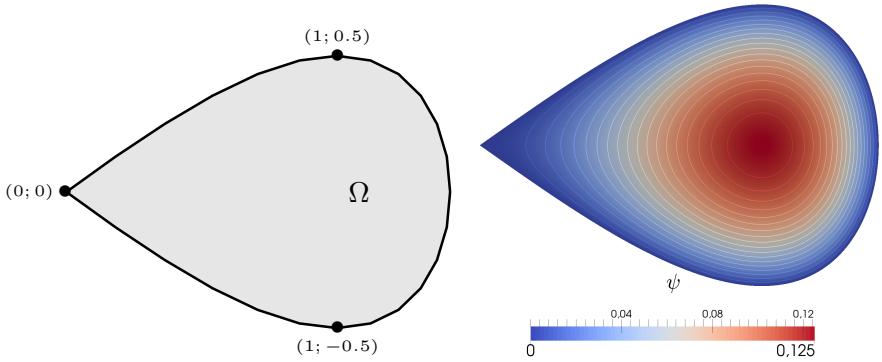
We consider here the 2D Grad-Shafranov problem already tested in [8] defined by

Grad-Shafranov problem

Find $\psi \in \mathrm{H}^2(\Omega)$ such that

$$-\Delta\psi + a_0\,\psi = f \quad \text{in } \Omega \subset \mathbb{R}^2, \tag{3.28}$$

$$\psi = 0 \quad \text{on } \Gamma \tag{3.29}$$



(a) Domain (b) Result

Figure 7: Grad-Shafranov problem

The functions a_0 and f satisfy for all $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2) \in \Omega$:

$$a_0(\boldsymbol{x}) = \frac{1}{\boldsymbol{x}_1},$$

$$f(\boldsymbol{x}) = \boldsymbol{x}_1^2 + 1$$

The geometry can be described by a parametric function as

$$\begin{cases} \boldsymbol{x}_1(t) = \sqrt{1+\cos(t)} & \forall t \in [0, 2\pi], \\ \boldsymbol{x}_2(t) = 0.5\,\sin(t) \end{cases}$$

The operator in (3.28) is the following one : $\mathcal{L}_{\mathbb{I},\mathbf{0},\mathbf{0},a_0}$.

The conormal derivative $\frac{\partial\psi}{\partial n_{\mathcal{L}}}$ is

$$\frac{\partial\psi}{\partial n_{\mathcal{L}}} := \langle \mathbb{A}\,\nabla\psi, \boldsymbol{n}\rangle - \langle \boldsymbol{b}\psi, \boldsymbol{n}\rangle = \frac{\partial\psi}{\partial n}.$$

The algorithm using the toolbox for solving (3.28)-(3.29) is the following:

Algorithm 3.7 Grad-Shafranov problem

```
1: T_h ← getMesh(...)                                   ▷ Load FreeFEM++ mesh
2: Dop ← Loperator(2, I, 0, 0, x ↦ 1/x_1)
3: PDE ← initPDE(Dop, T_h)
4: PDE ← setBC_PDE(PDE, 1, 1, 'Dirichlet', 0., ∅)      ▷ u = 0 on Γ_1
5: PDE.f ← x ↦ x_1^2 + 1
6: x ← SolvePDE(PDE)
```

The solution for a given mesh is shown on Figure 7.

3.3 Vector case

3.3.1 Elasticity problem

General case ($d = 2, 3$)

We consider here Hooke's law in linear elasticity, under small strain hypothesis (see for example [5]).

For a sufficiently regular vector field $\boldsymbol{u} = (u_1, \dots, u_d) : \Omega \to \mathbb{R}^d$, we define the linearized strain tensor $\underline{\boldsymbol{\epsilon}}$ by

$$\underline{\boldsymbol{\epsilon}}(\boldsymbol{u}) = \frac{1}{2}\left(\boldsymbol{\nabla}(\boldsymbol{u}) + \boldsymbol{\nabla}^t(\boldsymbol{u})\right).$$

We set $\underline{\boldsymbol{\epsilon}} = (\epsilon_{11}, \epsilon_{22}, 2\epsilon_{12})^t$ in 2d and $\underline{\boldsymbol{\epsilon}} = (\epsilon_{11}, \epsilon_{22}, \epsilon_{33}, 2\epsilon_{12}, 2\epsilon_{23}, 2\epsilon_{13})^t$ in 3d, with $\epsilon_{ij}(\boldsymbol{u}) = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$. Then the Hooke's law writes

$$\underline{\boldsymbol{\sigma}} = \mathbb{C}\underline{\boldsymbol{\epsilon}},$$

where $\underline{\boldsymbol{\sigma}}$ is the elastic stress tensor and $\mathbb{C}$ the elasticity tensor.

The material is supposed to be isotropic. Thus the elasticity tensor $\mathbb{C}$ is only defined by the Lamé parameters λ and μ, which satisfy $\lambda + \mu > 0$. We also set $\gamma = 2\,\mu + \lambda$. For $d = 2$ or $d = 3$, $\mathbb{C}$ is given by

$$\mathbb{C} = \begin{pmatrix} \lambda\mathbb{1}_2 + 2\mu\mathbb{I}_2 & 0 \\ 0 & \mu \end{pmatrix}_{3\times 3} \quad \text{or} \quad \mathbb{C} = \begin{pmatrix} \lambda\mathbb{1}_3 + 2\mu\mathbb{I}_3 & 0 \\ 0 & \mu\mathbb{I}_3 \end{pmatrix}_{6\times 6},$$

respectively, where $\mathbb{1}_d$ is a d-by-d matrix of ones, and $\mathbb{I}_d$ the d-by-d identity matrix.

For dimension $d = 2$ or $d = 3$, we have:

$$\boldsymbol{\sigma}_{\alpha\beta}(\boldsymbol{u}) \quad = \quad 2\,\mu\,\boldsymbol{\epsilon}_{\alpha\beta}(\boldsymbol{u}) + \lambda\,\mathrm{tr}(\boldsymbol{\epsilon}(\boldsymbol{u}))\delta_{\alpha\beta} \quad \forall \alpha, \beta \in [\![1, d]\!]$$

The problem to solve is the following

Elasticity problem

Find $\boldsymbol{u} = \mathrm{H}^2(\Omega)^d$ such that

$$-\operatorname{div}(\boldsymbol{\sigma}(\boldsymbol{u})) = \boldsymbol{f}, \quad \text{in } \Omega \subset \mathbb{R}^d, \tag{3.30}$$

$$\boldsymbol{\sigma}(\boldsymbol{u}).\boldsymbol{n} = \boldsymbol{0} \quad \text{on } \Gamma^R, \tag{3.31}$$

$$\boldsymbol{u} = \boldsymbol{0} \quad \text{on } \Gamma^D. \tag{3.32}$$

Now, with the following lemma, we obtain that this problem can be rewritten as the vector BVP defined by (2.10) to (2.12).

Lemma 1. *Let $\mathcal{H}$ be the d-by-d matrix of the second order linear differential operators defined in* (2.6) *where $\mathcal{H}_{\alpha,\beta} = \mathcal{L}_{\mathbb{A}^{\alpha,\beta},\boldsymbol{0},\boldsymbol{0},0}$, $\forall(\alpha,\beta) \in [\![1,d]\!]^2$, with*

$$(\mathbb{A}^{\alpha,\beta})_{k,l} = \mu\delta_{\alpha\beta}\delta_{kl} + \mu\delta_{k\beta}\delta_{l\alpha} + \lambda\delta_{k\alpha}\delta_{l\beta}, \ \forall(k,l) \in [\![1,d]\!]^2. \tag{3.33}$$

then

$$\mathcal{H}(\boldsymbol{u}) = -\operatorname{div}\boldsymbol{\sigma}(\boldsymbol{u}) \tag{3.34}$$

and, $\forall \alpha \in [\![1,d]\!]$,

$$\frac{\partial \boldsymbol{u}}{\partial n_{\mathcal{H}_\alpha}} = (\boldsymbol{\sigma}(\boldsymbol{u}).\boldsymbol{n})_\alpha. \tag{3.35}$$

The proof is given in appendix 11.4. So we obtain

Elasticity problem with $\mathcal{H}$ operator in dimension $d = 2$ or $d = 3$

Let $\mathcal{H}$ be the d-by-d matrix of the second order linear differential operators defined in (2.6) where $\forall(\alpha,\beta) \in [\![1,d]\!]^2$, $\mathcal{H}_{\alpha,\beta} = \mathcal{L}_{\mathbb{A}^{\alpha,\beta},\mathbf{0},\mathbf{0},0}$, with

- for $d = 2$,
$$\mathbb{A}^{1,1} = \begin{pmatrix} \gamma & 0 \\ 0 & \mu \end{pmatrix},\ \mathbb{A}^{1,2} = \begin{pmatrix} 0 & \lambda \\ \mu & 0 \end{pmatrix},\ \mathbb{A}^{2,1} = \begin{pmatrix} 0 & \mu \\ \lambda & 0 \end{pmatrix},\ \mathbb{A}^{2,2} = \begin{pmatrix} \mu & 0 \\ 0 & \gamma \end{pmatrix}$$

- for $d = 3$,
$$\mathbb{A}^{1,1} = \begin{pmatrix} \gamma & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \mu \end{pmatrix},\quad \mathbb{A}^{1,2} = \begin{pmatrix} 0 & \lambda & 0 \\ \mu & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},\quad \mathbb{A}^{1,3} = \begin{pmatrix} 0 & 0 & \lambda \\ 0 & 0 & 0 \\ \mu & 0 & 0 \end{pmatrix}$$
$$\mathbb{A}^{2,1} = \begin{pmatrix} 0 & \mu & 0 \\ \lambda & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},\quad \mathbb{A}^{2,2} = \begin{pmatrix} \mu & 0 & 0 \\ 0 & \gamma & 0 \\ 0 & 0 & \mu \end{pmatrix},\quad \mathbb{A}^{2,3} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & \lambda \\ 0 & \mu & 0 \end{pmatrix},$$
$$\mathbb{A}^{3,1} = \begin{pmatrix} 0 & 0 & \mu \\ 0 & 0 & 0 \\ \lambda & 0 & 0 \end{pmatrix},\quad \mathbb{A}^{3,2} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & \mu \\ 0 & \lambda & 0 \end{pmatrix},\quad \mathbb{A}^{3,3} = \begin{pmatrix} \mu & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \gamma \end{pmatrix}.$$

The elasticity problem (3.30) to (3.32) can be rewritten as :
Find $\boldsymbol{u} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_d) \in (\mathrm{H}^2(\Omega))^d$ such that

$$\mathcal{H}(\boldsymbol{u}) = \boldsymbol{f}, \qquad \text{in } \Omega, \tag{3.36}$$
$$\frac{\partial \boldsymbol{u}}{\partial n_{\mathcal{H}_\alpha}} = 0, \qquad \text{on } \Gamma^R_\alpha = \Gamma^R,\ \forall \alpha \in [\![1,d]\!] \tag{3.37}$$
$$\boldsymbol{u}_\alpha = 0, \qquad \text{on } \Gamma^D_\alpha = \Gamma^D,\ \forall \alpha \in [\![1,d]\!]. \tag{3.38}$$

2D example

For example, in 2d, we want to solve the elasticity problem (3.30) to (3.32) where Ω and its boundaries are given in Figure 8. We have $\Gamma^R = \Gamma^1 \cup \Gamma^2 \cup \Gamma^3$, $\Gamma^D = \Gamma^4$.

The material's properties are given by Young's modulus E and Poisson's coefficient ν. As we use plane strain hypothesis, Lame's coefficients verify

$$\mu = \frac{E}{2\,(1+\nu)}, \quad \lambda = \frac{E\,\nu}{(1+\nu)\,(1-2\,\nu)}, \quad \gamma = 2\mu + \lambda$$

The material is rubber so that $E = 21.10^5$Pa and $\nu = 0.45$. We also have $\boldsymbol{f} = \boldsymbol{x} \mapsto (0,-1)^t$.

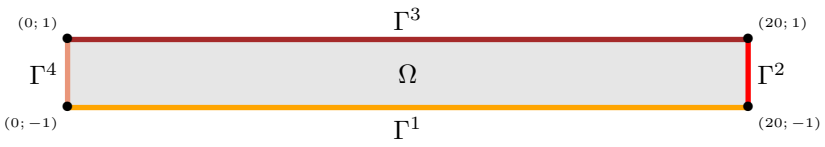


Figure 8: Domain for the 2D elasticity problem

The algorithm using the toolbox for solving (3.30)-(3.32) is the following:

Algorithm 3.8 2D elasticity

$\mathcal{T}_h \leftarrow$ getMesh(...) ▷ Load FreeFEM++ mesh
$\lambda \leftarrow \frac{E\nu}{(1+\nu)(1-2\nu)}$
$\mu \leftarrow \frac{E}{2(1+\nu)}$
Hop $\leftarrow$ InitHoperator$(2,2)$
Hop$(1,1) \leftarrow$ Loperator$(2,[2\mu+\lambda,0;0,\mu],\mathbf{0},\mathbf{0},0)$
Hop$(2,1) \leftarrow$ Loperator$(2,[0,\lambda;\mu,0],\mathbf{0},\mathbf{0},0)$
Hop$(1,2) \leftarrow$ Loperator$(2,[0,\mu;\lambda,0],\mathbf{0},\mathbf{0},0)$
Hop$(2,2) \leftarrow$ Loperator$(2,[\mu,0;0,2\mu+\lambda],\mathbf{0},\mathbf{0},0)$
pde $\leftarrow$ initPDE(Hop, $\mathcal{T}_h$)
pde $\leftarrow$ setBC_PDE(pde, $4, 1:2,$ 'Dirichlet', $\boldsymbol{x} \rightarrow \mathbf{0}$)
pde.$f \leftarrow \boldsymbol{x} \rightarrow [0,-1]$
$\boldsymbol{x} \leftarrow$ SolvePDE(pde)

We give respectively in Listing 3 and 4 the corresponding Matlab/Octave and Python codes.

Listing 3: **2D elasticity, Matlab/Octave code**

```
fprintf('1. Reading of the mesh\n');
Th=GetMesh2DOpt('bar4-15.msh');
fprintf('2. Definition of the BVP\n');
E = 21.5e4; nu = 0.45;
mu= E/(2*(1+nu));
lambda = E*nu/((1+nu)*(1-2*nu));
gamma=lambda+2*mu;
Hop=Hoperator(2,2);
Hop.H{1,1}=Loperator(2,{gamma,[];[],mu},[],[],[]);
Hop.H{1,2}=Loperator(2,{[],lambda;mu,[]},[],[],[]);
Hop.H{2,1}=Loperator(2,{[],mu;lambda,[]},[],[],[]);
Hop.H{2,2}=Loperator(2,{mu,[];[],gamma},[],[],[]);
% One can also use the preset operator function
%   Hop=buildHoperator(2,2,'name','StiffElas','lambda',lambda,'mu',mu);
pde=initPDE(Hop,Th);
pde.f={0,-1};
pde=setBC_PDE(pde,4  ,1:2, 'Dirichlet',{0,0});
fprintf('3. Solving BVP\n');
x=solvePDE(pde);
```

Listing 4: **2D elasticity, Python code**

```
print('1. Reading of the mesh')
Th=readFreeFEM('bar4-15.msh')
print('2. Definition of the BVP')
E = 21.5e4; nu = 0.45
mu= E/(2*(1+nu))
lam = E*nu/((1+nu)*(1-2*nu))
gam=lam+2*mu
Hop=Hoperator(d=2,m=2)
Hop.H[0][0]=Loperator(d=2,A=[[gam,None],[None,mu]])
Hop.H[0][1]=Loperator(d=2,A=[[None,lam],[mu,None]])
Hop.H[1][0]=Loperator(d=2,A=[[None,mu],[lam,None]])
Hop.H[1][1]=Loperator(d=2,A=[[mu,None],[None,gam]])
# One can also use the preset operator function
#    Hop=StiffElasHoperators(2,lam,mu)
pde=initPDE(Hop,Th)
pde.f=[0,-1]
pde=setBC_PDE(pde,4,[0,1],'Dirichlet',[0,0],None)
print('3. Solving BVP')
x=solvePDE(pde,split=True)
```

For a given mesh, its displacement scaled by a factor 10 is shown on Figure 9

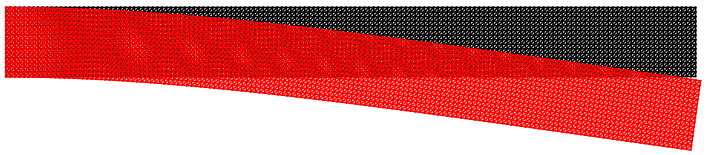


Figure 9: Mesh displacement scaled by a factor 10 for the 2D elasticity problem

3D example

Let $\Omega = [0,5] \times [0,1] \times [0,1] \subset \mathbb{R}^3$. The boundary of Ω is made of six faces and each one has a unique label : 1 to 6 respectively for faces $x_1 = 0$, $x_1 = 5$, $x_2 = 0$, $x_2 = 1$, $x_3 = 0$ and $x_3 = 1$. We represent them in Figure 10.

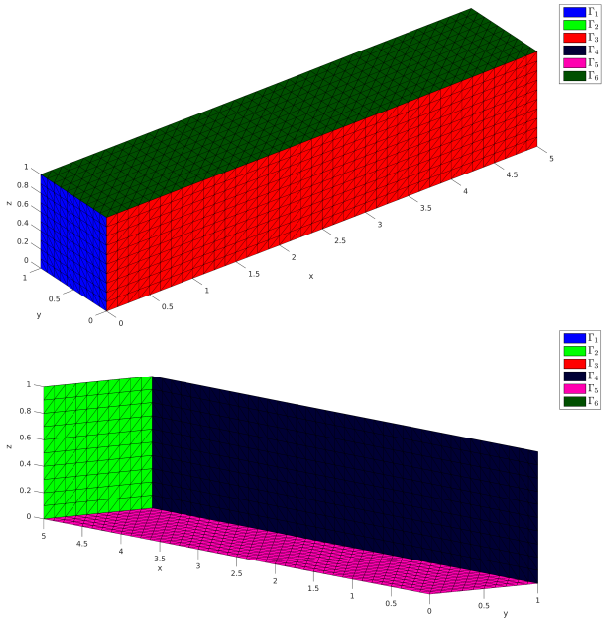


Figure 10: Domain Ω and boundaries for the 3D elasticity problem

We want to solve the elasticity problem (3.30) to (3.32) with $\Gamma^D = \Gamma_1$, $\Gamma^N = \bigcup_{i=2}^{6} \Gamma_i$ and $\boldsymbol{f} = \boldsymbol{x} \mapsto (0, 0, -1)^t$.

The algorithm using the toolbox for solving (3.30)-(3.32) is the following:

Algorithm 3.9 3D elasticity

$\mathcal{T}_h \leftarrow$ getMesh(...) ▷ Load FreeFEM++ mesh
$\lambda \leftarrow \frac{E\nu}{(1+\nu)(1-2\nu)}$
$\mu \leftarrow \frac{E}{2(1+\nu)}$
$\gamma \leftarrow \lambda + 2\mu$
Hop $\leftarrow$ Hopoperator$(3, 3)$
Hop.H$(1, 1) \leftarrow$ Loperator$(3, [\gamma, 0, 0; 0, \mu, 0; 0, 0, \mu], \mathbf{0}, \mathbf{0}, 0)$
Hop.H$(1, 2) \leftarrow$ Loperator$(3, [0, \lambda, 0; \mu, 0, 0; 0, 0, 0], \mathbf{0}, \mathbf{0}, 0)$
Hop.H$(1, 3) \leftarrow$ Loperator$(3, [0, 0, \lambda; 0, 0, 0; \mu, 0, 0], \mathbf{0}, \mathbf{0}, 0)$
Hop.H$(2, 1) \leftarrow$ Loperator$(3, [0, \mu, 0; \lambda, 0, 0; 0, 0, 0], \mathbf{0}, \mathbf{0}, 0)$
Hop.H$(2, 2) \leftarrow$ Loperator$(3, [\mu, 0, 0; 0, \gamma, 0; 0, 0, \mu], \mathbf{0}, \mathbf{0}, 0)$
Hop.H$(2, 3) \leftarrow$ Loperator$(3, [0, 0, 0; 0, 0, \lambda; 0, \mu, 0], \mathbf{0}, \mathbf{0}, 0)$
Hop.H$(3, 1) \leftarrow$ Loperator$(3, [0, 0, \mu; 0, 0, 0; \lambda, 0, 0], \mathbf{0}, \mathbf{0}, 0)$
Hop.H$(3, 2) \leftarrow$ Loperator$(3, [0, 0, 0; 0, 0, \mu; 0, \lambda, 0], \mathbf{0}, \mathbf{0}, 0)$
Hop.H$(3, 3) \leftarrow$ Loperator$(3, [\mu, 0, 0; 0, \mu, 0; 0, 0, \gamma], \mathbf{0}, \mathbf{0}, 0)$
pde $\leftarrow$ initPDE$(\text{Hop}, \mathcal{T}_h)$
pde $\leftarrow$ setBC_PDE$(\text{pde}, 1, 1:3, \texttt{'Dirichlet'}, \boldsymbol{x} \to \mathbf{0})$
pde.$f \leftarrow \boldsymbol{x} \to [0, 0, -1]$
$\boldsymbol{x} \leftarrow$ SolvePDE(pde)

We give respectively in Listings 5 and 6 the corresponding Matlab/Octave and Python codes.

Listing 5: **3D elasticity, Matlab/Octave code**

```
fprintf('1. Reading of the mesh\n');
Th=GetMesh3DOpt('elasticity3D-10.mesh','format','medit');
fprintf('2. Definition of the BVP\n');
E = 21.5e4; nu = 0.29;
mu= E/(2*(1+nu));
lambda = E*nu/((1+nu)*(1-2*nu));
H=buildHoperator(3,3,'name','StiffElas','lambda',lambda,'mu',mu);
pde=initPDE(H,Th);
pde.f={0,0,-1};
pde=setBC_PDE(pde,1 ,1:3, 'Dirichlet',{0,0,0});
fprintf('3. Solving BVP\n');
x=solvePDE(pde);
```

Listing 6: **3D elasticity, Python code**

```
print('1. Reading of the mesh')
Th=readFreeFEM3D('elasticity3D-10.mesh')
print('2. Definition of the BVP')
E = 21.5e4; nu = 0.29
mu= E/(2*(1+nu))
lam = E*nu/((1+nu)*(1-2*nu))
gam=lam+2*mu
Hop=Hoperator(d=3,m=3)
Hop.H[0][0]=Loperator(d=3,A=[[gam,None,None],[None,mu,None],[None,None,mu]])
Hop.H[0][1]=Loperator(d=3,A=[[None,lam,None],[mu,None,None],[None,None,None]])
Hop.H[0][2]=Loperator(d=3,A=[[None,None,lam],[None,None,None],[mu,None,None]])
Hop.H[1][0]=Loperator(d=3,A=[[None,mu,None],[lam,None,None],[None,None,None]])
Hop.H[1][1]=Loperator(d=3,A=[[mu,None,None],[None,gam,None],[None,None,mu]])
Hop.H[1][2]=Loperator(d=3,A=[[None,None,None],[None,None,lam],[None,mu,None]])
Hop.H[2][0]=Loperator(d=3,A=[[None,None,mu],[None,None,None],[lam,None,None]])
Hop.H[2][1]=Loperator(d=3,A=[[None,None,None],[None,None,mu],[None,lam,None]])
Hop.H[2][2]=Loperator(d=3,A=[[mu,None,None],[None,mu,None],[None,None,gam]])
# One can also use the preset operator function
#    Hop=StiffElasHoperators(d,lam,mu)
pde=initPDE(Hop,Th)
pde.f=[0,0,-1]
pde=setBC_PDE(pde,1,[0,1,2],'Dirichlet',[0,0,0],None)
print('3. Solving BVP')
x=solvePDE(pde,split=True)
```

The displacement scaled by a factor 100 for a given mesh is shown on Figure 11.

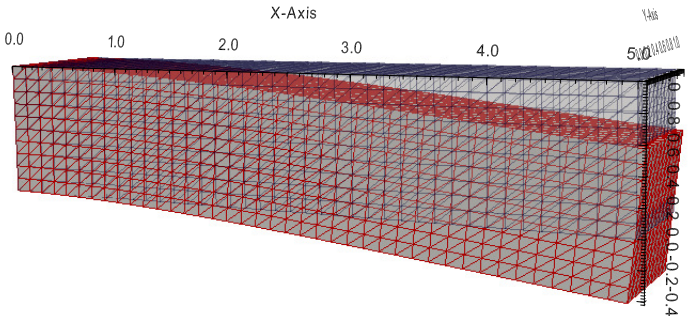


Figure 11: Result for the 3D elasticity problem

3.3.2 Stationary heat with potential flow in 2D

Let Γ_1 be the unit circle, Γ_{10} be the circle with center point $(0,0)$ and radius 0.3. Let Γ_{20}, Γ_{21}, Γ_{22} and Γ_{23} be the circles with radius 0.1 and respectively with center point $(0,-0.7)$, $(0,0.7)$, $(-0.7,0)$ and $(0.7,0)$. The domain $\Omega \subset \mathbb{R}^2$ is defined as the inner of Γ_1 and the outer of all other circles (see Figure 12).

The 2D problem to solve is the following

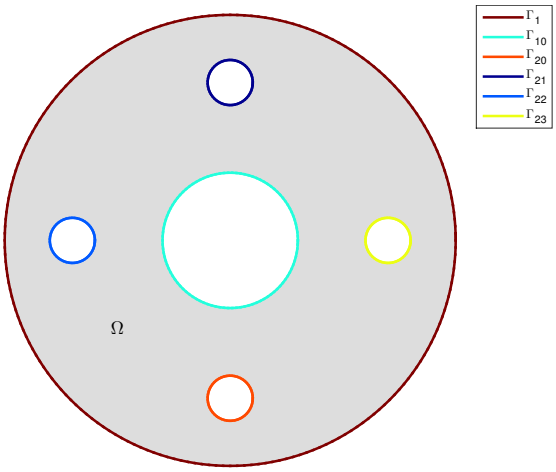


Figure 12: Domain and boundaries

2D problem : stationary heat with potential flow

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$
\begin{aligned}
-\operatorname{div}(\alpha \nabla u) + \langle \boldsymbol{V}, \nabla u \rangle + \beta u &= 0 \ \text{ in } \Omega \subset \mathbb{R}^2, && (3.39)\\
u &= 20 * \boldsymbol{x}_2 \ \text{ on } \Gamma_{21}, && (3.40)\\
u &= 0 \ \text{ on } \Gamma_{22} \cup \Gamma_{23}, && (3.41)\\
\frac{\partial u}{\partial n} &= 0 \ \text{ on } \Gamma_1 \cup \Gamma_{10} \cup \Gamma_{20} && (3.42)
\end{aligned}
$$

where Ω and its boundaries are given in Figure 12. This problem is well posed if $\alpha(\boldsymbol{x}) > 0$ and $\beta(\boldsymbol{x}) \geqslant 0$.
We choose α and β in Ω as :

$$
\begin{aligned}
\alpha(\boldsymbol{x}) &= 0.1 + \boldsymbol{x}_2^2,\\
\beta(\boldsymbol{x}) &= 0.01
\end{aligned}
$$

The potential flow is the velocity field $\boldsymbol{V} = \nabla \phi$ where the scalar function ϕ is the velocity potential solution of the BVP

Velocity potential in 2D

Find $\phi \in \mathrm{H}^2(\Omega)$ such that

$$
\begin{aligned}
-\Delta \phi &= 0 \ \text{ in } \Omega, && (3.43)\\
\phi &= -20 \ \text{ on } \Gamma_{21}, && (3.44)\\
\phi &= 20 \ \text{ on } \Gamma_{20}, && (3.45)\\
\frac{\partial \phi}{\partial n} &= 0 \ \text{ on } \Gamma_1 \cup \Gamma_{23} \cup \Gamma_{22} && (3.46)
\end{aligned}
$$

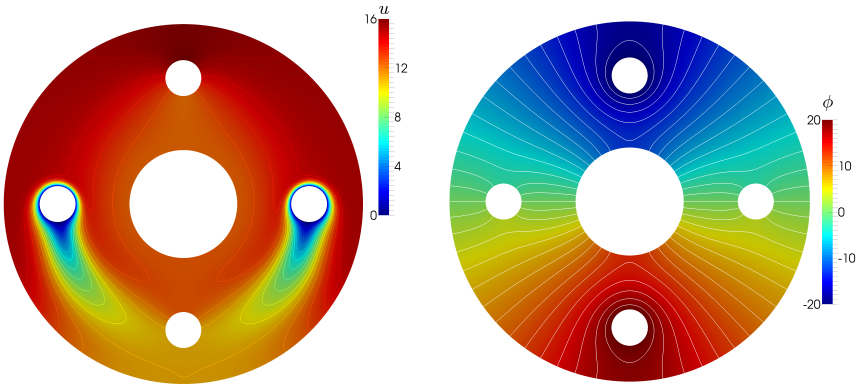
Then the potential flow $\boldsymbol{V}$ is solution of

Potential flow in 2D

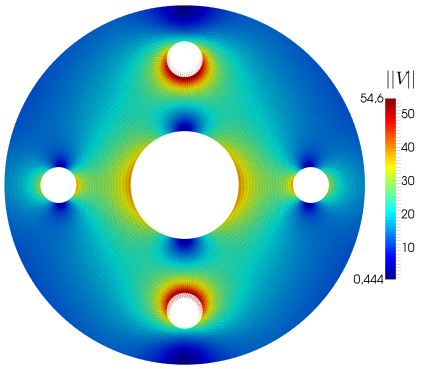
Find $\boldsymbol{V} = (\boldsymbol{V}_1, \boldsymbol{V}_2) \in \mathrm{H}^1(\Omega) \times \mathrm{H}^1(\Omega)$ such that

$$\boldsymbol{V} = \nabla \phi \text{ in } \Omega, \tag{3.47}$$

For a given mesh, the numerical result for heat u is represented in Figure 13a, velocity potential ϕ and potential flow $\boldsymbol{V}$ are shown on Figure 13.



(a) heat u (b) Velocity potential ϕ



(c) Potential flow $\boldsymbol{V}$

Figure 13: Stationary heat with potential flow in 2D

Now we will present two manners of solving these problems using `vecFEMP1` codes.

Method 1 : split in three parts

The 2D potential velocity problem (3.43)-(3.46) can be equivalently expressed as the scalar BVP (2.2)-(2.4) :

We present now to

2D potential velocity as a *scalar* BVP

Find $\phi \in \mathrm{H}^2(\Omega)$ such that

$$\begin{aligned} \mathcal{L}(\phi) &= f && \text{in } \Omega, \\ u &= g^D && \text{on } \Gamma^D, \\ \frac{\partial u}{\partial n_{\mathcal{L}}} + a^R u &= g^R && \text{on } \Gamma^R. \end{aligned}$$

where

- $\mathcal{L} := \mathcal{L}_{\mathbb{I},\mathbf{0},\mathbf{0},0}$, and then the conormal derivative of u is given by

$$\frac{\partial u}{\partial n_{\mathcal{L}}} := \langle \mathbb{A} \nabla u, \boldsymbol{n} \rangle - \langle \boldsymbol{b} u, \boldsymbol{n} \rangle = \frac{\partial u}{\partial n}.$$

- $f(\boldsymbol{x}) := 0$
- $\Gamma^D = \Gamma_{20} \cup \Gamma_{21}$
- $\Gamma^R = \Gamma_1 \cup \Gamma_{23} \cup \Gamma_{22}$
- $g^D := 20$ on Γ_{20}, and $g^D := -20$ on Γ_{21}
- $g^R = a^R := 0$ on Γ^R. and $g^D := -20$ on Γ_{21}

The algorithm using the toolbox for solving (3.43)-(3.46) is the following:

Algorithm 3.10 Velocity Potential in 2D

```
1: Dop ← LOPERATOR(𝕀, 0, 0, 0)                    ▷ Stiffness operator
2: pde ← INITPDE(Dop, 𝒯_h)
3: pde ← SETBCLABEL(pde, 20, 1, 'Dirichlet', 20)   ▷ u = 20 on Γ_20
4: pde ← SETBCLABEL(pde, 21, 1, 'Dirichlet', −20)  ▷ u = −20 on Γ_21
5: x_φ ← SOLVEPDE(pde)
```

Now to compute $\boldsymbol{V}$, we can write the potential flow problem (3.47) with $\mathcal{H}$-operators as

$$\mathcal{A} \begin{pmatrix} \boldsymbol{V}_1 \\ \boldsymbol{V}_2 \end{pmatrix} = \mathcal{B} \begin{pmatrix} \phi \\ \phi \end{pmatrix}$$

where

$$\mathcal{A} = \begin{pmatrix} \mathcal{L}_{\mathbb{O}_2,\mathbf{0}_2,\mathbf{0}_2,1} & 0 \\ 0 & \mathcal{L}_{\mathbb{O}_2,\mathbf{0}_2,\mathbf{0}_2,1} \end{pmatrix} \quad \text{and} \quad \mathcal{B} = \begin{pmatrix} \mathcal{L}_{\mathbb{O}_2,\mathbf{0}_2,(1,0)^t,1} & 0 \\ 0 & \mathcal{L}_{\mathbb{O}_2,\mathbf{0}_2,(0,1)^t,0} \end{pmatrix}$$

The algorithm using the toolbox for solving this problem is the following:

Algorithm 3.11 Potential flow in 2D

1: $\texttt{Aop} \leftarrow \textsc{InitHoperator}(2,2)$
2: $\texttt{Aop}.H(1,1) \leftarrow \textsc{Loperator}(2,\mathbb{O}_2,\mathbf{0},\mathbf{0},1)$
3: $\texttt{Aop}.H(2,2) \leftarrow \textsc{Loperator}(2,\mathbb{O}_2,\mathbf{0},\mathbf{0},1)$
4: $\texttt{Bop} \leftarrow \textsc{InitHoperator}(2,2)$
5: $\texttt{Bop}.H(1,1) \leftarrow \textsc{Loperator}(2,\mathbb{O}_2,\mathbf{0},\begin{pmatrix}1\\0\end{pmatrix},0)$
6: $\texttt{Bop}.H(2,2) \leftarrow \textsc{Loperator}(2,\mathbb{O}_2,\mathbf{0},\begin{pmatrix}0\\1\end{pmatrix},0)$
7: $\mathbb{A} \leftarrow \textsc{HAssemblyP1\_OptV3}(\texttt{Aop},\mathcal{T}_h)$
8: $\mathbb{B} \leftarrow \textsc{HAssemblyP1\_OptV3}(\texttt{Bop},\mathcal{T}_h)$
9: $\boldsymbol{b} \leftarrow \mathbb{B}\begin{pmatrix}\boldsymbol{x}_\phi\\\boldsymbol{x}_\phi\end{pmatrix}$
10: $\boldsymbol{V} \leftarrow \textsc{Solve}(\mathbb{A},\boldsymbol{b})$ ▷ Solve the linear system $\mathbb{A}\boldsymbol{V} = \boldsymbol{b}$

Finally, the stationary heat BVP (3.39)-(3.42) can be equivalently expressed as the scalar BVP (2.2)-(2.4) :

2D stationary heat as a *scalar* BVP

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$\begin{aligned} \mathcal{L}(u) &= f && \text{in } \Omega,\\ u &= g^D && \text{on } \Gamma^D,\\ \frac{\partial u}{\partial n_{\mathcal{L}}} + a^R u &= g^R && \text{on } \Gamma^R. \end{aligned}$$

where

- $\mathcal{L} := \mathcal{L}_{\begin{pmatrix}\alpha & 0\\ 0 & \alpha\end{pmatrix},\mathbf{0},\boldsymbol{V},\beta}$, and then the conormal derivative of u is given by

$$\frac{\partial u}{\partial n_{\mathcal{L}}} := \langle \mathbb{A} \bigtriangledown u, \boldsymbol{n}\rangle - \langle \boldsymbol{b}u, \boldsymbol{n}\rangle = \alpha \frac{\partial u}{\partial n}.$$

- $f := 0$
- $\Gamma^D = \Gamma_{21} \cup \Gamma_{22} \cup \Gamma_{23}$
- $\Gamma^R = \Gamma_1 \cup \Gamma_{10} \cup \Gamma_{20}$
- $g^D(x,y) := 20y$ on Γ_{21}, and $g^D := 0$ on $\Gamma_{22} \cup \Gamma_{23}$
- $g^R := 0$ and $a^R := 0$ on Γ^R

The algorithm using the toolbox for solving (3.39)-(3.42) is the following:

Algorithm 3.12 Stationary heat in 2D

1: $\texttt{Lop} \leftarrow \textsc{Loperator}(\begin{pmatrix}\alpha & 0\\ 0 & \alpha\end{pmatrix},\mathbf{0},\boldsymbol{V},\beta)$
2: pde $\leftarrow$ initPDE(Lop, $\mathcal{T}_h$)
3: pde $\leftarrow$ setBCLabel(pde, 21, 1, 'Dirichlet', $\boldsymbol{x} \mapsto 20\boldsymbol{x}_2$)
4: pde $\leftarrow$ setBCLabel(pde, 22, 1, 'Dirichlet', 0)
5: pde $\leftarrow$ setBCLabel(pde, 23, 1, 'Dirichlet', 0)
6: $\boldsymbol{u} \leftarrow$ SolvePDE(pde)

We give respectively in Listing 7 and 8 the corresponding Matlab/Octave and Python codes.

Listing 7: **Stationary heat with potential flow in 2D, Matlab/Octave code (method 1)**

```
fprintf('1. Reading of the mesh\n');
Th=GetMesh2DOpt('FlowVelocity2D01-50.msh');
fprintf('2.a) Definition of a 2D velocity potential BVP\n');
Lop=Loperator(Th.d,{1,[];[],1},[],[],[]);
pde=initPDE(Lop,Th);
pde=setBC_PDE(pde,20,1,'Dirichlet',20);
pde=setBC_PDE(pde,21,1,'Dirichlet',-20);
fprintf('2.b) Solving 2D velocity potential BVP\n')
phi=solvePDE(pde);
fprintf('3) Setting/Solving 2D velocity field problem\n');
m=2;
Hop=Hoperator(d,m);
Hop.H{1,1}=Loperator(d,[],[],[],1);
Hop.H{2,2}=Loperator(d,[],[],[],1);
Bop=Hoperator(d,m);
Bop.H{1,1}=Loperator(d,[],[],{1;0},[]);
Bop.H{2,2}=Loperator(d,[],[],{0;1},[]);
A=HAssemblyP1_OptV3(Th,Hop);
B=HAssemblyP1_OptV3(Th,Bop);
U=A\(B*[phi;phi]);
V=splitSol(U,2,Th.nq);
fprintf('4.a) Definition of a 2D stationary heat BVP with potential flow\n');
af=@(x,y) 0.1+y.^2;
Dop=Loperator(Th.d,{af,[];[],af},[],{V{1},V{2}},0.01);
pdeHeat=initPDE(Dop,Th);
pdeHeat=setBC_PDE(pdeHeat,21,1,'Dirichlet', @(x,y) 20*y   );
pdeHeat=setBC_PDE(pdeHeat,22,1,'Dirichlet', 0   );
pdeHeat=setBC_PDE(pdeHeat,23,1,'Dirichlet', 0   );
fprintf('4.b) Solving 2D stationary heat BVP with potential flow\n');
u=solvePDE(pdeHeat);
```

Listing 8: **Stationary heat with potential flow in 2D, Python code (method 1)**

```
d=2
print('1. Reading of the mesh')
Th=readFreeFEM("FlowVelocity2D01-50.msh")
print("2.a) Definition of a 2D velocity potential BVP")
Lop=Loperator(d=Th.d,A=[[1,None],[None,1]])
pde=initPDE(Lop,Th);
pde=setBC_PDE(pde,20,0,"Dirichlet",20,None);
pde=setBC_PDE(pde,21,0,"Dirichlet",-20,None);
print("2.b) Solving 2D velocity potential BVP")
phi=solvePDE(pde)
print("3. Setting/Solving 2D velocity field problem")
Hop=Hoperator(d=2,m=2)
Hop.H[0][0]=Loperator(d=d,a0=1)
Hop.H[1][1]=Loperator(d=d,a0=1)
Bop=Hoperator(d=2,m=2)
Bop.H[0][0]=Loperator(d=d,c=[1,0])
Bop.H[1][1]=Loperator(d=d,c=[0,1])
A=HAssemblyP1_OptV3(Th,Hop,1)
B=HAssemblyP1_OptV3(Th,Bop,1)
b=B*np.hstack([phi,phi])
U=spsolve(A,b)
V=splitSol(U,2,Th.nq)
print('4.a) Definition of a 2D stationary heat BVP with potential flow')
af=lambda x,y: 0.1+y**2;
Lop=Loperator(d=Th.d,A=[[af,None],[None,af]],c=[V[0],V[1]],a0=0.01);
pdeHeat=initPDE(Lop,Th)
pdeHeat=setBC_PDE(pdeHeat,21,0,'Dirichlet', lambda x,y: 20*y )
pdeHeat=setBC_PDE(pdeHeat,22,0,'Dirichlet',  0)
pdeHeat=setBC_PDE(pdeHeat,23,0,'Dirichlet',  0)
print('4.b) Solving 2D stationary heat PDE with potential flow')
u=solvePDE(pdeHeat)
```

Method 2 : have fun with $\mathcal{H}$-operators

We can merged velocity potential BVP (3.43)-(3.46) and potential flow to obtain the new BVP

Velocity potential and potential flow in 2D

Find $\phi \in \mathrm{H}^2(\Omega)$ and $\boldsymbol{V} = (\boldsymbol{V}_1, \boldsymbol{V}_2) \in \mathrm{H}^1(\Omega) \times \mathrm{H}^1(\Omega)$ such that

$$
\begin{aligned}
-\left(\frac{\partial \boldsymbol{V}_1}{\partial x} + \frac{\partial \boldsymbol{V}_2}{\partial y}\right) &= 0 \ \text{ in } \Omega, && (3.48)\\
\boldsymbol{V}_1 - \frac{\partial \phi}{\partial x} &= 0 \ \text{ in } \Omega, && (3.49)\\
\boldsymbol{V}_2 - \frac{\partial \phi}{\partial y} &= 0 \ \text{ in } \Omega, && (3.50)\\
\phi &= -20 \ \text{ on } \Gamma_{21}, && (3.51)\\
\phi &= 20 \ \text{ on } \Gamma_{20}, && (3.52)\\
\frac{\partial \phi}{\partial n} &= 0 \ \text{ on } \Gamma_1 \cup \Gamma_{23} \cup \Gamma_{22} && (3.53)
\end{aligned}
$$

We can also replace (3.48) by $-\Delta\phi = 0$.

Let $\boldsymbol{w} = \begin{pmatrix} \phi \\ \boldsymbol{V}_1 \\ \boldsymbol{V}_2 \end{pmatrix}$, the previous problem (3.48)-(3.53) can be equivalently expressed as the vector BVP (2.10)-(2.12) :

Vector BVP

Find $\boldsymbol{w} = (\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3) \in (\mathrm{H}^2(\Omega))^3$ such that

$$
\begin{aligned}
\mathcal{H}(\boldsymbol{w}) &= \boldsymbol{f} && \text{in } \Omega, && (3.54)\\
\boldsymbol{w}_\alpha &= g_\alpha^D && \text{on } \Gamma_\alpha^D, \ \forall \alpha \in [\![1,3]\!], && (3.55)\\
\frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{H}_\alpha}} + a_\alpha^R \boldsymbol{w}_\alpha &= g_\alpha^R && \text{on } \Gamma_\alpha^R, \ \forall \alpha \in [\![1,3]\!], && (3.56)
\end{aligned}
$$

where $\Gamma_\alpha^R = \Gamma_\alpha^D = \varnothing$ for all $\alpha \in \{2,3\}$ (no boundary conditions on $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$) and

- $\mathcal{H}$ is the 3-by-3 operator defined by

$$
\mathcal{H} = \begin{pmatrix} 0 & \mathcal{L}_{\mathbb{O},-\boldsymbol{e}_1,\boldsymbol{0},0} & \mathcal{L}_{\mathbb{O},-\boldsymbol{e}_2,\boldsymbol{0},0} \\ \mathcal{L}_{\mathbb{O},\boldsymbol{0},-\boldsymbol{e}_1,0} & \mathcal{L}_{\mathbb{O},\boldsymbol{0},\boldsymbol{0},1} & 0 \\ \mathcal{L}_{\mathbb{O},\boldsymbol{0},-\boldsymbol{e}_2,0} & 0 & \mathcal{L}_{\mathbb{O},\boldsymbol{0},\boldsymbol{0},1} \end{pmatrix}
$$

its conormal derivative are given by

$$
\begin{aligned}
&\frac{\partial \boldsymbol{w}_1}{\partial n_{\mathcal{H}_{1,1}}} = 0, && \frac{\partial \boldsymbol{w}_2}{\partial n_{\mathcal{H}_{1,2}}} = \boldsymbol{w}_2 \boldsymbol{n}_1, && \frac{\partial \boldsymbol{w}_3}{\partial n_{\mathcal{H}_{1,3}}} = \boldsymbol{w}_3 \boldsymbol{n}_2,\\
&\frac{\partial \boldsymbol{w}_1}{\partial n_{\mathcal{H}_{2,1}}} = 0, && \frac{\partial \boldsymbol{w}_2}{\partial n_{\mathcal{H}_{2,2}}} = 0, && \frac{\partial \boldsymbol{w}_3}{\partial n_{\mathcal{H}_{2,3}}} = 0\\
&\frac{\partial \boldsymbol{w}_1}{\partial n_{\mathcal{H}_{3,1}}} = 0, && \frac{\partial \boldsymbol{w}_2}{\partial n_{\mathcal{H}_{3,2}}} = 0, && \frac{\partial \boldsymbol{w}_3}{\partial n_{\mathcal{H}_{3,3}}} = 0.
\end{aligned}
$$

So we obtain

$$
\frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{H}_1}} \overset{\text{def}}{=} \sum_{\alpha=1}^{3} \frac{\partial \boldsymbol{w}_\alpha}{\partial n_{\mathcal{H}_{1,\alpha}}} = \langle \boldsymbol{V}, \boldsymbol{n} \rangle = \frac{\partial \phi}{\partial \boldsymbol{n}}, \qquad (3.57)
$$

and

$$\frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{H}_2}} = \frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{H}_3}} := 0. \tag{3.58}$$

From (3.58), we cannot impose boundary conditions on components 2 and 3.

- $\boldsymbol{f} := \boldsymbol{0}$
- $\Gamma_1^D = \Gamma_{20} \cup \Gamma_{21}$ and $\Gamma_1^R = \Gamma_1 \cup \Gamma_{10} \cup \Gamma_{22} \cup \Gamma_{23}$
- $g_1^D := 20$ on Γ_{20}, and $g_1^D := -20$ on Γ_{21}
- $g_1^R = a_1^R := 0$ on Γ_1^R

The solution of this vector BVP is given on lines 3 to 13 of Algorithm 3.13.

Algorithm 3.13 Stationary heat with potential velocity problem (method 2)

1: $\mathcal{T}_h \leftarrow$ getMesh(...) ▷ Load FreeFEM++ mesh
2: $\boldsymbol{e}_1 \leftarrow \begin{pmatrix}1\\0\end{pmatrix}$, $\boldsymbol{e}_2 \leftarrow \begin{pmatrix}0\\1\end{pmatrix}$
3: Hop $\leftarrow$ Hopoperator$(2,3)$
4: Hop.H$(1,2) \leftarrow$ Loperator$(\mathbb{O}_2, -\boldsymbol{e}_1, \boldsymbol{0}, 0)$
5: Hop.H$(1,3) \leftarrow$ Loperator$(\mathbb{O}_2, -\boldsymbol{e}_2, \boldsymbol{0}, 0)$
6: Hop.H$(2,1) \leftarrow$ Loperator$(\mathbb{O}_2, \boldsymbol{0}, -\boldsymbol{e}_1, 0)$
7: Hop.H$(2,2) \leftarrow$ Loperator$(\mathbb{O}_2, \boldsymbol{0}, \boldsymbol{0}, 1)$
8: Hop.H$(3,1) \leftarrow$ Loperator$(\mathbb{O}_2, \boldsymbol{0}, -\boldsymbol{e}_2, 0)$
9: Hop.H$(3,3) \leftarrow$ Loperator$(\mathbb{O}_2, \boldsymbol{0}, \boldsymbol{0}, 1)$
10: PDEflow $\leftarrow$ initPDE(Hop, $\mathcal{T}_h$)
11: PDEflow $\leftarrow$ setBC_PDE(PDEflow, 20, 1, 'Dirichlet', 20., $\varnothing$)
12: PDEflow $\leftarrow$ setBC_PDE(PDEflow, 21, 1, 'Dirichlet', $-20.$, $\varnothing$)
13: $[\boldsymbol{\phi}, \boldsymbol{V}_1, \boldsymbol{V}_2] \leftarrow$ SolvePDE(PDEflow)
14: $\alpha \leftarrow (x,y) \longmapsto 0.1 + y^2$
15: $g_{21} \leftarrow (x,y) \longmapsto 20y$
16: $\beta \leftarrow 0.01$
17: Dop $\leftarrow$ Loperator$\left(\begin{pmatrix}\alpha & 0\\ 0 & \alpha\end{pmatrix}, \boldsymbol{0}, \begin{pmatrix}\boldsymbol{V}_1\\ \boldsymbol{V}_2\end{pmatrix}, \beta\right)$
18: PDE $\leftarrow$ initPDE(Dop, $\mathcal{T}_h$) ▷ Set homogeneous 'Neumann' condition on all boundaries
19: PDE $\leftarrow$ setBC_PDE(PDE, 21, 1, 'Dirichlet', g_{21}, $\varnothing$) ▷ $u = 4$ on Γ_2
20: PDE $\leftarrow$ setBC_PDE(PDE, 22, 1, 'Dirichlet', 0, $\varnothing$) ▷ $u = -4$ on Γ_4
21: PDE $\leftarrow$ setBC_PDE(PDE, 23, 1, 'Dirichlet', 0, $\varnothing$ ▷ $u = 0$ on Γ_{20}
22: $\boldsymbol{u} \leftarrow$ SolvePDE(PDE)

We give respectively in Listing 9 and 10 the corresponding Matlab/Octave and Python codes.

Listing 9: **Stationary heat with potential flow in 2D, Matlab/Octave code (method 2)**

```
d=2;
fprintf('1. Reading of the mesh \n');
Th=GetMesh2DOpt('FlowVelocity2D01-50.msh');
fprintf('2. Setting 2D potential velocity/flow BVP\n');
Hop=Hoperator(d,3);
Hop.H{1,2}=Loperator(d,[],{-1,0},[],[]);
Hop.H{1,3}=Loperator(d,[],{0,-1},[],[]);
Hop.H{2,1}=Loperator(d,[],[],{-1,0},[]);
Hop.H{2,2}=Loperator(d,[],[],[],1);
Hop.H{3,1}=Loperator(d,[],[],{0,-1},[]);
Hop.H{3,3}=Loperator(d,[],[],[],1);
pdeFlow=initPDE(Hop,Th);
pdeFlow=setBC_PDE(pdeFlow,20,1,'Dirichlet',20);
pdeFlow=setBC_PDE(pdeFlow,21,1,'Dirichlet',-20);
fprintf('3. Solving 2D potential velocity/flow BVP\n')
U=solvePDE(pdeFlow,'split',true);
fprintf('4. Setting 2D stationary heat BVP with potential flow\n');
af=@(x,y) 0.1+y.^2;
Dop=Loperator(Th.d,{af,[];[],af},[],{U{2},U{3}},0.01);
pdeHeat=initPDE(Dop,Th);
pdeHeat=setBC_PDE(pdeHeat,21,1,'Dirichlet', @(x,y) 20*y  );
pdeHeat=setBC_PDE(pdeHeat,22,1,'Dirichlet', 0  );
pdeHeat=setBC_PDE(pdeHeat,23,1,'Dirichlet', 0  );
fprintf('5. Solving 2D stationary heat BVP with potential flow\n');
x=solvePDE(pdeHeat);
```

Listing 10: **Stationary heat with potential flow in 2D, Python code (method 2)**

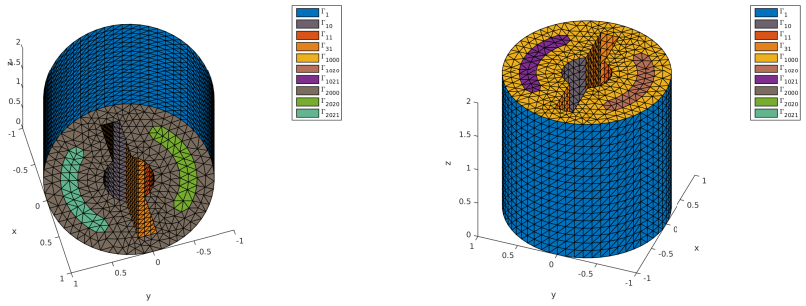
```
d=2;m=3;
print('1. Reading of the mesh')
Th=readFreeFEM("FlowVelocity2D01-50.msh")
print("2. Setting 2D potential velocity/flow BVP")
Hop=Hoperator(d=2,m=3)
#Hop.H[0][0]=Loperator(d=d,A=[[1,None],[None,1]])
Hop.H[0][1]=Loperator(d=d,b=[-1,None])
Hop.H[0][2]=Loperator(d=d,b=[None,-1])
Hop.H[1][0]=Loperator(d=d,c=[-1,None])
Hop.H[1][1]=Loperator(d=d,a0=1)
Hop.H[2][0]=Loperator(d=d,c=[None,-1])
Hop.H[2][2]=Loperator(d=d,a0=1)
pdeFlow=initPDE(Hop,Th);
pdeFlow=setBC_PDE(pdeFlow,20,0,"Dirichlet",20,None);
pdeFlow=setBC_PDE(pdeFlow,21,0,"Dirichlet",-20,None);
print("3. Solving 2D potential velocity/flow BVP")
U=solvePDE(pdeFlow,split=True)
print('4. Setting 2D stationary heat BVP with potential flow')
af=lambda x,y: 0.1+y**2;
Lop=Loperator(d=Th.d,A=[[af,None],[None,af]],c=[U[1],U[2]],a0=0.01);
pdeHeat=initPDE(Lop,Th)
pdeHeat=setBC_PDE(pdeHeat,21,0,'Dirichlet', lambda x,y: 20*y )
pdeHeat=setBC_PDE(pdeHeat,22,0,'Dirichlet', 0)
pdeHeat=setBC_PDE(pdeHeat,23,0,'Dirichlet', 0)
print('5. Solving 2D stationary heat PDE with potential flow')
u=solvePDE(pdeHeat)
```

3.3.3 Stationary heat with potential flow in 3D

Let $\Omega \subset \mathbb{R}^3$ be the cylinder given in Figure 14.

The bottom and top faces of the cylinder are respectively $\Gamma_{1000} \cup \Gamma_{1020} \cup \Gamma_{1021}$ and $\Gamma_{2000} \cup \Gamma_{2020} \cup \Gamma_{2021}$. The hole surface is $\Gamma_{10} \cup \Gamma_{11} \cup \Gamma_{31}$ where $\Gamma_{10} \cup \Gamma_{11}$ is the cylinder part and Γ_{31} the plane part.

The 3D problem to solve is the following



(a) Bottom face (b) Top face

Figure 14: Stationary heat with potential flow : 3d mesh

3D problem : stationary heat with potential flow

Find $u \in \mathrm{H}^2(\Omega)$ such that

$$
\begin{aligned}
-\operatorname{div}(\alpha \nabla u) + \langle \boldsymbol{V}, \nabla u \rangle + \beta u &= 0 \ \text{ in } \Omega \subset \mathbb{R}^3, && (3.59)\\
u &= 30 \ \text{ on } \Gamma_{1020} \cup \Gamma_{2020}, && (3.60)\\
u &= 10\delta_{|z-1|>0.5} \ \text{ on } \Gamma_{10}, && (3.61)\\
\frac{\partial u}{\partial n} &= 0 \ \text{ otherwise} && (3.62)
\end{aligned}
$$

where Ω and its boundaries are given in Figure 14. This problem is well posed if $\alpha(\boldsymbol{x}) > 0$ and $\beta(\boldsymbol{x}) \geqslant 0$.
We choose α and β in Ω as :

$$
\begin{aligned}
\alpha(\boldsymbol{x}) &= 1,\\
\beta(\boldsymbol{x}) &= 0.01
\end{aligned}
$$

The potential flow is the velocity field $\boldsymbol{V} = \nabla \phi$ where the scalar function ϕ is the velocity potential solution of the PDE :

Velocity potential in 3d

Find $\phi \in \mathrm{H}^2(\Omega)$ such that

$$
\begin{aligned}
-\Delta \phi &= 0 \ \text{ in } \Omega, && (3.63)\\
\phi &= 1 \ \text{ on } \Gamma_{1021} \cup \Gamma_{2021}, && (3.64)\\
\phi &= -1 \ \text{ on } \Gamma_{1020} \cup \Gamma_{2020}, && (3.65)\\
\frac{\partial \phi}{\partial n} &= 0 \ \text{ otherwise} && (3.66)
\end{aligned}
$$

To solve problem (3.59)-(3.62), we need to compute the velocity field $\boldsymbol{V}$. For that we can rewrite the potential flow problem (3.63)-(3.66), by introducing $\boldsymbol{V} = (\boldsymbol{V}_1, \boldsymbol{V}_2, \boldsymbol{V}_3)$ as unknowns :

Velocity potential and velocity field in 3d

Find $\phi \in \mathrm{H}^2(\Omega)$ and $\boldsymbol{V} \in \mathrm{H}^1(\Omega)^3$ such that

$$-\left(\frac{\partial \boldsymbol{V}_1}{\partial x} + \frac{\partial \boldsymbol{V}_2}{\partial y} + \frac{\partial \boldsymbol{V}_3}{\partial z}\right) = 0 \text{ in } \Omega, \tag{3.67}$$

$$\boldsymbol{V}_1 - \frac{\partial \phi}{\partial x} = 0 \text{ in } \Omega, \tag{3.68}$$

$$\boldsymbol{V}_2 - \frac{\partial \phi}{\partial y} = 0 \text{ in } \Omega, \tag{3.69}$$

$$\boldsymbol{V}_3 - \frac{\partial \phi}{\partial z} = 0 \text{ in } \Omega, \tag{3.70}$$

with boundary conditions (3.64) to (3.66).

We can also replace (3.67) by $-\Delta\phi = 0$.

Let $\boldsymbol{w} = \begin{pmatrix} \phi \\ \boldsymbol{V}_1 \\ \boldsymbol{V}_2 \\ \boldsymbol{V}_3 \end{pmatrix}$, the previous PDE can be written as a vector boundary value problem (see section 2.2) where the $\mathcal{H}$-operator is given by

$$\mathcal{H}(\boldsymbol{w}) = 0 \tag{3.71}$$

with

$$\mathcal{H}_{1,1} = 0, \quad \mathcal{H}_{1,2} = \mathcal{L}_{\mathbb{O},-\boldsymbol{e}_1,\boldsymbol{0},0}, \quad \mathcal{H}_{1,3} = \mathcal{L}_{\mathbb{O},-\boldsymbol{e}_2,\boldsymbol{0},0}, \quad \mathcal{H}_{1,4} = \mathcal{L}_{\mathbb{O},-\boldsymbol{e}_3,\boldsymbol{0},0}, \tag{3.72}$$

$$\mathcal{H}_{2,1} = \mathcal{L}_{\mathbb{O},\boldsymbol{0},-\boldsymbol{e}_1,0}, \quad \mathcal{H}_{2,2} = \mathcal{L}_{\mathbb{O},\boldsymbol{0},\boldsymbol{0},1}, \quad \mathcal{H}_{2,3} = 0, \quad \mathcal{H}_{2,4} = 0, \tag{3.73}$$

$$\mathcal{H}_{3,1} = \mathcal{L}_{\mathbb{O},\boldsymbol{0},-\boldsymbol{e}_2,0}, \quad \mathcal{H}_{3,2} = 0, \quad \mathcal{H}_{3,3} = \mathcal{L}_{\mathbb{O},\boldsymbol{0},\boldsymbol{0},1}, \quad \mathcal{H}_{3,4} = 0, \tag{3.74}$$

$$\mathcal{H}_{4,1} = \mathcal{L}_{\mathbb{O},\boldsymbol{0},-\boldsymbol{e}_3,0}, \quad \mathcal{H}_{4,2} = 0, \quad \mathcal{H}_{4,3} = 0, \quad \mathcal{H}_{4,4} = \mathcal{L}_{\mathbb{O},\boldsymbol{0},\boldsymbol{0},1}, \tag{3.75}$$

and $\boldsymbol{e}_1 = (1,0,0)^t$, $\boldsymbol{e}_2 = (0,1,0)^t$, $\boldsymbol{e}_3 = (0,0,1)^t$.

The conormal derivatives are given by

$$\frac{\partial \boldsymbol{w}_1}{\partial n_{\mathcal{H}_{1,1}}} = 0, \quad \frac{\partial \boldsymbol{w}_1}{\partial n_{\mathcal{H}_{2,1}}} = 0, \quad \frac{\partial \boldsymbol{w}_1}{\partial n_{\mathcal{H}_{3,1}}} = 0, \quad \frac{\partial \boldsymbol{w}_1}{\partial n_{\mathcal{H}_{4,1}}} = 0,$$

$$\frac{\partial \boldsymbol{w}_2}{\partial n_{\mathcal{H}_{1,2}}} = \boldsymbol{V}_1\boldsymbol{n}_1, \quad \frac{\partial \boldsymbol{w}_2}{\partial n_{\mathcal{H}_{2,2}}} = 0, \quad \frac{\partial \boldsymbol{w}_2}{\partial n_{\mathcal{H}_{3,2}}} = 0, \quad \frac{\partial \boldsymbol{w}_2}{\partial n_{\mathcal{H}_{4,2}}} = 0,$$

$$\frac{\partial \boldsymbol{w}_3}{\partial n_{\mathcal{H}_{1,3}}} = \boldsymbol{V}_2\boldsymbol{n}_2, \quad \frac{\partial \boldsymbol{w}_3}{\partial n_{\mathcal{H}_{2,3}}} = 0, \quad \frac{\partial \boldsymbol{w}_3}{\partial n_{\mathcal{H}_{3,3}}} = 0, \quad \frac{\partial \boldsymbol{w}_3}{\partial n_{\mathcal{H}_{4,3}}} = 0,$$

$$\frac{\partial \boldsymbol{w}_4}{\partial n_{\mathcal{H}_{1,4}}} = \boldsymbol{V}_3\boldsymbol{n}_3, \quad \frac{\partial \boldsymbol{w}_4}{\partial n_{\mathcal{H}_{2,4}}} = 0, \quad \frac{\partial \boldsymbol{w}_4}{\partial n_{\mathcal{H}_{3,4}}} = 0, \quad \frac{\partial \boldsymbol{w}_4}{\partial n_{\mathcal{H}_{4,4}}} = 0,$$

So we obtain

$$\sum_{\alpha=1}^{4} \frac{\partial \boldsymbol{w}_\alpha}{\partial n_{\mathcal{H}_{1,\alpha}}} = \langle \boldsymbol{V}, \boldsymbol{n} \rangle = \langle \nabla\phi, \boldsymbol{n} \rangle, \tag{3.76}$$

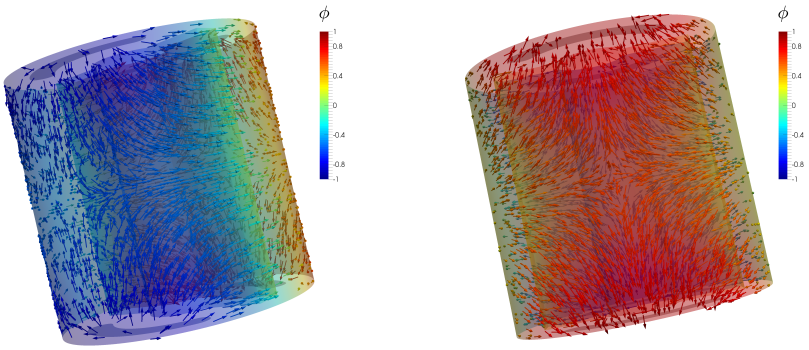
and

$$\sum_{\alpha=1}^{4} \frac{\partial \boldsymbol{w}_\alpha}{\partial n_{\mathcal{H}_{2,\alpha}}} = \sum_{\alpha=1}^{4} \frac{\partial \boldsymbol{w}_\alpha}{\partial n_{\mathcal{H}_{3,\alpha}}} = \sum_{\alpha=1}^{4} \frac{\partial \boldsymbol{w}_\alpha}{\partial n_{\mathcal{H}_{4,\alpha}}} = 0. \tag{3.77}$$

From (3.77), we cannot impose boundary conditions on components 2 to 4. Thus, with notation of section 2.2, we have $\Gamma_2^N = \Gamma_3^N = \Gamma_4^N = \Gamma$ with $g_2^N = g_3^N = g_4^N = 0$.

To take into account boundary conditions (3.64) to (3.66), we set $\Gamma_1^D = \Gamma_{1020} \cup \Gamma_{1021} \cup \Gamma_{2020} \cup \Gamma_{2021}$, $\Gamma_1^N = \Gamma \backslash \Gamma_1^D$ and $g_1^D = \delta_{\Gamma_{1020} \cup \Gamma_{2020}} - \delta_{\Gamma_{1021} \cup \Gamma_{2021}}$, $g_1^N = 0$.

The solution of this vector boundary value problem is given in lines 3 to 13 of Algorithm 3.14. A representation of velocity potential ϕ and potential flow $\boldsymbol{V}$ is given in Figure 15.



(a) ϕ and $\boldsymbol{V}$: first view (b) ϕ and $\boldsymbol{V}$: second view

Figure 15: HeatAndFlowVelocity3d01 problem

The operator in (3.59) is given by $\mathcal{L}_{\alpha\mathbb{I},\mathbf{0},\boldsymbol{V},\beta}$. The conormal derivative $\frac{\partial u}{\partial n_{\mathcal{L}}}$ is

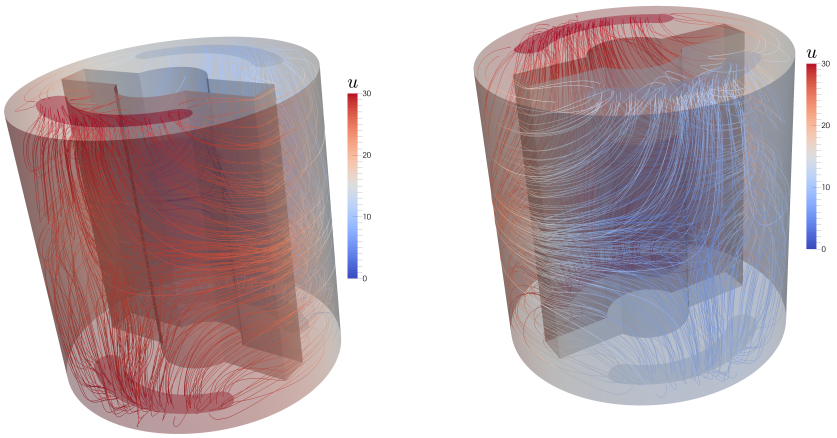
$$\frac{\partial u}{\partial n_{\mathcal{L}}} := \langle \mathbb{A} \nabla u, \boldsymbol{n} \rangle - \langle \boldsymbol{b} u, \boldsymbol{n} \rangle = \alpha \frac{\partial u}{\partial n}.$$

The algorithm using the toolbox for solving (3.67)-(3.70) is the following:

Algorithm 3.14 Stationary heat with potential velocity problem

1: $\mathcal{T}_h \leftarrow$ getMesh(...) ▷ Load FreeFEM++ mesh
2: $\boldsymbol{e}_1 \leftarrow \begin{pmatrix}1\\0\\0\end{pmatrix}$, $\boldsymbol{e}_2 \leftarrow \begin{pmatrix}0\\1\\0\end{pmatrix}$, $\boldsymbol{e}_3 \leftarrow \begin{pmatrix}0\\0\\1\end{pmatrix}$
3: Hop $\leftarrow$ Hopoperator$(3,4)$
4: Hop.H$(1,2) \leftarrow$ Loperator$(\mathbb{O}_3, -\boldsymbol{e}_1, \mathbf{0}, 0)$
5: Hop.H$(1,3) \leftarrow$ Loperator$(\mathbb{O}_3, -\boldsymbol{e}_2, \mathbf{0}, 0)$
6: Hop.H$(1,4) \leftarrow$ Loperator$(\mathbb{O}_3, -\boldsymbol{e}_3, \mathbf{0}, 0)$
7: Hop.H$(2,1) \leftarrow$ Loperator$(\mathbb{O}_3, \mathbf{0}, -\boldsymbol{e}_1, 0)$, Hop.H$(2,2) \leftarrow$ Loperator$(\mathbb{O}_3, \mathbf{0}, \mathbf{0}, 1)$
8: Hop.H$(3,1) \leftarrow$ Loperator$(\mathbb{O}_3, \mathbf{0}, -\boldsymbol{e}_2, 0)$, Hop.H$(3,3) \leftarrow$ Loperator$(\mathbb{O}_3, \mathbf{0}, \mathbf{0}, 1)$
9: Hop.H$(4,1) \leftarrow$ Loperator$(\mathbb{O}_3, \mathbf{0}, -\boldsymbol{e}_3, 0)$, Hop.H$(4,4) \leftarrow$ Loperator$(\mathbb{O}_3, \mathbf{0}, \mathbf{0}, 1)$
10: PDEflow $\leftarrow$ initPDE(Hop, $\mathcal{T}_h$)
11: PDEflow $\leftarrow$ setBC_PDE(PDEflow, 20, 1, 'Dirichlet', 20., $\varnothing$)
12: PDEflow $\leftarrow$ setBC_PDE(PDEflow, 21, 1, 'Dirichlet', $-20.$, $\varnothing$)
13: $[\boldsymbol{\phi}, \boldsymbol{V}_1, \boldsymbol{V}_2, \boldsymbol{V}_3] \leftarrow$ SolvePDE(PDEflow)
14: $\alpha \leftarrow (x,y,z) \longmapsto 1$
15: $g_{20} \leftarrow (x,y,z) \longmapsto 30$, $g_{10} \leftarrow (x,y,z) \longmapsto 10 * (|z-1| > 0.5)$
16: $\beta \leftarrow 0.01$
17: Dop $\leftarrow$ Loperator$\left(\begin{pmatrix}\alpha & 0 & 0\\ 0 & \alpha & 0\\ 0 & 0 & \alpha\end{pmatrix}, \mathbf{0}, \begin{pmatrix}\boldsymbol{V}_1\\ \boldsymbol{V}_2\\ \boldsymbol{V}_3\end{pmatrix}, \beta\right)$
18: PDE $\leftarrow$ initPDE(Dop, $\mathcal{T}_h$) ▷ Set homogeneous 'Neumann' condition on all boundaries
19: PDE $\leftarrow$ setBC_PDE(PDE, 1020, 1, 'Dirichlet', g_{20}, $\varnothing$)
20: PDE $\leftarrow$ setBC_PDE(PDE, 2022, 1, 'Dirichlet', g_{20}, $\varnothing$)
21: PDE $\leftarrow$ setBC_PDE(PDE, 10, 1, 'Dirichlet', g_{10}, $\varnothing$)
22: $\boldsymbol{u} \leftarrow$ SolvePDE(PDE)

1 The numerical solution for a given mesh is shown on Figure 16



(a) us solution with streamline : first viewview (b) us solution with streamline : second

Figure 16: HeatAndFlowVelocity3d01 problem

2 The biharmonic equation is the fourth-order partial PDE given by

3 $$\Delta^2 u = f \tag{3.78}$$

where $\Delta^2 u = \Delta(\Delta u) = \sum_{i=1}^{d}\sum_{j=1}^{d} \frac{\partial^4 u}{\partial x_i^2 \partial x_j^2} = \sum_{i=1}^{d} \frac{\partial^4 u}{\partial x_i^4} + 2\sum_{i=1}^{d}\sum_{j=i+1}^{d} \frac{\partial^4 u}{\partial x_i^2 \partial x_j^2}$

The boundary conditions on Γ can be

- *Clamped Plate* (CP) or *pure Dirichlet* type:

$$u = \frac{\partial u}{\partial \boldsymbol{n}} = g \tag{3.79}$$

- *Simply Supported Plate* (SSP) or *Navier* type :

$$u = \Delta u = g \tag{3.80}$$

- *Pure Hinged Plate* (PHP) or *Steklov* type :

$$u = \Delta u - (1-\sigma) K \frac{\partial u}{\partial \boldsymbol{n}} = g \tag{3.81}$$

- *Cahn-Hilliard* (CH) type

$$\frac{\partial u}{\partial n} = \frac{\partial \Delta u}{\partial n} = g \tag{3.82}$$

Link with $\mathcal{H}$-operator and boundary conditions

Classically the fourth-order PDE (3.78) is converted to the two second-order PDE

$$-\Delta u = v \tag{3.83}$$
$$-\Delta v = f \tag{3.84}$$

These two equations can be equivalently written as

$$\mathcal{G}\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix} \quad \text{or} \quad \mathcal{K}\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 0 \\ f \end{pmatrix} \tag{3.85}$$

where $\mathcal{G}$ and $\mathcal{K}$ are the $\mathcal{H}$-operators defined by

$$\mathcal{G} = \begin{pmatrix} 0 & \mathcal{L}_{\mathbb{I},\mathbf{0},\mathbf{0},0} \\ \mathcal{L}_{\mathbb{I},\mathbf{0},\mathbf{0},0} & \mathcal{L}_{\mathbb{O},\mathbf{0},\mathbf{0},-1} \end{pmatrix} \quad \text{and} \quad \mathcal{K} = \begin{pmatrix} \mathcal{L}_{\mathbb{I},\mathbf{0},\mathbf{0},0} & \mathcal{L}_{\mathbb{O},\mathbf{0},\mathbf{0},-1} \\ \mathcal{L}_{\mathbb{O},\mathbf{0},\mathbf{0},0} & \mathcal{L}_{\mathbb{I},\mathbf{0},\mathbf{0},0} \end{pmatrix} \tag{3.86}$$

Let $\boldsymbol{w} = (u, v)$. From (3.85), the components of the conormal derivative of $\boldsymbol{w}$ defined in (2.13) are given by

$$\begin{aligned} \frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{K}_1}} &\overset{\text{def}}{=} \sum_{\beta=1}^{2} \frac{\partial \boldsymbol{w}_\beta}{\partial n_{\mathcal{K}_{1,\beta}}} = \sum_{\beta=1}^{2} \left\langle \mathbb{A}^{1,\beta} \nabla \boldsymbol{w}_\beta, \boldsymbol{n} \right\rangle - \left\langle \boldsymbol{b}^{1,\beta} \boldsymbol{u}_\beta, \boldsymbol{n} \right\rangle \\ &= \left\langle \mathbb{I} \nabla \boldsymbol{w}_1, \boldsymbol{n} \right\rangle = \left\langle \nabla u, \boldsymbol{n} \right\rangle \\ &= \frac{\partial u}{\partial \boldsymbol{n}} = \frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{G}_2}} \end{aligned} \tag{3.87}$$

and

$$\begin{aligned} \frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{K}_2}} &\overset{\text{def}}{=} \sum_{\beta=1}^{2} \frac{\partial \boldsymbol{w}_\beta}{\partial n_{\mathcal{K}_{2,\beta}}} = \sum_{\beta=1}^{2} \left\langle \mathbb{A}^{2,\beta} \nabla \boldsymbol{w}_\beta, \boldsymbol{n} \right\rangle - \left\langle \boldsymbol{b}^{2,\beta} \boldsymbol{u}_\beta, \boldsymbol{n} \right\rangle \\ &= \left\langle \mathbb{I} \nabla \boldsymbol{w}_2, \boldsymbol{n} \right\rangle = \left\langle \nabla v, \boldsymbol{n} \right\rangle \\ &= \frac{\partial v}{\partial \boldsymbol{n}} = \frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{G}_1}} \end{aligned} \tag{3.88}$$

From *Vector* BVP (2.10)-(2.12), using $\mathcal{G}$ operator one can impose the following boundaries conditions with $\Gamma_\alpha^D \cap \Gamma_\alpha^R = \varnothing$, $\forall \alpha \in [\![1,2]\!]$,

$$\begin{aligned} \boldsymbol{w}_\alpha &= g_\alpha^D && \text{on } \Gamma_\alpha^D,\ \forall \alpha \in [\![1,2]\!], \\ \frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{G}_\alpha}} + a_\alpha^R \boldsymbol{w}_\alpha &= g_\alpha^R && \text{on } \Gamma_\alpha^R,\ \forall \alpha \in [\![1,2]\!] \end{aligned}$$

i.e.

$$\begin{aligned} u &= g_1^D && \text{on } \Gamma_1^D, & v &= g_2^D && \text{on } \Gamma_2^D, \\ \frac{\partial v}{\partial \boldsymbol{n}} + a_1^R u &= g_1^R && \text{on } \Gamma_1^R, & \frac{\partial u}{\partial \boldsymbol{n}} + a_2^R v &= g_2^R && \text{on } \Gamma_2^R \end{aligned}$$

With $\mathcal{K}$ operator one can impose the following boundaries conditions

$$\begin{aligned} \boldsymbol{w}_\alpha &= g_\alpha^D && \text{on } \Gamma_\alpha^D,\ \forall \alpha \in [\![1,2]\!], \\ \frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{K}_\alpha}} + a_\alpha^R \boldsymbol{w}_\alpha &= g_\alpha^R && \text{on } \Gamma_\alpha^R,\ \forall \alpha \in [\![1,2]\!] \end{aligned}$$

i.e.

$$\begin{aligned} u &= g_1^D && \text{on } \Gamma_1^D, & v &= g_2^D && \text{on } \Gamma_2^D, \\ \frac{\partial u}{\partial \boldsymbol{n}} + a_1^R u &= g_1^R && \text{on } \Gamma_1^R, & \frac{\partial v}{\partial \boldsymbol{n}} + a_2^R v &= g_2^R && \text{on } \Gamma_2^R \end{aligned}$$

One can neither impose *clamped plate* (3.79) nor *Pure Hinged Plate* (3.81) boundary conditions with $\mathcal{K}$ operator. This is why thereafter we will only use the $\mathcal{G}$ operator.

Clamped plate problem

***Usual* BVP 1 : Clamped plate problem**

Find u such that

$$\begin{cases} \Delta^2 u &= f, \quad \text{in } \Omega \subset \mathbb{R}^d,\ d = 2 \\ u &= 0, \quad \text{on } \Gamma \\ \dfrac{\partial u}{\partial \boldsymbol{n}} &= 0. \quad \text{on } \Gamma \end{cases} \tag{3.89}$$

Using the operator $\mathcal{G}$ defined in (3.86) and its conormal derivatives (3.87)-(3.88), we can equivalently write the vector BVP associated with (3.89) as

***Vector* BVP 2 : clamped plate problem (3.89) with $\mathcal{G}$ operator**

Find $\boldsymbol{w} = (\boldsymbol{w}_1, \boldsymbol{w}_2) \in (\mathrm{H}^2(\Omega))^2$ such that

$$\begin{aligned} \mathcal{G}(\boldsymbol{w}) &= \begin{pmatrix} f \\ 0 \end{pmatrix} && \text{in } \Omega, \\ \boldsymbol{w}_1 &= 0 && \text{on } \Gamma_1^D = \Gamma \text{ (so } \Gamma_1^R = \varnothing \text{)} \\ \frac{\partial \boldsymbol{w}}{\partial n_{\mathcal{G}_2}} &= 0 && \text{on } \Gamma_2^R = \Gamma \text{ (so } \Gamma_2^D = \varnothing \text{)} \end{aligned}$$

Remark 2. *We cannot use the operator $\mathcal{K}$ defined in* (3.86) *due to a boundary condition trouble. Indeed* $\dfrac{\partial \boldsymbol{w}}{\partial n_{\mathcal{K}_1}} = \dfrac{\partial u}{\partial n}$ *and we cannot set a Dirichlet condition* $\boldsymbol{w}_1 = 0$ *on* $\Gamma_1^D = \Gamma$ *with a Neumann condition* $\dfrac{\partial \boldsymbol{w}}{\partial n_{\mathcal{K}_1}} = 0$ *on* $\Gamma_1^R = \Gamma$.

3.3.4 Clamped plate problem with exact solution

Let $d = 2$ and $\Omega = [0,1] \times [0,1]$. When

$$\begin{aligned} f :=& 8\pi^4 \cos^2(\pi x)(\cos(\pi y)^2 - 2\sin(\pi y)^2) \\ & - 8\pi^4 \sin(\pi y)^2(2\cos(\pi x)^2 - 3\sin(\pi x)^2), \end{aligned}$$

the exact solution of *Vector* BVP 2 is $u := \sin^2(\pi x)\sin^2(\pi y)$. The algorithm using the toolbox to solve this vector BVP is the following:

Algorithm 3.15 Clamped plate problem with exact solution

$\mathcal{T}_h \leftarrow$ HyperCube$(2, N)$
d $\leftarrow 2$, m $\leftarrow 2$
Hop $\leftarrow$ Hopoperator$(2,2)$
Hop.H$(1,2) \leftarrow$ Loperator$(2, \mathbb{I}_2, \mathbf{0}, \mathbf{0}, 0)$
Hop.H$(2,1) \leftarrow$ Loperator$(2, \mathbb{I}_2, \mathbf{0}, \mathbf{0}, 0)$
Hop.H$(2,2) \leftarrow$ Loperator$(2, \mathbb{O}_2, \mathbf{0}, \mathbf{0}, -1)$
pde $\leftarrow$ initPDE$(\text{Hop}, \mathcal{T}_h)$
$f \leftarrow 8\pi^4 \cos^2(\pi x)(\cos(\pi y)^2 - 2\sin(\pi y)^2) - 8\pi^4 \sin(\pi y)^2(2\cos(\pi x)^2 - 3\sin(\pi x)^2)$
pde.f $\leftarrow \boldsymbol{x} \mapsto \begin{pmatrix} f \\ 0 \end{pmatrix}$
for $i \leftarrow 1$ **to** pde.nlab **do**
  pde $\leftarrow$ setBC_PDE(pde, pde.labels(i), 1, 'Dirichlet', 0., $\varnothing$)
end for
$\boldsymbol{W} \leftarrow$ SolvePDE(PDE)

We give in Listings 11 and 12 the corresponding Matlab/Octave and Python codes.

Listing 11: **2D clamped plate with exact solution, Matlab/Octave code**

```
f=@(x,y)8*pi^4*cos(pi*x).^2.*cos(pi*y).^2
    -16*pi^4*cos(pi*y).^2.*sin(pi*x).^2
    -16*pi^4*cos(pi*x).^2.*sin(pi*y).^2
    +24*pi^4*sin(pi*x).^2.*sin(pi*y).^2;
d=2;m=2;
N=input('N=');
Th=HyperCube(d,N);
Hop=Hoperator(d,m);
Hop.H{1,2}=Loperator(d,{1,0;0,1},[],[],[]);
Hop.H{2,1}=Loperator(d,{1,0;0,1},[],[],[]);
Hop.H{2,2}=Loperator(d,[],[],[],-1);
pde=initPDE(Hop,Th);
for i=pde.labels
  pde=setBC_PDE(pde,i,1,'Dirichlet',0);
end
pde.f={f,[]};
W=solvePDE(pde,'Split',true);
```

Listing 12: **2D clamped plate with exact solution, Python code**

```
from numpy import sin,cos
u=lambda x,y:
    sin(pi*x)**2*sin(pi*y)**2
f=lambda
    x,y:(8*pi**4)*cos(pi*x)**2*cos(pi*y)**2
    -(16*pi**4)*cos(pi*y)**2*sin(pi*x)**2
    -(16*pi**4)*cos(pi*x)**2*sin(pi*y)**2
    +(24*pi**4)*sin(pi*x)**2*sin(pi*y)**2
Th=HyperCube(2,100)
d=2;m=2
Hop=Hoperator(d=d,m=m)
Hop.H[0][1]=Loperator(d=d,A=[[1,None],[None,1]])
Hop.H[1][0]=Loperator(d=d,A=[[1,None],[None,1]])
Hop.H[1][1]=Loperator(d=d,a0=-1.0)
pde=initPDE(Hop,Th)
pde.f=[f,0]
for i in pde.labels:
  pde=setBC_PDE(pde,i,0,'Dirichlet',0)
W=solvePDE(pde,split=True)
```

We give in Figure 17, the relative error between the numerical solution and $\pi_h(u)$ in L_2 and H^1 norms.

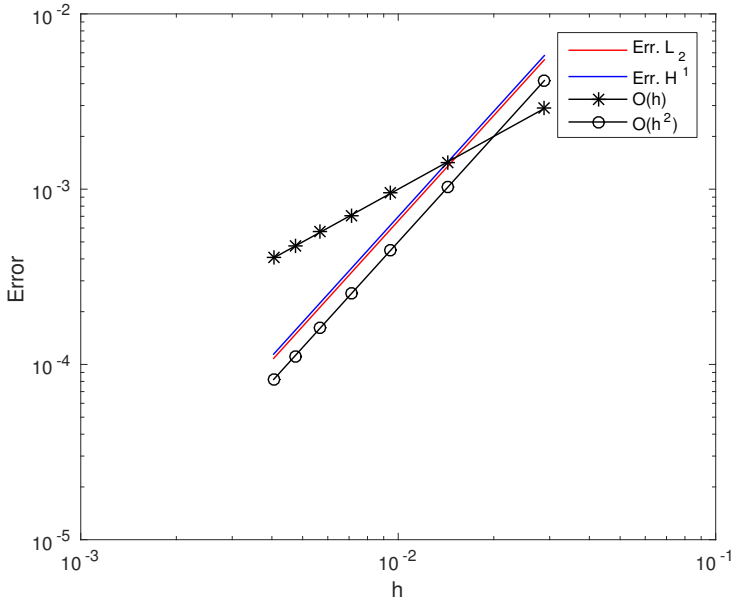


Figure 17: Relative error for clamped plate with exact solution

3.3.5 Clamped plate problem : sample 1

In this part, we take examples of the thesis of T. Gerasimov [6] (page 138). Let $d = 2$, $\Omega = [-1, 6] \times [-1, 1] \subset \mathbb{R}^d$ and

$$f := \exp(-100((x + 0.75)^2 + (y - 0.75)^2)).$$

The domain Ω could be generated with the HyperCube function :

$$\mathcal{T}_h \leftarrow \text{HyperCube}(2, [70, 20], \boldsymbol{x} \mapsto (-1 + 7\boldsymbol{x}_1, -1 + 2\boldsymbol{x}_2)).$$

In Figure 18 we represent Ω and its boundary $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \cup \Gamma_4$.

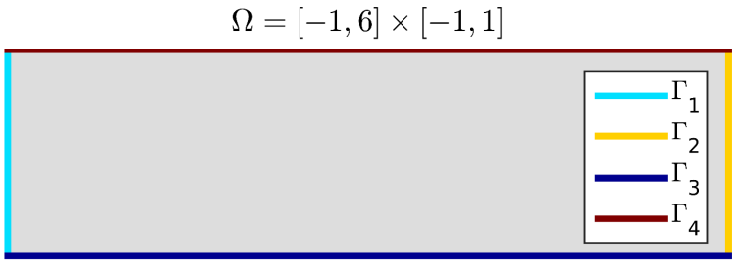


Figure 18: Mesh from HyperCube$(2, [70, 20], \boldsymbol{x} \mapsto (-1 + 7\boldsymbol{x}_1, -1 + 2\boldsymbol{x}_2))$

The algorithm using the toolbox to solve this vector BVP is the following:

Algorithm 3.16 Clamped plate problem

$\mathcal{T}_h \leftarrow$ HyperCube$(2, [70, 20], \boldsymbol{x} \mapsto (-1 + 7\boldsymbol{x}_1, -1 + 2\boldsymbol{x}_2))$
$\mathtt{d} \leftarrow 2,\ \mathtt{m} \leftarrow 2$
$\mathtt{Hop} \leftarrow$ Hopoperator$(2, 2)$
$\mathtt{Hop.H}(1, 2) \leftarrow$ Loperator$(2, \mathbb{I}_2, \mathbf{0}, \mathbf{0}, 0)$
$\mathtt{Hop.H}(2, 1) \leftarrow$ Loperator$(2, \mathbb{I}_2, \mathbf{0}, \mathbf{0}, 0)$
$\mathtt{Hop.H}(2, 2) \leftarrow$ Loperator$(2, \mathbb{O}_2, \mathbf{0}, \mathbf{0}, -1)$
pde $\leftarrow$ initPDE$(\mathtt{Hop}, \mathcal{T}_h)$
pde.f $\leftarrow \boldsymbol{x} \mapsto \begin{pmatrix} \exp(-100((\boldsymbol{x}_1 + 0.75)^2 + (\boldsymbol{x}_2 - 0.75)^2)) \\ 0 \end{pmatrix}$
for $i \leftarrow 1$ **to** pde.nlab **do**
  pde $\leftarrow$ setBC_PDE(pde, pde.labels(i), 1, 'Dirichlet', 0., $\varnothing$)
end for
$\boldsymbol{X} \leftarrow$ SolvePDE(PDE)

We give in Listings 13 and 14 the corresponding Matlab/Octave and Python codes.

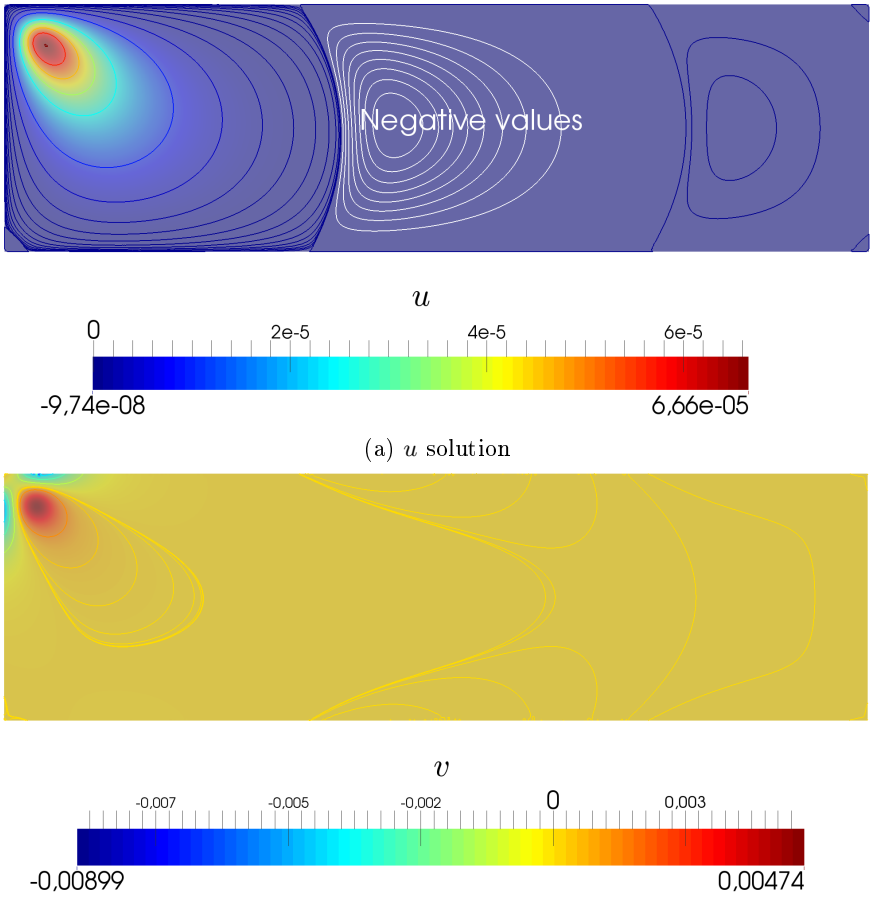
Listing 13: **2D clamped plate, Matlab/Octave code**

```
d=2;m=2;
fprintf('1. Reading of the
    mesh\n');
Th=HyperCube(d,50*[7,2],
    @(q)[7*q(1,:)-1;2*q(2,:)-1]);
fprintf('2. Definition of
    the BVP\n')
Hop=Hoperator(d,m);
Hop.H{1,2}=Loperator(d,{1,0;0,1},[],[],[]);
Hop.H{2,1}=Loperator(d,{1,0;0,1},[],[],[]);
Hop.H{2,2}=Loperator(d,[],[],[],-1);
pde=initPDE(Hop,Th);
pde.f={@(x,y)exp(-100*((x+0.75).^2
    +(y-0.75).^2)),0};
for i=1:pde.nlab
  pde=setBC_PDE(pde,pde.labels(i),1,
    'Dirichlet',0);
end
fprintf('3. Solving BVP\n')
W=solvePDE(pde,'split',true);
```

Listing 14: **2D clamped plate, Python code**

```
d=2;m=2;
print('1. Reading of the
    mesh');
Th=HyperCube(d,[20*7,20*2],trans=lambda
    q:np.c_[7*q[:,0]-1,2*q[:,1]-1])
print('2. Definition of the
    BVP')
Hop=Hoperator(d=2,m=2)
Hop.H[0][1]=Loperator(d=2,
    A=[[1,None],[None,1]])
Hop.H[1][0]=Loperator(d=2,
    A=[[1,None],[None,1]])
Hop.H[1][1]=Loperator(d=2,a0=-1)
pde=initPDE(Hop,Th)
pde.f=[lambda
    x,y:exp(-100*((x+0.75)**2
    +(y-0.75)**2)),0]
for l in pde.labels:
  pde=setBC_PDE(pde,l,0,'Dirichlet',0,None)
print('3. Solving BVP')
x=solvePDE(pde,split=True)
```

The numerical solution for a given mesh is shown on Figure 19



(a) u solution

(b) v solution

Figure 19: Clamped plate problem

4 Data structures

In this section several data structures are defined. Each structure makes the algorithms shorter and easier to read. First the Mesh structure allows to describe the mesh of a domain $\Omega \subset \mathbb{R}^d$ made of d-simplices . Then the boundary mesh data structure is presented which allows to store and identify distinct parts of the boundary mesh. After that we define the structures associated with the boundary conditions and with the operators $\mathcal{L}$ and $\mathcal{H}$. Finally we explain the PDE structure which allows to completely describe the BVP.

4.1 Structure for meshes

We suppose that Ω is equipped with a mesh $\mathcal{T}_h$ (locally conforming) where its elements are d-simplices . We denote by Ω_h the union of the elements belonging to the mesh, $\Omega_h = \bigcup_{K \in \mathcal{T}_h} K$, and by Γ_h its boundary, $\Gamma_h = \partial\Omega_h$.

The following data structure is associated to the mesh $\mathcal{T}_h$ and employs many notations already used in FreeFEM++ (see [7, 8]).

Mesh structure associated to $\mathcal{T}_h$

d	:	integer space dimension
n_q	:	integer number of vertices
n_{me}	:	integer number of elements (d-simplices)
n_{be}	:	integer number of boundary elements ((d−1)-simplices)
q	:	d-by-n_q array of reals array of vertex coordinates
me	:	(d+1)-by-n_{me} array of integers connectivity array for **mesh elements**
mel	:	1-by-n_{me} array of integers array of mesh element labels
be	:	d-by-n_{be} array of integers connectivity array for **boundary elements**
bel	:	1-by-n_{be} array of integers array of boundary elements labels
vols	:	1-by-n_{me} array of reals array of mesh elements volumes
h	:	double mesh step size (=maximum edge length in the mesh)
info	:	two field structure containing the name and the format of the mesh file

More precisely

- $q(\nu, j)$ is the ν-th coordinate of the j-th vertex, $\nu \in \{1, \ldots, d\}$, $j \in \{1, \ldots, n_q\}$. The j-th vertex will be also denoted by $q^j = q(:, j)$.

- me(β, k) is the storage index of the β-th vertex of the k-th element (d-simplex), in the array q, for $\beta \in \{1, ..., \mathrm{d}+1\}$ and $k \in \{1, \ldots, \mathrm{n_{me}}\}$. So q(:, me$(\beta, k)$) represents the coordinates of the β-th vertex of the k-th mesh element.

- be(β, l) is the storage index of the β-th vertex of the l-th boundary element ((d$-$1)-simplex), in the array q, for $\beta \in \{1, ..., \mathrm{d}\}$ and $l \in \{1, \ldots, \mathrm{n_{be}}\}$. So q(:, be$(\beta, l)$) represents the coordinates of the β-th vertex of the l-th boundary element.

- vols(k) is the volume of the k-th d-simplex .

- info.name is a string of the short name of the mesh file.
  info.format is the format of the mesh file: it can be 'freefem', 'medit' or 'gmsh'.

See in Section 11.1 an example of the mesh data structure for a ring.

4.2 Structure for boundary meshes

In this section we define a boundary mesh data structure which allows to easily differentiate parts of the boundary mesh. This will be very useful to clearly identify the Dirichlet and Robin boundaries in scalar and vector BVP's and also to simplify the contributions of boundary conditions.

Let Σ_h be a non-empty part of $\Gamma_h = \partial\Omega_h$ extracted from $\mathcal{T}_h$. The boundary mesh Σ_h is defined by its $\mathrm{n_q}$ *vertices* $\mathrm{q}^r \in \mathbb{R}^d$, $r \in [\![1, \mathrm{n_q}]\!]$. Its $\mathrm{n_{me}}$ *faces* or boundary elements are (d−1)-simplices extracted from mesh elements of $\mathcal{T}_h$. They are given by the connectivity array me such that $\mathrm{q}^{\mathrm{me}(\alpha,k)}$ is the α-th vertex of the k-th boundary element. We denote by $\mathcal{I}_{\Sigma_h}$ the ordered subset of $[\![1, \mathcal{T}_h.\mathrm{n_q}]\!]$ such that $\#\mathcal{I}_{\Sigma_h} = \Sigma_h.\mathrm{n_q}$ and

$$\forall r \in [\![1, \mathrm{n_q}]\!] \quad \Sigma_h.\mathrm{q}^r \equiv \mathcal{T}_h.\mathrm{q}^i_{|\Sigma_h} \text{ where } i = \mathcal{I}_{\Sigma_h}(r). \tag{4.1}$$

The boundary mesh data structure associated to Σ_h and extracted from $\mathcal{T}_h$ is given by

Boundary mesh structure associated to $\Sigma_h \subset \Gamma_h$

`d`	:	integer dimension of d-simplices : $\mathrm{d} = \mathcal{T}_h.d-1$
$\mathrm{n_q}$	:	integer number of vertices
$\mathrm{n_{me}}$	:	integer number of boundary elements (d-simplices)
q	:	(d+1)-by-$\mathrm{n_q}$ array of reals array of vertex coordinates
me	:	(d+1)-by-$\mathrm{n_{me}}$ array of integers connectivity array for boundary mesh elements
vols	:	1-by-$\mathrm{n_{me}}$ array of reals array of d-simplices volumes
`toGlobal`	:	1-by-$\mathrm{n_q}$ array of integers $\mathcal{T}_h.\mathrm{q}(:, \texttt{toGlobal}) \equiv \mathrm{q}$
`nqGlobal`	:	integer ($\mathcal{T}_h.\mathrm{n_q}$) `nqGlobal`$\equiv \mathcal{T}_h.\mathrm{n_q}$
`label`	:	value of $labels(l)$

More precisely

- $\mathrm{q}(\nu, j)$ is the ν-th coordinate of the j-th vertex, $\nu \in \{1, \ldots, \mathrm{d}+1\}$, $j \in \{1, \ldots, \mathrm{n_q}\}$. The j-th vertex will be also denoted by q^j, $\mathrm{q}^j \in \mathbb{R}^{\mathcal{T}_h.d}$.
- $\mathrm{me}(\beta, k)$ is the storage index of the β-th vertex of the k-th element (d-simplex), in the array q, for $\beta \in \{1, ..., \mathrm{d}+1\}$ and $k \in \{1, \ldots, \mathrm{n_{me}}\}$.
- $\mathrm{vol}(k)$ is the k-th d-simplex volume.

Now we explain how this structure can be initialized from a mesh structure $\mathcal{T}_h$.

Let *labels* be the ordered set of unique boundary labels of $\mathcal{T}_h$.bel, $\mathrm{n_{lab}}$ be its cardinality and, for any $i \in$ *labels*, Γ_h^i be the union of boundary elements of label i. So we have

$$\Gamma_h = \bigcup_{l=1}^{\mathrm{n_{lab}}} \Gamma_h^{labels(l)} = \bigcup_{i \in labels} \Gamma_h^i.$$

The Algorithm 4.1 gives the function BuildBoundaryMesh which creates the boundary mesh structure associated to $\Sigma_h = \Gamma_h^{\mathtt{Label}}$, where $\mathtt{Label} \in$ *labels*.

Algorithm 4.1 function BuildBoundaryMesh

Input :

$\mathcal{T}_h$: a mesh structure of Ω

`Label` : an integer as a label of a mesh boundary

Output :

Σ_h : the boundary mesh structure of $\Gamma_h^{\mathtt{Label}}$.

Function $\Sigma_h \leftarrow$ BuildBoundaryMesh($\mathcal{T}_h$, `Label`)
  Σ_h.d $\leftarrow \mathcal{T}_h$.d $- 1$
  Σ_h.label $\leftarrow$ `Label`
  `I` $\leftarrow$ Find($\mathcal{T}_h$.bel $\equiv$ `Label`)
  `BE` $\leftarrow \mathcal{T}_h$.be(:, `I`)
  `indQ` $\leftarrow$ Unique(`BE`)
  Σ_h.q $\leftarrow \mathcal{T}_h$.q(:, `indQ`)
  Σ_h.nq $\leftarrow$ Length(`indQ`)
  `J` $\leftarrow \mathbf{0}_{\mathcal{T}_h.\mathrm{nq}}$; `J`(`indQ`) $\leftarrow 1 :$ length(`indQ`)
  Σ_h.me $\leftarrow$ `J`(`BE`)
  Σ_h.nme $\leftarrow$ Size(Σ_h.me, 2)
  Σ_h.`toGlobal` $\leftarrow$ `indQ`
  Σ_h.`nqGlobal` $\leftarrow \mathcal{T}_h$.nq
  Σ_h.vols $\leftarrow$ ComputeVolVec(Σ_h.d, Σ_h.q, Σ_h.me)
end Function

The Algorithm 11.1 defines the function BuildBoundaryMeshes which creates a 1-by-$\mathrm{n_{lab}}$ array $\mathcal{B}_h$ of boundary mesh structures. For any $l \in [\![1, \mathrm{n_{lab}}]\!]$, $\mathcal{B}_h(l)$ is the boundary mesh structure of $\Gamma_h^{labels(l)}$.

Algorithm 4.2 function BuildBoundaryMeshes

Input :

$\mathcal{T}_h$: a mesh structure of Ω

Output :

$\mathcal{B}_h$: 1-by-$\mathrm{n_{lab}}$ array of boundary mesh structures.

Function $\mathcal{B}_h \leftarrow$ BuildBoundaryMeshes($\mathcal{T}_h$)
  `labels` $\leftarrow$ Unique($\mathcal{T}_h$.bel)
  `nlab` $\leftarrow$ Length(`labels`)
  for $l \leftarrow 1$ **to** `nlab` **do**
    $\mathcal{B}_h(l) \leftarrow$ BuildBoundaryMesh($\mathcal{T}_h$, `labels`(l))
  end for
end Function

Let l in $[\![1, \mathrm{n_{lab}}]\!]$. To simplify the notation (4.1) for $\Sigma_h = \Gamma_h^{labels(l)}$, we set $\mathcal{I}_l = \mathcal{I}_{\Gamma_h^{labels(l)}}$. So $\mathcal{I}_l$ is given by $\mathcal{B}_h(l)$.toGlobal $\subset [\![1, \mathcal{T}_h.\mathrm{n_q}]\!]$. We also denote by $\mathrm{n_{q,}}_l$ the cardinality of $\mathcal{I}_l$ and thus $\mathrm{n_{q,}}_l = \mathcal{B}_h(l).\mathrm{n_q}$. Finally (4.1) becomes

$$\forall r \in [\![1, \mathrm{n_{q,}}_l]\!], \quad j = \mathcal{I}_l(r) \ \text{ with } \mathcal{B}_h(l).\mathrm{q}^r \equiv \mathcal{T}_h.\mathrm{q}^j. \tag{4.2}$$

See in Section 11.2 an example of the boundary mesh data structure.

4.3 Structure for boundary conditions

In the *vector* case, for all $\alpha \in [\![1, m]\!]$, the discrete Dirichlet and Robin boundaries $\Gamma_{h,\alpha}^D$ and $\Gamma_{h,\alpha}^R$ are parts of Γ_h and respectively denoted by $\Gamma_{h,\alpha}^D$ and $\Gamma_{h,\alpha}^R$. Let $\mathcal{I}_{labels}^{D,\alpha}$ and $\mathcal{I}_{labels}^{R,\alpha}$ be the subsets of $[\![1, \mathrm{n_{lab}}]\!]$ (possibly empty) such that

$$\Gamma_{h,\alpha}^D = \bigcup_{l \in \mathcal{I}_{labels}^{D,\alpha}} \Gamma_h^{labels(l)}, \quad \Gamma_{h,\alpha}^R = \bigcup_{l \in \mathcal{I}_{labels}^{R,\alpha}} \Gamma_h^{labels(l)}. \tag{4.3}$$

For the vector BVP problem, the Dirichlet boundary conditions (2.11) can be written on Γ_h as

$$\boldsymbol{u}_\alpha = g_\alpha^D \ \text{ on } \Gamma_{h,\alpha}^D, \ \forall \alpha \in [\![1, m]\!].$$

Using notations (4.3), we can also write equivalently

$$\boldsymbol{u}_\alpha = g_\alpha^D \ \text{ on } \Gamma_h^{labels(l)}, \ \forall l \in \mathcal{I}_{labels}^{D,\alpha}, \ \forall \alpha \in [\![1, m]\!]. \tag{4.4}$$

To store all these Dirichlet boundary conditions, we choose to represent them as a m-by-$\mathrm{n_{lab}}$ array, named bclD, of one-field structures g such that

$$\mathrm{bclD}(\alpha, l).\mathrm{g} \leftarrow \begin{cases} g_\alpha^D{}_{|\Gamma_h^{labels(l)}}, & \text{if } l \in \mathcal{I}_{labels}^{D,\alpha} \\ \varnothing & \text{otherwise} \end{cases}$$

In the same way, the Robin boundary conditions (2.12) of the vector BVP problem can be written on Γ_h as

$$\frac{\partial \boldsymbol{u}}{\partial n_{\mathcal{H}_\alpha}} + a_\alpha^R \boldsymbol{u}_\alpha = g_\alpha^R \ \text{ on } \Gamma_h^{labels(l)}, \ \forall l \in \mathcal{I}_{labels}^{R,\alpha}, \ \forall \alpha \in [\![1, m]\!] \tag{4.5}$$

and we choose to represent them by an m-by-$\mathrm{n_{lab}}$ array, named bclR, of two-field structures with fields g and ar such that

$$\mathrm{bclR}(\alpha, l).\mathrm{g} \leftarrow \begin{cases} g_\alpha^R{}_{|\Gamma_h^{labels(l)}}, & \text{if } l \in \mathcal{I}_{labels}^{R,\alpha} \\ \varnothing & \text{otherwise} \end{cases}$$

and

$$\mathrm{bclR}(\alpha, l).\mathrm{ar} \leftarrow \begin{cases} a_\alpha^R{}_{|\Gamma_h^{labels(l)}}, & \text{if } l \in \mathcal{I}_{labels}^{R,\alpha} \\ \varnothing & \text{otherwise} \end{cases}$$

In the *scalar* case, the Dirichlet and Robin boundaries Γ^D and Γ^R are parts of Γ_h and respectively denoted by Γ_h^D and Γ_h^R. Let $\mathcal{I}_{labels}^D$ and $\mathcal{I}_{labels}^R$ be the subsets of $[\![1, \mathrm{n_{lab}}]\!]$ such that

$$\Gamma_h^D = \bigcup_{l \in \mathcal{I}_{labels}^D} \Gamma_h^{labels(l)}, \quad \Gamma_h^R = \bigcup_{l \in \mathcal{I}_{labels}^R} \Gamma_h^{labels(l)}. \tag{4.6}$$

To describe the boundary conditions, one can use the previous arrays bclD and bclR with $m = 1$.

4.4 Structure for operators

4.4.1 *Scalar* operators

$\mathcal{L} = \mathcal{L}_{\mathbb{A},\boldsymbol{b},\boldsymbol{c},a_0}$ given in (2.1) is defined by

`Loperator` data structure

`d`	:	dimension of the operator
`A`	:	d-by-d array of functions $\Omega \to \mathbb{R}$
`b`	:	1-by-d array of functions $\Omega \to \mathbb{R}$
`c`	:	1-by-d array of functions $\Omega \to \mathbb{R}$
`a0`	:	function $\Omega \to \mathbb{R}$
`order`	:	integer order of the operator ($0 \leqslant$ `order` $\leqslant 2$)

Using the function Loperator defined in Algorithm 4.3, we can easily create a `Loperator` structure associated to a $\mathcal{L}$-operator. For example

- the *Mass* operator in dimension d is defined by $\mathcal{L}_{\mathbb{O}_{d\times d},\mathbf{0}_d,\mathbf{0}_d,1}$, so

$$\texttt{LMass} \leftarrow \textsc{Loperator}(d, \mathbb{O}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, 1).$$

- the *Stiffness* operator in dimension d is defined by $\mathcal{L}_{\mathbb{I}_{d\times d},\mathbf{0}_d,\mathbf{0}_d,0}$, so

$$\texttt{LStiff} \leftarrow \textsc{Loperator}(d, \mathbb{I}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, 0).$$

Algorithm 4.3 function Loperator

Input :

d	:	d space dimension of $\Omega \subset \mathbb{R}^d$,
$\mathbb{A}$	:	d-by-d matrix of functions $\Omega \to \mathbb{R}$,
$\boldsymbol{b}$	:	1-by-d vector of functions $\Omega \to \mathbb{R}$,
$\boldsymbol{c}$	:	1-by-d vector of functions $\Omega \to \mathbb{R}$,
a_0	:	function $\Omega \to \mathbb{R}$

Output :

L	:	`Loperator` structure associated with the $\mathcal{L}$-operator $\mathcal{L}_{\mathbb{A},\boldsymbol{b},\boldsymbol{c},a_0}$.

Function L $\leftarrow$ Loperator$(d, \mathbb{A}, \boldsymbol{b}, \boldsymbol{c}, a_0)$
  L.d $\leftarrow d$
  L.A $\leftarrow \mathbb{A}$
  L.b $\leftarrow \boldsymbol{b}$
  L.c $\leftarrow \boldsymbol{c}$
  L.a0 $\leftarrow a_0$
  if $\mathbb{A} \not\equiv \mathbb{O}_{d\times d}$ **then**
    L.order $\leftarrow 2$
  else
    if $\boldsymbol{b} \not\equiv \boldsymbol{O}_d$ or $\boldsymbol{c} \not\equiv \boldsymbol{O}_d$ **then**
      L.order $\leftarrow 1$
    else
      L.order $\leftarrow 0$
    end if
  end if
end Function

The discretisation of this operator by a $\mathbb{P}^1$-Lagrange finite element method on Ω_h is given by

Ddata or Ldata structure

`d`	:	dimension
`A`	:	d-by-d arrays of `Fdata`
`b`	:	1-by-d arrays of `Fdata`
`c`	:	1-by-d arrays of `Fdata`
`a0`	:	`Fdata`

where `Fdata` denotes the $\mathbb{P}^1$-Lagrange finite element approximation of a function $f : \Omega \longrightarrow \mathbb{R}$ on Ω_h such that $\texttt{Fdata}(i) = f(\mathrm{q}^i)$, $\forall i \in [\![1, \mathrm{n_q}]\!]$. To initialize a `Ddata` structure from a mesh we can use Algorithm 4.4.

Algorithm 4.4 function setDdata

Input :

D : Doperator structure.
$\mathcal{M}_h$: $\mathcal{T}_h$ mesh structure (see Section 4) or $\mathcal{B}_h$ boundary mesh structure

Output :

$\mathrm{D_h}$: Ddata structure.

Function $\mathrm{D_h} \leftarrow$ setDdata$(\mathrm{D}, \mathcal{M}_h)$
  $\mathrm{D_h}.\mathrm{d} \leftarrow \mathrm{D}.\mathrm{d}$
  for $i \leftarrow 1$ **to** D.d **do**
    for $j \leftarrow 1$ **to** D.d **do**
      $\mathrm{D_h}.\mathrm{A}(i,j) \leftarrow$ setFdata$(\mathrm{D}.\mathrm{A}(i,j), \mathcal{M}_h)$
    end for
    $\mathrm{D_h}.\mathrm{b}(i) \leftarrow$ setFdata$(\mathrm{D}.\mathrm{b}(i), \mathcal{M}_h)$
    $\mathrm{D_h}.\mathrm{c}(i) \leftarrow$ setFdata$(\mathrm{D}.\mathrm{c}(i), \mathcal{M}_h)$
  end for
  $\mathrm{D_h}.\mathrm{a0} \leftarrow$ setFdata$(\mathrm{D}.\mathrm{a0}, \mathcal{M}_h)$
end Function

Algorithm 4.5 function setFdata

Input :

f : function from Ω or Γ to $\mathbb{R}$.
$\mathcal{M}_h$: $\mathcal{T}_h$ mesh structure (see Section 4) or $\mathcal{B}_h$ boundary mesh structure

Output :

$\boldsymbol{f}_h$: $\mathbb{R}^{\mathrm{n_q}}$ vector such that $\boldsymbol{f}_h(i) = f(\mathcal{M}_h.q^i)$.

Function $\boldsymbol{f}_h \leftarrow$ setFdata$(f, \mathcal{M}_h)$
  for $i \leftarrow 1$ **to** $\mathcal{M}_h.\mathrm{n_q}$ **do**
    $\boldsymbol{f}_h(i) \leftarrow f(\mathcal{M}_h.q^i)$
  end for
end Function

We can easily obtain the local data associated to the k-th mesh element (see Algorithm 4.6)

Algorithm 4.6 function GETLOCFDATA

Input :

f_h : Fdata : 1-by-n_q array of doubles.
$\mathcal{T}_h$: mesh structure associated to Ω_h (see section 4.1)
k : index of a mesh element

Output :

fL : Local Fdata. 1-by-$(d+1)$ array of doubles

1: **Function** $fL \leftarrow$ GETLOCFDATA$(f_h, \mathcal{T}_h, k)$
2: $fL \leftarrow f_h(\mathcal{T}_h.\text{me}(:,k))$
3: **end Function**

4.4.2 *Vector* operators

The operator $\mathcal{H}$ used in (2.6) is defined by the `Hoperator` structure

Hoperator structure

`d`	:	space dimension
`m`	:	operator dimension (m in definition (2.6))
`H`	:	`m`-by-`m` array of `Loperator`

The initialization of a `Hoperator` structure to the zero $\mathcal{H}$ operator is given in Algorithm 4.7.

Algorithm 4.7 function HOPOPERATOR

Input :

d : d space dimension of $\Omega \subset \mathbb{R}^d$,
m : operator dimension

Output :

Hop : `Hopoperator` structure corresponding to
: the zero $\mathcal{H}$ operator

1: **Function** Hop $\leftarrow$ HOPOPERATOR(d, m)
2: Hop.d $\leftarrow d$
3: Hop.m $\leftarrow m$
4: **for** $i \leftarrow 1$ **to** m **do**
5: **for** $j \leftarrow 1$ **to** m **do**
6: Hop.H$(i,j) \leftarrow \varnothing$
7: **end for**
8: **end for**
9: **end Function**

For example, the operator $\mathcal{H}$ given by

$$\mathcal{H} = \begin{pmatrix} \mathcal{L}_{\mathbb{I}_{d\times d},\mathbf{0}_d,\mathbf{0}_d,1} & \mathcal{L}_{\mathbb{O}_{d\times d},\mathbf{0}_d,\mathbf{0}_d,-1} \\ 0 & \mathcal{L}_{\mathbb{I}_{d\times d},\mathbf{0}_d,\mathbf{0}_d,0} \end{pmatrix}$$

can be initialized with

1: `Hop` $\leftarrow$ HOPOPERATOR$(d, 2)$
2: Hop.H$(1,1) \leftarrow$ LOPERATOR$(d, \mathbb{I}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, 1)$
3: Hop.H$(1,2) \leftarrow$ LOPERATOR$(d, \mathbb{O}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, -1)$
4: Hop.H$(2,2) \leftarrow$ LOPERATOR$(d, \mathbb{I}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, 0)$

4.5 Structure for PDE's

The PDE structure which contains all the data necessary to solve a scalar or vector BVP is now described here

PDE structure

d	:	integer space dimension
m	:	integer operator dimension (1 for $\mathcal{L}$ operator)
op	:	operator $\mathcal{L}$ operator or $\mathcal{H}$ operator
f	:	1-by-m array of functions
$\mathcal{T}_h$	:	mesh structure
$\mathcal{B}_h$	:	1-by-nlab array of boundary mesh structures
labels	:	1-by-nlab array of integers Boundary mesh labels
nlab	:	integer number of boundary labels
bclR	:	m-by-nlab array of `BCrobin` structures
bclD	:	m-by-nlab array of `BCdirichlet` structures

The Algorithm 4.8 defines the function initPDE which initializes a PDE structure pde from an operator and a mesh. In this version default boundary conditions when they exist are homogeneous Neumann. The function setBC_PDE in the Algorithm 4.9 allows to modify boundary conditions and to choose Dirichlet, Neumann or Robin ones.

For the moment we use the function SolvePDE as a black box. This function allows to solve by $\mathbb{P}^1$-Lagrange finite element method the partial differential equation contained in a PDE structure on a given mesh structure. The goal of Sections 6 and 7 is to introduce a generic version of this function.

Algorithm 4.8 function initPDE

Input :

Op : operator structure (see Section 4) such that Op.d = $\mathcal{T}_h$.d

$\mathcal{T}_h$: mesh structure associated to Ω_h (see Section 4)

Output :

pde : a PDE structure. All boundary conditions are set by default to homogeneous Neumann boundary conditions.
Fields are *d*, *m*, *op*, *f*, *Th*, *Bh*, *labels*, *nlab*, *bclD*, *bclR*.

Function pde ← initPDE(Op, $\mathcal{T}_h$)
  pde.d ← $\mathcal{T}_h$.d; pde.m ← Op.m
  pde.op ← Op
  pde.f ← $\mathbf{0}_{\text{pde.m}}$
  pde.$\mathcal{T}_h$ ← $\mathcal{T}_h$
  pde.$\mathcal{B}_h$ ← BuildBoundaryMeshes($\mathcal{T}_h$)
  pde.labels ← Unique($\mathcal{T}_h$.bel)
  pde.nlab ← Length(pde.labels)
  for $i \leftarrow 1$ **to** pde.m **do**
    for $l \leftarrow 1$ **to** pde.nlab **do**
      pde.bclR(i,l).g ← $\varnothing$
      pde.bclR(i,l).ar ← $\varnothing$
      pde.bclD(i,l).g ← $\varnothing$
    end for
  end for
end Function

Algorithm 4.9 function setBC_PDE

Input :

pde : a PDE structure

label : label of the boundary (in pde.labels)

α : index

type : string for the name of the boundary condition. It can be "Dirichlet", "Neumann" or "Robin"

g : function $\Gamma \to \mathbb{R}$ corresponding to the source term in Dirichlet, Neumann or Robin boundary conditions

ar : function $\Gamma \to \mathbb{R}$ corresponding to the weight function in the Robin boundary condition

Output :

pde : the modified PDE structure

Function pde ← setBC_PDE(pde, label, α, type, g, ar)
  l ← Find(pde.labels ≡ label)
  if type is "Dirichlet" **then**
    pde.bclD(α,l).g ← g
  end if
  if type is "Robin" **then**
    pde.bclR(α,l).g ← g, pde.bclR(α,l).ar ← ar,
  end if
  if type is "Neumann" **then**
    pde.bclR(α,l).g ← g, pde.bclR(α,l).ar ← $\varnothing$,
  end if
end Function

5 Finite Element Approximation

We first recall some notations and results on Sobolev spaces. We define here some finite dimensional spaces which will be used in the sequel.

5.1 Sobolev spaces

The Sobolev space $\mathrm{H}^1(\Omega)$ is set by

$$\mathrm{H}^1(\Omega) := \left\{ v \in L^2(\Omega) \mid \frac{\partial v}{\partial x_i} \in L^2(\Omega),\ \forall i \in [\![1, d]\!] \right\} \tag{5.1}$$

This space is a Banach space with respect to the norm

$$\|v\|_{1,\Omega} = \left(\int_\Omega |v|^2 d\mathrm{q} + \sum_{i=1}^{d} \int_\Omega |\frac{\partial v}{\partial x_i}|^2 d\mathrm{q} \right)^{1/2} \tag{5.2}$$

Theorem 3. *Suppose that Ω has a Lipschitz boundary. Then there exists a constant C such that*

$$\|v\|^2_{L^2(\partial\Omega)} \leqslant C \|v\|_{L^2(\Omega)} \|v\|_{\mathrm{H}^1(\Omega)}, \quad \forall v \in \mathrm{H}^1(\Omega) \tag{5.3}$$

We will use the notation $\mathrm{H}^1_0(\Omega)$ to denote the subset of $\mathrm{H}^1(\Omega)$, consisting of functions whose trace on $\partial\Omega$ is zero, that is

$$\mathrm{H}^1_0(\Omega) = \left\{ v \in \mathrm{H}^1(\Omega) \mid v_{|\partial\Omega} = 0 \text{ in } L^2(\partial\Omega) \right\} \tag{5.4}$$

Let Σ be a Lipschitz continuous subset of the boundary $\partial\Omega$. The space $\mathrm{H}^{1/2}(\Sigma)$ is characterized by

$$\mathrm{H}^{1/2}(\Sigma) = \left\{ v_{|\Sigma} \mid v \in \mathrm{H}^1(\Omega) \right\}. \tag{5.5}$$

Let $g \in \mathrm{H}^{1/2}(\Sigma)$, we define the space $\mathrm{H}^1_{g,\Sigma}(\Omega)$, which is not a vector space, to characterize functions which satisfy a Dirichlet boundary condition on Σ as follows

$$\mathrm{H}^1_{g,\Sigma}(\Omega) = \left\{ v \in \mathrm{H}^1(\Omega) \mid v_{|\Sigma} = g \right\}. \tag{5.6}$$

5.2 $\mathbb{P}_1$-Lagrange finite elements on meshes

Let $\mathcal{T}_h$ be an unstructured simplicial mesh of Ω defined in 4.1 and $\Omega_h = \bigcup_{K \in \mathcal{T}_h} K$. Let $\mathbb{P}_k$, $k \geqslant 0$, be the space of polynomials of degree less than or equal to k in the variables $x_1, \ldots, x_d$. We set

$$\mathrm{H}^{1,h}(\Omega_h) = \{ v \in \mathcal{C}^0(\overline{\Omega_h}),\ \ v_{|K} \in \mathbb{P}_1,\ \forall K \in \mathcal{T}_h \} \tag{5.7}$$

which will be called the space of $\mathbb{P}_1$-Lagrange finite elements.

We set $\{\mathcal{T}_h.\varphi_i\}_{i \in [\![1, \mathcal{T}_h.\mathrm{n_q}]\!]}$ as the $\mathbb{P}_1$-Lagrange basis functions on $\mathcal{T}_h$ and when there is no ambiguity we set $\varphi_i \equiv \mathcal{T}_h.\varphi_i$, $\mathrm{n_q} \equiv \mathcal{T}_h.\mathrm{n_q}$, ... Then, for all $u_h \in \mathrm{H}^{1,h}(\Omega_h)$, we have

$$u_h = \sum_{i=1}^{\mathrm{n_q}} u_h(\mathrm{q}^i) \varphi_i \tag{5.8}$$

For all $\boldsymbol{u}_h = (\boldsymbol{u}_{h,1}, \ldots, \boldsymbol{u}_{h,m})^t \in (\mathrm{H}^{1,h}(\Omega_h))^m$ we have

$$\boldsymbol{u}_h = \begin{pmatrix} \boldsymbol{u}_{h,1} \\ \vdots \\ \boldsymbol{u}_{h,m} \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{\mathrm{n_q}} \boldsymbol{u}_{1,i}\varphi_i \\ \vdots \\ \sum_{i=1}^{\mathrm{n_q}} \boldsymbol{u}_{m,i}\varphi_i \end{pmatrix} = \sum_{\alpha=1}^{m} \left(\sum_{i=1}^{\mathrm{n_q}} \boldsymbol{u}_{\alpha,i}\varphi_i \right) \boldsymbol{e}_\alpha = \sum_{\alpha=1}^{m} \boldsymbol{u}_{h,\alpha}\boldsymbol{e}_\alpha \tag{5.9}$$

where, $\forall \alpha \in [\![1, m]\!]$, $\boldsymbol{u}_\alpha \in \mathbb{R}^{\mathrm{n_q}}$ are defined $\forall i \in [\![1, \mathrm{n_q}]\!]$ by $\boldsymbol{u}_{\alpha,i} = \boldsymbol{u}_{h,\alpha}(\mathcal{T}_h.\mathrm{q}^i)$ and $\boldsymbol{e}_\alpha$ is the $\alpha-$th vector of the canonical basis of $\mathbb{R}^m$. We also obtain

$$\mathrm{Span}\left(\mathrm{H}^{1,h}(\Omega_h)^m\right) = \{\varphi_i \boldsymbol{e}_\alpha \mid i \in [\![1, \mathrm{n_q}]\!],\ \alpha \in [\![1, m]\!]\} \tag{5.10}$$

Let π_h be the $\mathbb{P}^1$-Lagrange interpolation operator valued in the finite element space $\mathrm{H}^{1,h}(\Omega_h)$ given by

$$\pi_h(v) := \sum_{i=1}^{\mathrm{n_q}} v(\mathrm{q}^i)\varphi_i, \quad \forall v \in \mathcal{C}^0(\overline{\Omega_h}). \tag{5.11}$$

5.3 $\mathbb{P}_1$-Lagrange finite elements on boundary meshes

Let $\Sigma_h \subset \Gamma_h$ be a boundary mesh deduced from $\mathcal{T}_h$ and defined in 4.2. We denote by $\mathrm{H}^{1,h}(\Sigma_h)$ the space spanned by the classical $\mathbb{P}_1$-Lagrange basis functions acting on Σ_h and denoted by $(\Sigma_h.\varphi_r)_{r \in [\![1, \Sigma_h.\mathrm{n_q}]\!]}$. Then, for all $g_h \in \mathrm{H}^{1,h}(\Sigma_h)$, we have

$$g_h = \sum_{r=1}^{\Sigma_h.\mathrm{n_q}} g_h(\mathrm{q}^r)\Sigma_h.\varphi_r \tag{5.12}$$

With definition (4.1) of $\mathcal{I}_{\Sigma_h}$, the function $\tilde{g}_h \in \mathrm{H}^{1,h}(\Omega_h)$ defined by

$$\tilde{g}_h = \sum_{i \in \mathcal{I}_{\Sigma_h}} g_h(\mathrm{q}^i)\mathcal{T}_h.\varphi_i \tag{5.13}$$

is the $\mathbb{P}_1$-Lagrange *discrete extension to zero* of function g_h and we have

$$\forall r \in [\![1, \Sigma_h.\mathrm{n_q}]\!] \quad \Sigma_h.\varphi_r \equiv \mathcal{T}_h.\varphi_i \text{ where } i = \mathcal{I}_{\Sigma_h}(r). \tag{5.14}$$

Using usual *trace operator* γ_{Σ_h} acting on Σ_h, we have, $\forall u_h \in \mathrm{H}^{1,h}(\Omega_h)$, $\gamma_{\Sigma_h}(u_h) \in \mathrm{H}^{1,h}(\Sigma_h)$ and

$$\gamma_{\Sigma_h}(u_h) = \sum_{r \in [\![1, \Sigma_h.\mathrm{n_q}]\!]} u_h(\Sigma_h.\mathrm{q}^r)\Sigma_h.\varphi_r. \tag{5.15}$$

Now, we can define the space $\mathrm{H}^{1,h}_{g_h,\Sigma_h}(\Omega_h)$ where $g_h \in \mathrm{H}^{1,h}(\Sigma_h)$ by

$$\mathrm{H}^{1,h}_{g_h,\Sigma_h}(\Omega_h) = \left\{u_h \in \mathrm{H}^{1,h}(\Omega_h) \mid \gamma_{\Sigma_h}(u_h) = g_h\right\} \tag{5.16}$$

If $g_h \neq 0$, $\mathrm{H}^{1,h}_{g_h,\Sigma_h}(\Omega_h)$ is not a vector space. Furthermore, from (4.1), we have $\forall u_h \in \mathrm{H}^{1,h}_{g_h,\Sigma_h}(\Omega_h)$

$$u_h = \sum_{i \in \mathcal{I}^c_{\Sigma_h}} u_h(\mathcal{T}_h.\mathrm{q}^i)\mathcal{T}_h.\varphi_i + \sum_{i \in \mathcal{I}_{\Sigma_h}} g(\mathcal{T}_h.\mathrm{q}^i)\mathcal{T}_h.\varphi_i \tag{5.17}$$

6 From the *scalar* BVP to its matrix representation

In this section the basic technique for solving the scalar BVP (2.2)-(2.4) by $\mathbb{P}_1$-Lagrange finite element method is presented. To do so, ones needs to obtain the linear system coming from the discretization of the variational formulation associated with the scalar BVP. The assembly of the matrix associated to the $\mathcal{L}$ operator and the computation of the right-hand side are detailed. Then we describe how to take into account the boundary conditions. Finally the linear system is solved.

6.1 Variational formulation

The first step is to obtain the variational formulation of the BVP (2.2)-(2.4).

For that we multiply (2.2) by a test function $v \in \mathrm{H}^1(\Omega)$ and by integration on Ω we obtain

$$\int_\Omega \mathcal{L}(u)v d\mathrm{q} = -\int_\Omega \operatorname{div}(\mathbb{A}\nabla u)\, v d\mathrm{q} + \int_\Omega \operatorname{div}(\boldsymbol{b}u)\, v d\mathrm{q} + \int_\Omega \langle \boldsymbol{c}, \nabla u\rangle v d\mathrm{q} + \int_\Omega a_0 uv d\mathrm{q} \tag{6.1}$$

We apply the following theorem on the first two integrals

Theorem : Green Formula [12, Page 11]

If $\boldsymbol{u} \in H(\operatorname{div};\Omega) = \left\{\boldsymbol{w} \in \left(L^2(\Omega)\right)^d \mid \operatorname{div}\boldsymbol{w} \in L^2(\Omega)\right\}$ and $v \in \mathrm{H}^1(\Omega)$ then

$$\int_\Omega v \operatorname{div}\boldsymbol{u} d\mathrm{q} = -\int_\Omega \langle \boldsymbol{u}, \nabla v\rangle d\mathrm{q} + \int_\Gamma \langle \boldsymbol{u}, \boldsymbol{n}\rangle v d\sigma. \tag{6.2}$$

So we have

$$-\int_\Omega \operatorname{div}(\mathbb{A}\nabla u)\, v d\mathrm{q} = \int_\Omega \langle \mathbb{A}\nabla u, \nabla v\rangle d\mathrm{q} - \int_\Gamma \langle \mathbb{A}\nabla u, \boldsymbol{n}\rangle v d\sigma, \tag{6.3}$$

$$\int_\Omega \operatorname{div}(\boldsymbol{b}u)\, v d\mathrm{q} = -\int_\Omega \langle \boldsymbol{b}u, \nabla v\rangle d\mathrm{q} + \int_\Gamma \langle \boldsymbol{b}u, \boldsymbol{n}\rangle v d\sigma. \tag{6.4}$$

Let $\mathcal{D}_\mathcal{L} = \mathcal{D}_{\mathbb{A},\boldsymbol{b},\boldsymbol{c},a_0}$ be the first order bilinear differential operator acting on *scalar fields* associated to the $\mathcal{L}$ operator defined $\forall (u,v) \in (\mathrm{H}^1(\Omega))^2$ by

$$\mathcal{D}_\mathcal{L}(u,v) = \langle \mathbb{A}\nabla u, \nabla v\rangle - (u\langle \boldsymbol{b}, \nabla v\rangle - v\langle \nabla u, \boldsymbol{c}\rangle) + a_0 uv. \tag{6.5}$$

Let $\dfrac{\partial u}{\partial n_\mathcal{L}}$ be the *conormal* derivative of u defined by

$$\frac{\partial u}{\partial n_\mathcal{L}} := \langle \mathbb{A}\nabla u, \boldsymbol{n}\rangle - \langle \boldsymbol{b}u, \boldsymbol{n}\rangle \tag{6.6}$$

So using (6.3) and (6.4) gives

$$\int_\Omega \mathcal{L}(u)v d\mathrm{q} = \int_\Omega \mathcal{D}_\mathcal{L}(u,v) d\mathrm{q} - \int_\Gamma \frac{\partial u}{\partial n_\mathcal{L}} v\, d\sigma, \quad \forall\, v \in \mathrm{H}^1(\Omega) \tag{6.7}$$

The variational formulation associated to the scalar boundary value problem (2.2)-(2.4) is given by

Variational formulation

Find $u \in \mathrm{H}^1_{g^D,\Gamma^D}(\Omega)$ such that

$$\mathcal{A}_{\mathcal{L}}(u,v) = \mathcal{F}(v) \quad \forall v \in \mathrm{H}^1_{0,\Gamma^D}(\Omega) \tag{6.8}$$

where

$$\mathcal{A}_{\mathcal{L}}(u,v) = \int_\Omega \mathcal{D}_{\mathcal{L}}(u,v)d\mathrm{q} + \int_{\Gamma^R} a^R uv d\sigma \tag{6.9}$$

$$\mathcal{F}(v) = \int_\Omega fv d\mathrm{q} + \int_{\Gamma^R} g^R v d\sigma \tag{6.10}$$

By the extension theorem, if $g^D \in \mathrm{H}^{1/2}(\Gamma^D)$, there exists $R^D \in \mathrm{H}^1(\Omega)$ such that $\gamma_{\Gamma^D}(R^D) = g^D$ where γ_{Γ^D} is the trace operator acting on Γ^D. We set $w = u - R^D$. Then the previous variational formulation can be written equivalently as

Variational formulation with an extension function R^D

Find $w \in \mathrm{H}^1_{0,\Gamma^D}(\Omega)$ such that

$$\mathcal{A}_{\mathcal{L}}(w,v) = \mathcal{F}(v) - \mathcal{A}_{\mathcal{L}}(R^D,v) \quad \forall v \in \mathrm{H}^1_{0,\Gamma^D}(\Omega) \tag{6.11}$$

6.2 Discrete variational formulation

The discretisation of the variational formulation (6.8) is given by

Discrete variational formulation

Find $u_h \in \mathrm{H}^{1,h}_{g_h^D,\Gamma_h^D}(\Omega_h)$ such that

$$\mathcal{A}^h_{\mathcal{L}}(u_h,v_h) = \mathcal{F}^h(v_h)\ \forall v_h \in \mathrm{H}^{1,h}_{0,\Gamma_h^D}(\Omega_h) \tag{6.12}$$

where

$$\mathcal{A}^h_{\mathcal{L}}(u_h,v_h) = \int_{\Omega_h} \mathcal{D}_{\mathcal{L}}(u_h,v_h)d\mathrm{q} + \int_{\Gamma_h^R} a^R u_h v_h d\sigma \tag{6.13}$$

$$\mathcal{F}^h(v_h) = \int_{\Omega_h} fv_h d\mathrm{q} + \int_{\Gamma_h^R} g^R v_h d\sigma \tag{6.14}$$

Let $R^D_h \in \mathrm{H}^{1,h}(\Omega_h)$ be the discrete extension by zero of g^D defined by

$$\forall i \in [\![1, \mathcal{T}_h.\mathrm{n_q}]\!],\ R^D_h(\mathrm{q}^i) = \begin{cases} g^D(\mathrm{q}^i) & \text{if } \mathrm{q}^i \in \overline{\Gamma^D_h}, \\ 0 & \text{otherwise.} \end{cases} \tag{6.15}$$

Then the discretisation of the variational formulation with an extension function (6.11) is given by

Discrete variational formulation with an extension function R_h^D

Find $w_h \in \mathrm{H}^{1,h}_{0,\Gamma_h^D}(\Omega_h)$ such that

$$\mathcal{A}_{\mathcal{L}}^h(w_h, v_h) = \mathcal{F}^h(v_h) - \mathcal{A}_{\mathcal{L}}^h(R_h^D, v_h) \quad \forall v_h \in \mathrm{H}^{1,h}_{0,\Gamma_h^D}(\Omega_h) \tag{6.16}$$

The discrete space variational formulations (6.12) and (6.16) are equivalent with $u_h = w_h + R_h^D$.

6.3 Matrix representation of the variational formulation : the *scalar* case

In this section, we set $\mathrm{n_q} = \mathcal{T}_h.\mathrm{n_q}$, $\mathrm{q}^i = \mathcal{T}_h.\mathrm{q}^i$ and $\varphi_i = \mathcal{T}_h.\varphi_i$.
Let $\mathcal{I}_D$ the set defined by

$$\mathcal{I}_D = \left\{ i \in [\![1, \mathrm{n_q}]\!] \mid \mathrm{q}^i \in \overline{\Gamma_h^D} \right\} \tag{6.17}$$

and $\mathcal{I}_D^c$ its complement in $[\![1, \mathrm{n_q}]\!]$.
Let $R_h^D \in \mathrm{H}^{1,h}(\Omega_h)$ be the discrete extension by zero of g^D such that $R_h^D(\mathrm{q}^i) = g^D(\mathrm{q}^i)$, $\forall i \in \mathcal{I}_D$, and $R_h^D(\mathrm{q}^i) = 0$, $\forall i \in \mathcal{I}_D^c$. By bilinearity of $\mathcal{A}_{\mathcal{L}}^h$ and linearity of $\mathcal{F}^h$, the variational formulation (6.16) is equivalent to find $w_h \in \mathrm{H}^{1,h}_{0,\Gamma_h^D}(\Omega_h)$ such that

$$\mathcal{A}_{\mathcal{L}}^h(w_h, \varphi_i) = \mathcal{F}^h(\varphi_i) - \mathcal{A}_{\mathcal{L}}^h(R_h^D, \varphi_i), \ \forall i \in \mathcal{I}_D^c \tag{6.18}$$

We set $\boldsymbol{u}$, $\boldsymbol{w}$ and $\boldsymbol{R}$ as three vectors of $\mathbb{R}^{\mathrm{n_q}}$ such that

$$u_h = \sum_{j=1}^{\mathrm{n_q}} \boldsymbol{u}_j \varphi_j = \sum_{j \in \mathcal{I}_D^c} \boldsymbol{u}_j \varphi_j + \sum_{j \in \mathcal{I}_D} g^D(\mathrm{q}^j)\varphi_j,$$

$$w_h = \sum_{j=1}^{\mathrm{n_q}} \boldsymbol{w}_j \varphi_j = \sum_{j \in \mathcal{I}_D^c} \boldsymbol{w}_j \varphi_j,$$

$$R_h^D = \sum_{j=1}^{\mathrm{n_q}} \boldsymbol{R}_j \varphi_j = \sum_{j \in \mathcal{I}_D} g^D(\mathrm{q}^j)\varphi_j.$$

where u_h is solution of (6.12) and w_h is solution of (6.16). We note that $u_h = w_h + R_h^D$ and $\boldsymbol{u} = \boldsymbol{w} + \boldsymbol{R}$. By construction we have

$$\boldsymbol{R}_j = \begin{cases} g^D(\mathrm{q}^j) & \text{if } j \in \mathcal{I}_D \\ 0 & \text{otherwise.} \end{cases} \tag{6.19}$$

By bilinearity of $\mathcal{A}_{\mathcal{L}}^h$ and linearity of $\mathcal{F}^h$, the variational formulation (6.12) is equivalent to

Matrix representation of the discrete variational formulation (6.12)

Find $\boldsymbol{u} \in \mathbb{R}^{\mathrm{n_q}}$ such that

$$\sum_{j=1}^{\mathrm{n_q}} \mathcal{A}_{\mathcal{L}}^h(\varphi_j, \varphi_i)\boldsymbol{u}_j = \mathcal{F}^h(\varphi_i), \quad \forall i \in \mathcal{I}_D^c, \tag{6.20}$$

$$\boldsymbol{u}_i = g^D(\mathrm{q}^i), \quad \forall i \in \mathcal{I}_D. \tag{6.21}$$

and (6.16) is equivalent to

Matrix representation of the discrete variational formulation with an extension function (6.16)

Find $(\boldsymbol{w}_j)_{j\in\mathcal{I}_D^c}$ such that, $\forall i \in \mathcal{I}_D^c$,

$$\sum_{j\in\mathcal{I}_D^c} \mathcal{A}_{\mathcal{L}}^h(\varphi_j,\varphi_i)\boldsymbol{w}_j = \mathcal{F}^h(\varphi_i) - \sum_{j\in\mathcal{I}_D} \mathcal{A}_{\mathcal{L}}^h(\varphi_j,\varphi_i)\boldsymbol{R}_j. \tag{6.22}$$

We now explain how to obtain the vector $\boldsymbol{u}$ (i.e the solution u_h of (6.12) with $u_h = \sum_{j=1}^{\mathrm{n_q}} \boldsymbol{u}_j\varphi_j$). The $\mathrm{n_q}$-by-$\mathrm{n_q}$ matrix $\mathbb{A}^{\mathcal{L}}$ and the vector $\boldsymbol{b}^{\mathcal{L}} \in \mathbb{R}^{\mathrm{n_q}}$ are defined $\forall (i,j) \in [\![1,\mathrm{n_q}]\!]^2$ by

$$\mathbb{A}^{\mathcal{L}}_{i,j} = \mathcal{A}_{\mathcal{L}}^h(\varphi_j,\varphi_i), \tag{6.23}$$

$$\boldsymbol{b}^{\mathcal{L}}_i = \mathcal{F}^h(\varphi_i). \tag{6.24}$$

More precisely, using (6.13) we have $\forall (i,j) \in [\![1,\mathrm{n_q}]\!]^2$

$$\mathbb{A}^{\mathcal{L}}_{i,j} = \int_{\Omega_h} \mathcal{D}_{\mathcal{L}}(\varphi_j,\varphi_i)d\mathrm{q} + \int_{\Gamma_h^R} a^R\varphi_j\varphi_i d\sigma \tag{6.25}$$

Similarly using (6.14) we have $\forall i \in [\![1,\mathrm{n_q}]\!]$

$$\boldsymbol{b}^{\mathcal{L}}_i = \int_{\Omega_h} f\varphi_i d\mathrm{q} + \int_{\Gamma_h^R} g^R\varphi_i d\sigma \tag{6.26}$$

To make the computations of $\mathbb{A}^{\mathcal{L}}$ and $\boldsymbol{b}^{\mathcal{L}}$ easier, they are split in inner and Robin parts. So we denote by $\mathbb{D}^{\mathcal{L}}$ the $\mathrm{n_q}$-by-$\mathrm{n_q}$ matrix and by $\boldsymbol{b}^f$ the vector of $\mathbb{R}^{\mathrm{n_q}}$ which correspond to the inner parts of (6.25) and (6.26) and verify

$$\mathbb{D}^{\mathcal{L}}_{i,j} = \int_{\Omega_h} \mathcal{D}_{\mathcal{L}}(\varphi_j,\varphi_i)d\mathrm{q} \tag{6.27}$$

$$\boldsymbol{b}^f_i = \int_{\Omega_h} f\varphi_i d\mathrm{q}. \tag{6.28}$$

For the terms corresponding to the *Robin* boundary condition we define a $\mathrm{n_q}$-by-$\mathrm{n_q}$ matrix denoted by $\mathbb{B}^R$ and a vector of $\mathbb{R}^{\mathrm{n_q}}$ denoted by $\boldsymbol{b}^R$ such that

$$\mathbb{B}^R_{i,j} = \int_{\Gamma_h^R} a^R\varphi_j\varphi_i d\sigma \tag{6.29}$$

$$\boldsymbol{b}^R_i = \int_{\Gamma_h^R} g^R\varphi_i d\sigma. \tag{6.30}$$

Finally we have

$$\mathbb{A}^{\mathcal{L}} = \mathbb{D}^{\mathcal{L}} + \mathbb{B}^R \tag{6.31}$$

$$\boldsymbol{b}^{\mathcal{L}} = \boldsymbol{b}^f + \boldsymbol{b}^R. \tag{6.32}$$

We will see later in Sections 6.4, 6.5 and 6.6.1 how to calculate these terms.

The Dirichlet terms will be treated in Qection 6.6.2 and allow to define the sets $\mathcal{I}_D$ and $\mathcal{I}_D^c$ and the extension $\boldsymbol{R}$.

From (6.22) we obtain that the restriction of $\boldsymbol{w}$ to $\mathcal{I}_D^c$ indices $\boldsymbol{w}(\mathcal{I}_D^c)$ is solution of

$$\text{Find } \boldsymbol{x} \text{ such that } \quad \mathbb{A}^{\mathcal{L}}(\mathcal{I}_D^c, \mathcal{I}_D^c)\boldsymbol{x} = \boldsymbol{b}^{\mathcal{L}}(\mathcal{I}_D^c) - \mathbb{A}^{\mathcal{L}}(\mathcal{I}_D^c, \mathcal{I}_D)\boldsymbol{R}(\mathcal{I}_D) \tag{6.33}$$

where $\mathbb{A}^{\mathcal{L}}(\mathcal{I}, \mathcal{J})$ is the submatrix of $\mathbb{A}^{\mathcal{L}}$ obtained by deleting $\mathcal{I}^c$ rows and $\mathcal{J}^c$ columns. By construction of $\boldsymbol{R}$, (6.34) is equivalent to

$$\mathbb{A}^{\mathcal{L}}(\mathcal{I}_D^c, \mathcal{I}_D^c)\boldsymbol{x} = \boldsymbol{b}^{\mathcal{L}}(\mathcal{I}_D^c) - \mathbb{A}^{\mathcal{L}}(\mathcal{I}_D^c, :)\boldsymbol{R} \tag{6.34}$$

In Algorithm 6.1 the successive steps to obtain the vector $\boldsymbol{u} = \boldsymbol{w} + \boldsymbol{R}$ are given.

Algorithm 6.1 Steps for solving the *Scalar* Boundary Value Problem

1: Assembly of the matrix $\mathbb{D}^{\mathcal{L}}$
2: Computation of $\boldsymbol{b}^f$
3: Computation of *Robin* contributions : $\mathbb{B}^R$ and $\boldsymbol{b}^R$
4: Computation of *Dirichlet* contributions : $\mathcal{I}_D$, $\mathcal{I}_D^c$ and $\boldsymbol{R}$.
5: $\boldsymbol{b}^{\mathcal{L}} \leftarrow \boldsymbol{b}^f + \boldsymbol{b}^R$
6: $\mathbb{A}^{\mathcal{L}} \leftarrow \mathbb{D}^{\mathcal{L}} + \mathbb{B}^R$
7: $\boldsymbol{u}(\mathcal{I}_D^c) \leftarrow$ Solve$(\mathbb{A}^{\mathcal{L}}(\mathcal{I}_D^c, \mathcal{I}_D^c), \boldsymbol{b}(\mathcal{I}_D^c) - \mathbb{A}^{\mathcal{L}}(\mathcal{I}_D^c, \mathcal{I}_D)\boldsymbol{R}(\mathcal{I}_D))$
8: $\boldsymbol{u}(\mathcal{I}_D) \leftarrow \boldsymbol{R}(\mathcal{I}_D)$

Now we focus on the lines 1 to 4 in the Algorithm 6.1.

6.4 Matrix assembly : the classical approach

This section deals with the assembly of the matrix $\mathbb{D}^{\mathcal{L}}$ defined by (6.27). We first remind the reader unused to finite element methods of the usual way to assemble the matrix $\mathbb{D}^{\mathcal{L}}$ given in the Algorithm 6.5.
We have for any $(i, j) \in [\![1, \mathrm{n_q}]\!]^2$

$$\mathbb{D}^{\mathcal{L}}_{i,j} = \sum_{k=1}^{\mathrm{n_{me}}} \int_{T^k} \mathcal{D}_{\mathcal{L}}(\varphi_j, \varphi_i) d\mathrm{q}$$

The Algorithm 6.2 corresponds to the naive implementation of $\mathbb{D}^{\mathcal{L}}$ computation using previous formula. We may interchange loops which is done in Algorithm 6.3.

Algorithm 6.2 Naive finite element assembly algorithm

```
1: D^L ← 0
2: for i ← 1 to n_q do
3:    for j ← 1 to n_q do
4:       for k ← 1 to n_me do
5:          D^L_{i,j} ← D^L_{i,j} + ∫_{T^k} D_L(φ_j, φ_i) dq
6:       end for
7:    end for
8: end for
```

Algorithm 6.3 Naive finite element assembly algorithm : loop interchange

```
1: D^L ← 0
2: for k ← 1 to n_me do
3:    for i ← 1 to n_q do
4:       for j ← 1 to n_q do
5:          D^L_{i,j} ← D^L_{i,j} + ∫_{T^k} D_L(φ_j, φ_i) dq
6:       end for
7:    end for
8: end for
```

By property of the supports of the functions φ_i, we have

$$T^k \subset \mathrm{supp}(\varphi_i) \quad \text{iff} \quad \mathrm{q}^i \text{ is a vertex of } T^k.$$

We derive that $\varphi_i \equiv 0$ on T^k iff $i \notin \mathrm{me}(:,k)$ which allows us to write the Algorithm 6.4.

We recall that the $\mathbb{P}^1$-Lagrange basis functions on a d-simplex K are the barycentric coordinates $(\lambda_\alpha)_{\alpha\in[\![1,d+1]\!]}$. We introduce the $(d+1)$-by-$(d+1)$ element matrix $\mathbb{D}^{e,\mathcal{L}}(K)$, associated to $\mathbb{D}^{\mathcal{L}}$ on the d-simplex K and defined by

$$\mathbb{D}^{e,\mathcal{L}}_{\alpha,\beta}(K) = \int_K \mathcal{D}_{\mathcal{L}}(\lambda_\beta, \lambda_\alpha)(\mathrm{q})d\mathrm{q} \quad \forall(\alpha,\beta) \in [\![1,d+1]\!]^2. \tag{6.35}$$

On $K = T^k$, we have $\lambda_\alpha = \varphi_{\mathrm{me}(\alpha,k)}$, $\forall \alpha \in [\![1,d+1]\!]$. Now we can write the Algorithm 6.5 using these element matrices.

Algorithm 6.4 Classical finite element assembly algorithm

```
1:  𝔻^ℒ ← 0
2:  for k ← 1 to n_me do
3:     for α ← 1 to d + 1 do
4:        i ← me(α, k)
5:        for β ← 1 to d + 1 do
6:           j ← me(β, k)
7:           𝔻^ℒ_{i,j} ← 𝔻^ℒ_{i,j} + ∫_{T^k} 𝒟_ℒ(φ_j, φ_i)dq
8:        end for
9:     end for
10: end for
```

Algorithm 6.5 Classical finite element assembly algorithm with element matrices

```
1:  𝔻^ℒ ← 0
2:  for k ← 1 to n_me do
3:     𝔼 ← 𝔻^{e,ℒ}(T^k)
4:     for α ← 1 to d + 1 do
5:        i ← me(α, k)
6:        for β ← 1 to d + 1 do
7:           j ← me(β, k)
8:           𝔻^ℒ_{i,j} ← 𝔻^ℒ_{i,j} + 𝔼_{α,β}
9:        end for
10:    end for
11: end for
```

We also have the classical theorem (see for example [11].

Theorem 4. *Let K be a* d-simplex and $(\lambda_i(\mathrm{q}))_{i\in[\![1,d+1]\!]}$ be the barycentric coordinates of the point $\mathrm{q} \in K$, then for every $n_i \in \mathbb{N}$, $i \in [\![1,d+1]\!]$, we have

$$\int_K \prod_{i=1}^{d+1} \lambda_i^{n_i}(\mathrm{q})d\mathrm{q} = |K| \frac{d! \prod_{i=1}^{d+1} n_i!}{(d + \sum_{i=1}^{d+1} n_i)!}. \tag{6.36}$$

where $|K|$ is the volume of K.

Remark 5. *For any $\alpha \in [\![1,d+1]\!]$ as λ_α is a polynomial of degree 1 on K, its derivatives are constant on K and we set $\lambda^K_{\alpha,i} = \frac{\partial \lambda_\alpha}{\partial x_i}_{|K}$ $\forall i \in [\![1,d]\!]$. In the appendix 11.5 a method for computing these derivatives is proposed and two algorithms are given to compute them : the first one is local (Algorithm 11.2) and the second one is vectorized (Algorithm 11.3).*

To complete the writing, the element matrices $\mathbb{D}^{e,\mathcal{L}}(T_k)$ defined by (6.35) should be computed. This is the goal of Sections 6.4.1 to 6.4.6.

In the sequel we set $\mathrm{n_q} = \mathcal{T}_h.\mathrm{n_q}$, $\varphi_i = \mathcal{T}_h.\varphi_i$, $\mathrm{q}^i = \mathcal{T}_h.\mathrm{q}^i$, $\mathrm{me} = \mathcal{T}_h.\mathrm{me}$ and so on.

6.4.1 Splitting of the operator $\mathcal{D}_{\mathcal{L}}$

Using (6.5) and (6.35), we obtain

$$\mathbb{D}^{e,\mathcal{L}}_{\alpha,\beta}(K) = \sum_{i=1}^{d}\sum_{j=1}^{d}\int_K \mathbb{A}_{i,j}\frac{\partial\lambda_\beta}{\partial x_j}\frac{\partial\lambda_\alpha}{\partial x_i}d\mathrm{q} - \sum_{i=1}^{d}\int_K \boldsymbol{b}_i\frac{\partial\lambda_\alpha}{\partial x_i}\lambda_\beta d\mathrm{q} + \sum_{i=1}^{d}\int_K \boldsymbol{c}_i\frac{\partial\lambda_\beta}{\partial x_i}\lambda_\alpha d\mathrm{q} + \int_K a_0\lambda_\alpha\lambda_\beta d\mathrm{q}.$$

Let $(i,j) \in [\![1,d]\!]^2$ and $g \in L^\infty(\Omega)$, we define the four $(d+1)$-by-$(d+1)$ matrices $\mathbb{D}^e_{uv}(K,g)$, $\mathbb{D}^e_{dudv}(K,g,i,j)$, $\mathbb{D}^e_{duv}(K,g,i)$ and $\mathbb{D}^e_{duv}(K,g,i)$ by $\forall(\alpha,\beta) \in [\![1,d+1]\!]^2$

$$\left(\mathbb{D}^e_{uv}(K,g)\right)_{\alpha,\beta} = \int_K g\lambda_\alpha\lambda_\beta d\mathrm{q}, \tag{6.37}$$

$$\left(\mathbb{D}^e_{dudv}(K,g,i,j)\right)_{\alpha,\beta} = \int_K g\frac{\partial\lambda_\beta}{\partial x_j}\frac{\partial\lambda_\alpha}{\partial x_i}d\mathrm{q} \tag{6.38}$$

$$\left(\mathbb{D}^e_{duv}(K,g,i)\right)_{\alpha,\beta} = \int_K g\frac{\partial\lambda_\beta}{\partial x_i}\lambda_\alpha d\mathrm{q} \tag{6.39}$$

$$\left(\mathbb{D}^e_{udv}(K,g,i)\right)_{\alpha,\beta} = \int_K g\frac{\partial\lambda_\alpha}{\partial x_i}\lambda_\beta d\mathrm{q} \tag{6.40}$$

Then, the element matrix $\mathbb{D}^{e,\mathcal{L}}(K)$ can be written as

$$\mathbb{D}^{e,\mathcal{L}}(K) = \sum_{i=1}^{d}\sum_{j=1}^{d}\mathbb{D}^e_{dudv}(K,\mathbb{A}_{i,j},i,j) - \sum_{i=1}^{d}\mathbb{D}^e_{udv}(K,\boldsymbol{b}_i,i) + \sum_{i=1}^{d}\mathbb{D}^e_{duv}(K,\boldsymbol{c}_i,i) + \mathbb{D}^e_{uv}(K,a_0) \tag{6.41}$$

Thereafter we approximate the function g by its $\mathbb{P}^1$-Lagrange interpolation $g_h = \pi_h g$:

$$g_h = \sum_{i=1}^{\mathrm{n_q}} g_i\varphi_i, \quad \text{where } g_i = g(\mathrm{q}^i).$$

We have on a mesh element $K = T_k$

$$g_h(\mathrm{q}) = \sum_{l=1}^{d+1}\tilde{g}_l\lambda_l(q), \quad \forall \mathrm{q} \in K \tag{6.42}$$

where $\forall l \in [\![1,d+1]\!]$, $\lambda_l = \varphi_i$ and $\tilde{g}_l = g_i$ with $i = \mathrm{me}(l,k)$.

We also set $\tilde{g}^s = \sum_{l=1}^{d+1}\tilde{g}_l$.

6.4.2 Computation of the element matrix $\mathbb{D}^e_{uv}(K,g)$

From (6.37) and (6.42), we have

$$\begin{aligned}(\mathbb{D}^e_{uv}(K,g))_{\alpha,\beta} &= \int_K g(\mathrm{q})\lambda_\beta(\mathrm{q})\lambda_\alpha(\mathrm{q})d\mathrm{q} \approx \int_K g_h(\mathrm{q})\lambda_\beta(\mathrm{q})\lambda_\alpha(\mathrm{q})d\mathrm{q}\\ &\approx \sum_{l=1}^{d+1}\tilde{g}_l\int_K \lambda_l(\mathrm{q})\lambda_\beta(\mathrm{q})\lambda_\alpha(\mathrm{q})d\mathrm{q}.\end{aligned}$$

Formula (6.36) gives

Lemma 6. *We have*

$$(\mathbb{D}^e_{uv}(K,g))_{\alpha,\beta} \approx \frac{d!|K|}{(d+3)!}(1+\delta_{\alpha,\beta})(\tilde{g}_\alpha+\tilde{g}_\beta+\sum_{l=1}^{d+1}\tilde{g}_l). \tag{6.43}$$

This approximation (6.43) *is exact if g is a polynomial of degree 1 or 0 on K. Furthermore, if $g = g_K$ is constant on K we obtain*

$$(\mathbb{D}^e_{uv}(K,g))_{\alpha,\beta} = \frac{d!|K|}{(d+2)!}(1+\delta_{\alpha,\beta})g_K.$$

The Algorithm 6.6 describes the function DElemP1_guv which computes the element matrix.

Algorithm 6.6 function DElemP1_guv. Computation of the element matrix $\mathbb{D}^e_{uv}(K,g)$.

Input :
d : dimension of the simplex K,
vol : K volume,
$\tilde{\boldsymbol{g}}$: 1-by-$(d+1)$ array of g values on K vertices,
Output :
$\mathbb{D}^e$: $(d+1)$-by-$(d+1)$ matrix corresponding to $\mathbb{D}^e_{uv}(K,g)$ given by formula (6.43).

```
Function D^e ← DElemP1_guv( d, vol, g̃)
  g̃^s ← sum(g̃)
  for α ← 1 to d+1 do
    for β ← 1 to d+1 do
      D^e(α,β) ← d!/(d+3)! (1+δ_{α,β}) * vol * (g̃^s + g̃(α) + g̃(β))
    end for
  end for
end Function
```

6.4.3 Computation of the element matrix $\mathbb{D}^e_{dudv}(K,g,i,j)$

Due to the properties of λ_β given in remark 5 and using an interpolation of g as in (6.42), we obtain

$$\begin{aligned}(\mathbb{D}^e_{dudv}(K,g,i,j))_{\alpha,\beta} &= \int_K g(\mathrm{q})\frac{\partial\lambda_\alpha}{\partial x_i}(\mathrm{q})\frac{\partial\lambda_\beta}{\partial x_j}(\mathrm{q})d\mathrm{q}\\ &\approx \lambda^K_{\alpha,i}\,\lambda^K_{\beta,j}\sum_{l=1}^{d+1}\tilde{g}_l\int_K\lambda_l(\mathrm{q})d\mathrm{q}.\end{aligned}$$

Formula (6.36) gives

Lemma 7. *We have*

$$(\mathbb{D}^e_{dudv}(K,g,i,j))_{\alpha,\beta} \approx \frac{d!|K|}{(d+1)!}\lambda^K_{\alpha,i}\,\lambda^K_{\beta,j}\sum_{l=1}^{d+1}\tilde{g}_l \tag{6.44}$$

This approximation (6.44) *is exact if g is a polynomial of degree 1 or 0 on K. Furthermore, if $g = g_K$ is constant on K we obtain*

$$(\mathbb{D}^e_{dudv}(K,g,i,j))_{\alpha,\beta} = |K|\lambda^K_{\alpha,i}\,\lambda^K_{\beta,j}\,g_K.$$

The Algorithm 6.7 describes the function DElemP1_gdudv which computes the element matrix.

Algorithm 6.7 function DElemP1_gdudv. Computation of the element matrix $\mathbb{D}^e_{dudv}(K,g,i,j)$

Input :

d	:	dimension of the simplex K,
vol	:	mesh element volume $\|K\|$,
$\tilde{\boldsymbol{g}}$	:	1-by-$(d+1)$ array of g values on K vertices,
i,j	:	in $[\![1,d]\!]$,
$\boldsymbol{G}$	:	gradients arrays (d-by-$(d+1)$) on a mesh element K $\boldsymbol{G}(i,\alpha) = \lambda^K_{\alpha,i}$, $\forall\alpha \in [\![1,d+1]\!]$

Output :

$\mathbb{D}^e$	:	$(d+1)$-by-$(d+1)$ matrix $\mathbb{D}^e_{dudv}(K,g,i,j)$ given by formula 6.44.

```
Function 𝔻^e ← DElemP1_gdudv( d, vol, g̃, i, j, G)
  g̃^s ← sum(g̃)
  for α ← 1 to d+1 do
    for β ← 1 to d+1 do
      𝔻^e(α,β) ← d!/(d+1)! * vol * g̃^s * G(j,β) * G(i,α)
    end for
  end for
end Function
```

6.4.4 Computation of the element matrix $\mathbb{D}^e_{duv}(K,g,i)$

Due to the properties of λ_α given in remark 5 and using the interpolation of g as in (6.42), we obtain

$$\begin{aligned}(\mathbb{D}^e_{duv}(K,g,i))_{\alpha,\beta} &= \int_K g(\mathrm{q})\frac{\partial\lambda_\beta}{\partial x_i}(\mathrm{q})\lambda_\alpha(\mathrm{q})d\mathrm{q}\\ &\approx \lambda^K_{\beta,i}\sum_{l=1}^{d+1}\tilde{g}_l\int_K \lambda_l(\mathrm{q})\lambda_\alpha(\mathrm{q})d\mathrm{q}.\end{aligned}$$

Formula (6.36) gives

Lemma 8.

$$(\mathbb{D}^e_{duv}(K,g,i))_{\alpha,\beta} \approx \frac{d!|K|}{(d+2)!}\lambda^K_{\beta,i}\,(\tilde{g}_\alpha + \sum_{l=1}^{d+1}\tilde{g}_l). \tag{6.45}$$

This approximation (6.45) *is exact if g is a polynomial of degree 1 or 0 on K. Furthermore, if $g = g_K$ is constant on K we obtain*

$$(\mathbb{D}^e_{duv}(K,g,i))_{\alpha,\beta} = \frac{d!|K|}{(d+1)!}\lambda^K_{\beta,i}g_K.$$

The Algorithm 6.8 describes the function DElemP1_gduv which computes the element matrix.

Algorithm 6.8 Function DElemP1_gduv. Computation of the element matrix $\mathbb{D}^e_{duv}(K,g,i)$ on a d-simplex K.

Input :

d	:	dimension of the simplex K,
vol	:	mesh element volume $\|K\|$,
$\tilde{\boldsymbol{g}}$	:	1-by-$(d+1)$ array of g values on K vertices,
i	:	index $\in [\![1,d]\!]$,
$\boldsymbol{G}$	:	d-by-$(d+1)$ array of gradients on the mesh element K $\boldsymbol{G}(i,\alpha) = \lambda^K_{\alpha,i}$, $\forall \alpha \in [\![1,d+1]\!]$

Output :

$\mathbb{D}^e$	:	$(d+1)$-by-$(d+1)$ matrix $\mathbb{D}^e_{duv}(K,g,i)$ given by (6.45).

1: **Function** $\mathbb{D}^e \leftarrow$ DElemP1_gduv(d, vol, $\tilde{\boldsymbol{g}}, i, \boldsymbol{G}$)
2: $\tilde{g}_s \leftarrow$ sum($\tilde{\boldsymbol{g}}$)
3: **for** $\alpha \leftarrow 1$ **to** $d+1$ **do**
4: **for** $\beta \leftarrow 1$ **to** $d+1$ **do**
5: $\mathbb{D}^e(\alpha,\beta) \leftarrow \frac{d!}{(d+2)!} * \text{vol} * (\tilde{g}_s + \tilde{\boldsymbol{g}}(\alpha)) * \boldsymbol{G}(i,\beta)$
6: **end for**
7: **end for**
8: **end Function**

6.4.5 Computation of the element matrix $\mathbb{D}^e_{udv}(K,g,i)$

Let $i \in [\![1,d]\!]$, $g \in L^\infty(\Omega)$. Due to the properties of λ_α given in remark 5 and using the interpolation of g (6.42), we obtain

$$\begin{aligned}(\mathbb{D}^e_{udv}(K,g,i))_{\alpha,\beta} &= \int_K g(\mathrm{q})\lambda_\beta(\mathrm{q})\frac{\partial \lambda_\alpha}{\partial x_i}(\mathrm{q})d\mathrm{q} \\ &\approx \lambda^K_{\alpha,i}\sum_{l=1}^{d+1}\tilde{g}_l\int_K \lambda_l(\mathrm{q})\lambda_\beta(\mathrm{q})d\mathrm{q}.\end{aligned}$$

Formula (6.36) gives

Lemma 9.

$$(\mathbb{D}^e_{udv}(K,g,i))_{\beta,\alpha} \approx \frac{d!|K|}{(d+2)!}\lambda^K_{\alpha,i}(\tilde{g}_\beta + \sum_{l=1}^{d+1}\tilde{g}_l). \tag{6.46}$$

This approximation compute exact if g is polynomial of degree 1 or 0 on K. Furthermore, if $g = g_K$ is constant on K we obtain

$$(\mathbb{D}^e_{udv}(K,g,i))_{\alpha,\beta} = \frac{d!|K|}{(d+1)!}\lambda^K_{\alpha,i}g_K.$$

The Algorithm 6.9 describes the function DElemP1_gudv which computes the element matrix.

Algorithm 6.9 Function DElemP1_gudv. Computation of the element matrix $\mathbb{D}^e_{udv}(K, g, i)$ on a mesh element K.

Input :

d	:	dimension of the simplex K,
vol	:	mesh element volume $\|K\|$,
$\tilde{\boldsymbol{g}}$	:	1-by-$(d+1)$ array of g values on K vertices,
i	:	in $[\![1, d]\!]$,
$\boldsymbol{G}$	:	d-by-$(d+1)$ array of gradients on the mesh element K $\boldsymbol{G}(i, \alpha) = \lambda^K_{\alpha,i}$, $\forall \alpha \in [\![1, d+1]\!]$

Output :

$\mathbb{D}^e$	:	$(d+1)$-by-$(d+1)$ matrix $\mathbb{D}^e_{udv}(K, g, i)$ given by 6.46.

1: **Function** $\mathbb{D}^e \leftarrow$ DElemP1_gudv($d, \text{vol}, \tilde{\boldsymbol{g}}, i, \boldsymbol{G}$)
2: $\tilde{g}_s \leftarrow$ sum($\tilde{\boldsymbol{g}}$)
3: **for** $\alpha \leftarrow 1$ **to** $d+1$ **do**
4: **for** $\beta \leftarrow 1$ **to** $d+1$ **do**
5: $\mathbb{D}^e(\alpha, \beta) \leftarrow \frac{d!}{(d+2)!} * \text{vol} * (\tilde{g}_s + \tilde{\boldsymbol{g}}(\beta)) * \boldsymbol{G}(i, \alpha)$
6: **end for**
7: **end for**
8: **end Function**

6.4.6 Computation of the element matrix $\mathbb{D}^{e,\mathcal{L}}(K)$

Using (6.41) and the previous functions, the Algorithm 6.10 specifies the function DElemP1 which computes the element matrix $\mathbb{D}^{e,\mathcal{L}}(K)$. We can notice that the gradients of $\mathbb{P}^1$-Lagrange local functions are not computed if the order of the operator $\mathcal{L}$ is zero (i.e. if $\mathbb{A}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ are zero).

Algorithm 6.10 function DElemP1

Input :

$\mathcal{T}_h$	:	mesh structure associated to Ω_h (see Section 4.1)
$\mathcal{L}_h$	:	discretized $\mathcal{L}$ operator,
k	:	index of the k-th mesh element $K = T_k$,

Output :

$\mathbb{D}^e$	:	$(d+1)$-by-$(d+1)$ matrix with $d = \mathcal{T}_h.d$.

1: **Function** $\mathbb{D}^e \leftarrow$ DElemP1($\mathcal{T}_h, \mathcal{L}_h, k$)
2: vol $\leftarrow \mathcal{T}_h$.vols(k)
3: $\mathbb{D}^e \leftarrow$ DElemP1_guv(d, vol, getLocFdata($\mathcal{L}_h.a0, \mathcal{T}_h, k$))
4: **if** $\mathcal{L}_h$.order > 0 **then**
5: $\boldsymbol{G} \leftarrow$ Gradients($d, \mathcal{T}_h$.q(:, $\mathcal{T}_h$.me(:, k)), vol)
6: **for** $i \leftarrow 1$ **to** d **do**
7: **for** $j \leftarrow 1$ **to** d **do**
8: $\mathbb{D}^e \leftarrow \mathbb{D}^e +$DElemP1_gdudv($d$, vol, getLocFdata($\mathcal{L}_h.A(i, j), \mathcal{T}_h, k$), i, j, G)
9: **end for**
10: $\mathbb{D}^e \leftarrow \mathbb{D}^e -$ DElemP1_gudv(d, vol, getLocFdata($\mathcal{L}_h.b(i), \mathcal{T}_h, k$), i, G)
11: $\mathbb{D}^e \leftarrow \mathbb{D}^e +$ DElemP1_gduv(d, vol, getLocFdata($\mathcal{L}_h.c(i), \mathcal{T}_h, k$), i, G)
12: **end for**
13: **end if**
14: **end Function**

See also : DElemP1_guv, Alg. 6.6 - DElemP1_gdudv, Alg. 6.7 - DElemP1_gudv, Alg. 6.9 - DElemP1_gduv, Alg. 6.8 - Gradients, Alg. 11.2 - getLocFdata, Alg. 4.6.

6.4.7 Assembly of the matrix $\mathbb{D}^{\mathcal{L}}$

From classical Algorithm 6.5, we can write the function DAssemblyP1_base given in Algorithm 6.11 which computes the matrix $\mathbb{D}^{\mathcal{L}}$ defined by (6.27).

Algorithm 6.11 Function DAssemblyP1_base. Matrix $\mathbb{D}^{\mathcal{L}}$ assembly

Input :

$\mathcal{T}_h$: mesh structure associated to Ω_h (see Section 4)

$\mathcal{L}$: `Loperator` structure (see Section 4)

Output :

$\mathbb{D}^{\mathcal{L}}$: $\mathcal{T}_h.\mathrm{n_q}$-by-$\mathcal{T}_h.\mathrm{n_q}$ sparse matrix.

Function $\mathbb{D}^{\mathcal{L}} \leftarrow$ DAssemblyP1_base$(\mathcal{T}_h, \mathcal{L})$
  $\mathbb{D}^{\mathcal{L}} \leftarrow$ Sparse$(\mathcal{T}_h.\mathrm{n_q}, \mathcal{T}_h.\mathrm{n_q})$
  $\mathcal{L}_h \leftarrow$ setDdata$(\mathcal{L}, \mathcal{T}_h)$
  for $k \leftarrow 1$ **to** $\mathcal{T}_h.\mathrm{n_{me}}$ **do**
    $\mathbb{D}^e \leftarrow$ DElemP1$(\mathcal{T}_h, \mathcal{L}_h, k)$
    for $\alpha \leftarrow 1$ **to** $\mathcal{T}_h.d + 1$ **do**
      $s \leftarrow \mathcal{T}_h.\mathrm{me}(\alpha, k)$
      for $\beta \leftarrow 1$ **to** $\mathcal{T}_h.d + 1$ **do**
        $t \leftarrow \mathcal{T}_h.\mathrm{me}(\beta, k)$
        $\mathbb{D}^{\mathcal{L}}_{s,t} \leftarrow \mathbb{D}^{\mathcal{L}}_{s,t} + \mathbb{D}^e_{\alpha,\beta}$
      end for
    end for
  end for
end Function

We can use this function to compute, for example,

- the computation of the *Mass* matrix $\mathbb{M}$ associated to the operator $\mathcal{L}_{\mathbb{O},\mathbf{0},1}$ is given in Algorithm 6.12.

Algorithm 6.12 Computation of the *Mass* matrix $\mathbb{M}$ in dimension d

`LMass` $\leftarrow$ Loperator$(d, \mathbb{O}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, 1)$
$\mathbb{M} \leftarrow$ DAssemblyP1_base$(\mathcal{T}_h$,`LMass`$)$

- the computation of the *Stiffness* matrix $\mathbb{S}$ associated to the operator $\mathcal{L}_{\mathbb{I},\mathbf{0},0}$ is given in Algorithm 6.13.

Algorithm 6.13 Computation of the *Stiffness* matrix $\mathbb{S}$

`LStiff` $\leftarrow$ Loperator$(d, \mathbb{I}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, 0)$
$\mathbb{S} \leftarrow$ DAssemblyP1_base$(\mathcal{T}_h$,`LStiff`$)$

As the function DAssemblyP1_base achieves the first line of the Algorithm 6.1 the computation of $\boldsymbol{b}^f$ defined by (6.28) needs to be done now.

6.5 Computation of the vector $\boldsymbol{b}^f$

To compute the vector defined in (6.28), we use the $\mathbb{P}^1$-Lagrange interpolate $\pi_h(f)$ of f (see (5.11)) and we set $\boldsymbol{f} \in \mathbb{R}^{\mathrm{n_q}}$ as the vector defined by $\boldsymbol{f}_i = f(\mathrm{q}^i)$, $\forall i \in [\![1, \mathrm{n_q}]\!]$. Then we obtain

$$\boldsymbol{b}_i^f = \int_{\Omega_h} f\varphi_i d\mathrm{q} \approx \int_{\Omega_h} \pi_h(f)\varphi_i d\mathrm{q} = \sum_{j=1}^{\mathrm{n_q}} \boldsymbol{f}_j \int_{\Omega_h} \varphi_j\varphi_i \, d\mathrm{q}.$$

So if we set $\mathbb{M}$ as the Mass matrix of dimension $\mathrm{n_q}$-by-$\mathrm{n_q}$ defined by $\mathbb{M}_{i,j} = \int_{\Omega_h} \varphi_j\varphi_i d\mathrm{q}$ then we have

$$\boldsymbol{b}^f \approx \mathbb{M}\boldsymbol{f}.$$

The computation is summarized in the function RHS in the Algorithm 6.14.

Algorithm 6.14 RHS function: $\mathbb{P}^1$-Lagrange approximation of the right-hand side $\boldsymbol{b}^f$ defined by (6.28)

Input :
$\mathcal{T}_h$: mesh structure associated to Ω_h (see Section 4)
f : function from Ω to $\mathbb{K}$
Output :
$\boldsymbol{b}^f$: vector of dimension $\mathrm{n_{dof}} = \mathcal{T}_h.\mathrm{n_q}$

1: **Function** $\boldsymbol{b}^f \leftarrow \mathrm{RHS}(\mathcal{T}_h, f)$
2: DMass $\leftarrow$ Loperator$(\mathcal{T}_h.d, \mathbb{O}, \mathbf{0}, 1)$
3: $\mathbb{M} \leftarrow$ DAssemblyP1_base$(\mathcal{T}_h, \mathrm{DMass})$ ▷ see Algorithm 6.11
4: $\boldsymbol{b}^f \leftarrow \mathbb{M} *$ setFData$(f, \mathcal{T}_h)$ ▷ see Algorithm 4.5
5: **end Function**

Now the first two lines of the Algorithm 6.1 have been explained. We now look at the remaining lines which contain the contributions of the boundary conditions.

6.6 Boundary conditions

As seen in Algorithm 6.1, we need to compute the contributions of the boundary conditions.

- *Generalized Robin* boundary condition : it means to define the vector $\boldsymbol{b}^R$ and matrix $\mathbb{B}^R$ respectively given by (6.30) and (6.29),
- *Dirichlet* boundary condition : it means to define the sets $\mathcal{I}_D$, $\mathcal{I}_D^c$ and the vector $\boldsymbol{R}$ respectively set in Section 6.3.

6.6.1 Robin boundary conditions

From formula (4.6), the discrete *generalized Robin* boundary is given by

$$\Gamma_h^R = \bigcup_{l \in \mathcal{I}_{labels}^R} \Gamma_h^{labels(l)}.$$

As seen in Algorithm 6.1, we must compute the contributions of *generalized Robin* boundary conditions given by the vector $\boldsymbol{b}^R \in \mathbb{R}^{\mathrm{n_q}}$ and the $\mathrm{n_q}$-by-$\mathrm{n_q}$ matrix $\mathbb{A}^R$ respectively defined in (6.30) and (6.29).

We first explain how to compute the vector $\boldsymbol{b}^R \in \mathbb{R}^{\mathcal{T}_h.nq}$. For that, we set, for $l \in \mathcal{I}_{labels}^R$, g^l as the restriction of g^R to $\Gamma_h^{labels(l)}$ given by $g^l = \texttt{bclR}(1, l).g$. So we obtain

$$\boldsymbol{b}_i^R = \sum_{l \in \mathcal{I}_{labels}^R} \int_{\Gamma_h^{labels(l)}} g^l \varphi_i d\mathrm{q}.$$

We set, $\forall l \in \mathcal{I}^R_{labels}$, $\boldsymbol{b}^l \in \mathbb{R}^{\mathcal{T}_h.\mathrm{n_q}}$ as the vector verifying $\forall i \in [\![1, \mathrm{n_q}]\!]$

$$\boldsymbol{b}^l_i = \int_{\Gamma_h^{labels(l)}} g^l \, \varphi_i \, d\mathrm{q}, \tag{6.47}$$

and then

$$\boldsymbol{b}^R = \sum_{l \in \mathcal{I}^R_{labels}} \boldsymbol{b}^l. \tag{6.48}$$

From notations of Section 4.2, the data structure associated to $\Gamma_h^{labels(l)}$ is given by $\mathcal{B}_h(l)$. Let $l \in \mathcal{I}^R_{labels}$ and $\mathrm{n_{q,}}_l = \mathcal{B}_h(l).\mathrm{n_q}$. We suppose that $g^l \in \mathrm{H}^{1/2}(\Gamma^{labels(l)})$. So the $\mathbb{P}^1$-Lagrange interpolation of g^l, defined in (5.12), is given by

$$g^l_h(\mathrm{q}) = \sum_{r=1}^{\mathrm{n_{q,}}_l} g^l(\mathcal{B}_h(l).\mathrm{q}^r)\mathcal{B}_h(l).\varphi_r(q), \ \forall \mathrm{q} \in \Gamma_h^{labels(l)}.$$

Then the $\mathbb{P}^1$-Lagrange approximation of $\boldsymbol{b}^{R,l}_i$ is

$$\boldsymbol{b}^l_i \approx \sum_{r=1}^{\mathrm{n_{q,}}_l} g^l(\mathcal{B}_h(l).\mathrm{q}^r) \int_{\Gamma_h^{labels(l)}} \mathcal{B}_h(l).\varphi_r(q) \times \mathcal{T}_h.\varphi_i(\mathrm{q}) d\mathrm{q}. \tag{6.49}$$

From (4.2) and $\mathbb{P}^1$-Lagrange basis function properties, we deduce that if $i \notin \mathcal{I}_l$ then $\mathcal{T}_h.\varphi_i \equiv 0$ on $\Gamma_h^{labels(l)}$ and thus $\boldsymbol{b}^l_i = 0$. Otherwise, there exist $s \in [\![1, \mathrm{n_{q,}}_l]\!]$ such that $i = \mathcal{I}_l(s)$. So we obtain

$$\text{on } \Gamma_h^{labels(l)}, \quad \mathcal{T}_h.\varphi_i = \begin{cases} 0 & \text{if } i \notin \mathcal{I}_l \\ \mathcal{B}_h(l).\varphi_s & \text{if } i \in \mathcal{I}_l, \text{ with } i = \mathcal{I}_l(s). \end{cases} \tag{6.50}$$

Now we set $\mathbb{M}^l$ as the $\mathrm{n_{q,}}_l$-by-$\mathrm{n_{q,}}_l$ *boundary Mass matrix* associated to $\Gamma_h^{labels(l)}$ defined by

$$\mathbb{M}^l_{r,s} = \int_{\Gamma_h^{labels(l)}} \mathcal{B}_h(l).\varphi_r \times \mathcal{B}_h(l).\varphi_s d\mathrm{q} \ \forall (r,s) \in [\![1, \mathrm{n_{q,}}_l]\!]^2. \tag{6.51}$$

Let $\boldsymbol{g}^l \in \mathbb{R}^{\mathrm{n_{q,}}_l}$ such that $\boldsymbol{g}^l_r = g^l(\mathcal{B}_h(l).\mathrm{q}^r)$, $\forall r \in [\![1, \mathrm{n_{q,}}_l]\!]$. Then from (6.51), we have

$$\begin{array}{lcll} \boldsymbol{b}^l_{\mathcal{I}_l(s)} & \approx & (\mathbb{M}^l \boldsymbol{g}^l)_s & \forall s \in [\![1, \mathrm{n_{q,}}_l]\!] \\ \boldsymbol{b}^l_i & = & 0 & \forall i \in [\![1, \mathrm{n_q}]\!] \backslash \mathcal{I}_l. \end{array}$$

So in vector form, we obtain

$$\begin{cases} \boldsymbol{b}^l(\mathcal{I}_l) & \approx & \mathbb{M}^l \boldsymbol{g}^l \\ \boldsymbol{b}^l(\mathcal{I}_l^c) & = & 0. \end{cases} \tag{6.52}$$

As $\Gamma_h^{labels(l)}$ is a mesh of (d−1)-simplices in $\mathbb{R}^d$, one could compute the matrix $\mathbb{M}^l$ with the DAssemblyP1_base function given in Algorithm 6.11. A complete algorithm to calculate $\boldsymbol{b}^R$ is given in the Algorithm 6.15.

Algorithm 6.15 Vector $\boldsymbol{b}^R$ assembly

```
LMass ← Loperator(T_h.d, 𝕆_{d×d}, 0_d, 0_d, 1)
b^R ← O_{n_q}
for l ∈ I^R_labels do
  g^l ← setFdata(pde.bclR(l).g, pde.B_h(l))
  M^l ← DAssemblyP1_base(B_h(l), LMass)
  I_l ← B_h(l).toGlobal
  b^R(I_l) ← b^R(I_l) + M^l * g^l
end for
```

As a second step we explain how to compute the matrix $\mathbb{B}^R$. Let $l \in \mathcal{I}^R_{labels}$ and a^l be the restriction of a^R to $\Gamma_h^{labels(l)}$ given by $a^l = \texttt{bclR}(1,l).\texttt{ar}$. Let $\mathbb{B}^l$ $\mathbb{B}^l$ be the $\mathrm{n_q}$-by-$\mathrm{n_q}$ matrix defined by

$$\mathbb{B}^l_{i,j} = \int_{\Gamma_h^{labels(l)}} a^l(\mathrm{q}) \times \mathcal{T}_h.\varphi_j(\mathrm{q}) \times \mathcal{T}_h.\varphi_i(\mathrm{q}) d\mathrm{q}, \ \forall (i,j) \in [\![1, \mathrm{n_q}]\!]^2.$$

Then, we have

$$\mathbb{B}^R_{i,j} = \sum_{l \in \mathcal{I}^R_{labels}} \mathbb{B}^l_{i,j}$$

Using (6.50), if $i \notin \mathcal{I}_l$ and $j \notin \mathcal{I}_l$ we have $\mathbb{B}^l_{i,j} = 0$. Otherwise, there exists $(r,s) \in [\![1, \mathrm{n_{q,}}_l]\!]^2$ such that $i = \mathcal{I}_l(r)$ and $j = \mathcal{I}_l(s)$. So we obtain

$$\mathbb{B}^l_{i,j} = \int_{\Gamma_h^{labels(l)}} a^l(\mathrm{q}) \times \mathcal{B}_h(l).\varphi_s(\mathrm{q}) \times \mathcal{B}_h(l).\varphi_r(\mathrm{q}) d\mathrm{q}$$

We define the $\mathrm{n_{q,}}_l$-by-$\mathrm{n_{q,}}_l$ *boundary weighted Mass matrix* by

$$\mathbb{W}^l_{r,s} = \int_{\Gamma_h^{labels(l)}} a^l(\mathrm{q}) \times \mathcal{B}_h(l).\varphi_s(\mathrm{q}) \times \mathcal{B}_h(l).\varphi_r(\mathrm{q}) d\mathrm{q} \tag{6.53}$$

and then we obtain

$$\mathbb{B}^l_{i,j} = \begin{cases} 0 & \text{if } i \notin \mathcal{I}_l \text{ and } j \notin \mathcal{I}_l, \\ \mathbb{W}^l_{r,s} & \text{otherwise, with } i = \mathcal{I}_l(r) \text{ and } j = \mathcal{I}_l(s). \end{cases} \tag{6.54}$$

Each matrix $\mathbb{W}^l$ can be approximated by using the DAssemblyP1_base function (see Algorithm 6.11) and then we compute a $\mathbb{P}^1$-Lagrange approximation of $\mathbb{B}^R$ by using Algorithm 6.16

Algorithm 6.16 Matrix $\mathbb{B}^R$ assembly

$d \leftarrow \mathcal{T}_h$.d
$\mathbb{B}^R \leftarrow \mathbb{O}$
for $l \in \mathcal{I}^R_{labels}$ **do**
  LMassW $\leftarrow$ Loperator$(d, \mathbb{O}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, \mathrm{bcl}(l).\mathrm{ar})$
  $\mathbb{W}^l \leftarrow$ DAssemblyP1_base$(\mathcal{B}_h(l), \mathrm{LMassW})$
  $\mathcal{I}_l \leftarrow \mathcal{B}_h(l)$.toGlobal
  $\mathbb{B}^R(\mathcal{I}_l, \mathcal{I}_l) \leftarrow \mathbb{B}^R(\mathcal{I}_l, \mathcal{I}_l) + \mathbb{W}^l$
end for

A complete algorithm to calculate $\mathbb{B}^R$ and $\boldsymbol{b}^R$ is given in Algorithm 6.17 in the function RobinBC.

Algorithm 6.17 function RobinBC (scalar version) : computation of the $\boldsymbol{b}^R$ vector and the $\mathbb{B}^R$ matrix

Input :

pde : a PDE structure ▷ see paragraph 4.5

Output :

$\boldsymbol{b}^R$: vector of dimension $\mathrm{n_{dof}} = \mathrm{pde}.\mathcal{T}_h.\mathrm{n_q}$ defined by (6.30)

$\mathbb{B}^R$: $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ matrix defined by (6.29).

1: **Function** $[\mathbb{B}^R, \boldsymbol{b}^R] \leftarrow$ RobinBC(pde)
2: $\mathrm{n_{dof}} \leftarrow \mathrm{pde}.\mathcal{T}_h.\mathrm{n_q}$, $d \leftarrow$ pde.d
3: $\boldsymbol{b}^R \leftarrow \mathbf{0}_{\mathrm{n_{dof}}}$
4: $\mathbb{B}^R \leftarrow \mathbb{O}_{\mathrm{n_{dof}} \times \mathrm{n_{dof}}}$ ▷ sparse matrix
5: DMass $\leftarrow$ Loperator$(d, \mathbb{O}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, 1)$
6: **for** $l \leftarrow 1$ **to** $\mathrm{pde}.\mathrm{n_{lab}}$ **do** ▷ loop over boundary meshes
7: $\mathcal{I}_l \leftarrow \mathrm{pde}.\mathcal{B}_h(l)$.toGlobal
8: **if** $\mathrm{pde.bclR}(1,l).\mathrm{g} \neq \varnothing$ **then**
9: $\mathbb{M}^l \leftarrow$ DAssemblyP1_base$(\mathrm{pde}.\mathcal{B}_h(l), \mathrm{DMass})$
10: $\boldsymbol{g}^l \leftarrow$ setFdata$(\mathrm{pde.bclR}(l).g, \mathrm{pde}.\mathcal{B}_h(l))$
11: $\boldsymbol{b}^R(\mathcal{I}_l) \leftarrow \boldsymbol{b}^R(\mathcal{I}_l) + \mathbb{M}^l * \boldsymbol{g}^l$
12: **end if**
13: **if** $\mathrm{pde.bclR}(1,l).\mathrm{ar} \neq \varnothing$ **then**
14: DMassR $\leftarrow$ Loperator$(\mathrm{pde.d}, \mathbb{O}, \mathbf{0}, \mathbf{0}, \mathrm{pde.bclR}(l).\mathrm{ar})$
15: $\mathbb{W}^l \leftarrow$ DAssemblyP1_base$(\mathrm{pde}.\mathcal{B}_h(l), \mathrm{DMassR})$
16: $\mathbb{B}^R(\mathcal{I}_l, \mathcal{I}_l) \leftarrow \mathbb{B}^R(\mathcal{I}_l, \mathcal{I}_l) + \mathbb{W}^l$
17: **end if**
18: **end for**
19: **end Function**

See also : DAssemblyP1_base, Alg. 6.11 - setFdata, Alg. 4.5 - Loperator, Alg. 4.3

6.6.2 Dirichlet boundary conditions

As explained in Section 6.3 and Algorithm 6.1 taking into account *Dirichlet* boundary conditions requires to compute the sets $\mathcal{I}_D$ and $\mathcal{I}_D^c$ as well as the discrete extension by zero of g^D given by the vector $\boldsymbol{R}$.

Using the notations of Sections 4.2 and 4.3, we have

$$\Gamma_h^D = \bigcup_{l \in \mathcal{I}^D_{labels}} \Gamma_h^{labels(l)}.$$

Let g^l be the restriction of g^D to $\Gamma_h^{labels(l)}$ given by $g^l = \mathtt{bclD}(1,l).\mathtt{g}$. Then we obtain

$$\begin{aligned} \mathcal{I}_D &= \bigcup_{l \in \mathcal{I}^D_{labels}} \mathcal{I}_l, \\ \mathcal{I}_D^c &= [\![1, \mathrm{n_q}]\!] \backslash \mathcal{I}_D \end{aligned}$$

and

$$\begin{aligned} \boldsymbol{R}_{\mathcal{I}_l(r)} &= g^l(\mathcal{B}_h(l).\mathrm{q}^r),\ \forall l \in \mathcal{I}^D_{labels},\ \forall r \in [\![1, \mathcal{B}_h(l).\mathrm{n_q})]\!], \\ \boldsymbol{R}_i &= 0,\ \forall i \in \mathcal{I}_D^c \end{aligned}$$

This is the object of the Algorithm 6.18 and it completes the writing of the Algorithm 6.1 which is detailed is the next section.

Algorithm 6.18 function DirichletBC (scalar version)

Input :

pde : a PDE structure

Output :

$\mathcal{I}_D$: $\mathcal{I}_D = \left\{ i \in [\![1, \mathrm{n_{dof}}]\!] \mid \mathrm{q}^i \in \overline{\Gamma^D} \right\}$ with $\mathrm{n_{dof}} = \mathrm{pde}.\mathcal{T}_h.\mathrm{n_q}$

$\mathcal{I}_D^c$: complement of $\mathcal{I}_D$ in $[\![1, \mathrm{n_{dof}}]\!]$

$\boldsymbol{R}$: vector of dimension $\mathrm{n_{dof}}$ defined by (6.19).

Function $[\mathcal{I}_D, \mathcal{I}_D^c, \boldsymbol{R}] \leftarrow$ DirichletBC(pde)
  $\mathrm{n_{dof}} \leftarrow \mathrm{pde}.\mathcal{T}_h.\mathrm{n_q}$
  $\boldsymbol{R} \leftarrow \mathbf{0}_{\mathrm{n_{dof}}}$
  $\mathcal{I}_D \leftarrow \varnothing$
  for $l \leftarrow 1$ **to** $\mathrm{pde.n_{lab}}$ **do** ▷ loop over boundary meshes
    if $\mathrm{pde.bclD}(l).\mathrm{g} \neq \varnothing$ **then** ▷ Dirichlet boundary condition
      $\mathcal{I}_l \leftarrow \mathrm{pde}.\mathcal{B}_h(l).\mathrm{toGlobal}$
      $\boldsymbol{R}(\mathcal{I}_l) \leftarrow$ setFdata$(\mathrm{pde.bclD}(l).\mathrm{g}, \mathrm{pde}.\mathcal{B}_h(l))$
      $\mathcal{I}_D \leftarrow \mathcal{I}_D \cup \mathcal{I}_l$
    end if
  end for
  $\mathcal{I}_D^c \leftarrow [\![1, \mathrm{n_{dof}}]\!] \backslash \mathcal{I}_D$
end Function

6.7 Construction and solution of the linear system

From Algorithm 6.1 and using Algorithms 6.11 to 6.18, the function SolvePDE in the Algorithm 6.19 creates and solves the linear system associated to the scalar boundary value problem (2.2) to (2.4) described in a PDE structure.

Algorithm 6.19 function SolvePDE : construction and solution of a scalar BVP

Input :

pde : a PDE structure ▷ see paragraph 4.5

Output :

$\boldsymbol{u}$: vector of dimension $\mathrm{n_{dof}} = \mathrm{pde}.\mathcal{T}_h.\mathrm{n_q}$.

Function $\boldsymbol{u} \leftarrow$ SolvePDE(pde)
  $\mathbb{D}^{\mathcal{L}} \leftarrow$ DAssemblyP1_base$(\mathrm{pde}.\mathcal{T}_h, \mathrm{pde.op})$
  $\boldsymbol{b}^f \leftarrow$ RHS$(\mathrm{pde}.\mathcal{T}_h, \mathrm{pde.f})$
  $[\boldsymbol{b}^R, \mathbb{B}^R] \leftarrow$ RobinBC(pde)
  $[\mathcal{I}_D, \mathcal{I}_D^c, \boldsymbol{R}] \leftarrow$ DirichletBC(pde)
  $\boldsymbol{b}^{\mathcal{L}} \leftarrow \boldsymbol{b}^f + \boldsymbol{b}^R$
  $\mathbb{A}^{\mathcal{L}} \leftarrow \mathbb{D}^{\mathcal{L}} + \mathbb{B}^R$
  $\boldsymbol{u}(\mathcal{I}_D^c) \leftarrow$ Solve$(\mathbb{A}^{\mathcal{L}}(\mathcal{I}_D^c, \mathcal{I}_D^c), \boldsymbol{b}(\mathcal{I}_D^c) - \mathbb{A}^{\mathcal{L}}(\mathcal{I}_D^c, \mathcal{I}_D)\boldsymbol{R}(\mathcal{I}_D))$
  $\boldsymbol{u}(\mathcal{I}_D) \leftarrow \boldsymbol{R}(\mathcal{I}_D)$
end Function

7 From the vector BVP to its matrix representation

In this section we explain how to create the linear system associated to the vector BVP (2.10)-(2.12) using a $\mathbb{P}^1$-Lagrange finite element method. The method is similar to the one used to solve the scalar BVP (see section 6).

7.1 Variational formulation

Let $\boldsymbol{v} = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m) \in \left(\mathrm{H}^1(\Omega)\right)^m$ a test function and $\alpha \in [\![1, m]\!]$. We multiply the α-th equation of (2.10) by $\boldsymbol{v}_\alpha$ then integrate on Ω to obtain

$$\sum_{\beta=1}^{m} \int_\Omega \mathcal{H}_{\alpha,\beta}(\boldsymbol{u}_\beta)\boldsymbol{v}_\alpha d\mathrm{q} = \int_\Omega \boldsymbol{f}_\alpha \boldsymbol{v}_\alpha d\mathrm{q} \tag{7.1}$$

As $\mathcal{H}_{\alpha,\beta}$ is a $\mathcal{L}$-operator, we use (6.7) to obtain $\forall \beta \in [\![1, m]\!]$

$$\int_\Omega \mathcal{H}_{\alpha,\beta}(\boldsymbol{u}_\beta)\boldsymbol{v}_\alpha d\mathrm{q} = \int_\Omega \mathcal{D}_{\mathcal{H}_{\alpha,\beta}}(\boldsymbol{u}_\beta, \boldsymbol{v}_\alpha) d\mathrm{q} - \int_\Gamma \frac{\partial \boldsymbol{u}_\beta}{\partial n_{\mathcal{H}_{\alpha,\beta}}} \boldsymbol{v}_\alpha \, d\sigma, \tag{7.2}$$

where $\mathcal{D}_{\mathcal{H}_{\alpha,\beta}}$ is given by (6.5) with $\mathcal{L} = \mathcal{H}_{\alpha,\beta}$. Then (7.1) becomes

$$\sum_{\beta=1}^{m} \int_\Omega \mathcal{D}_{\mathcal{H}_{\alpha,\beta}}(\boldsymbol{u}_\beta, \boldsymbol{v}_\alpha) d\mathrm{q} - \sum_{\beta=1}^{m} \int_\Gamma \frac{\partial \boldsymbol{u}_\beta}{\partial n_{\mathcal{H}_{\alpha,\beta}}} \boldsymbol{v}_\alpha d\sigma = \int_\Omega \boldsymbol{f}_\alpha \boldsymbol{v}_\alpha d\mathrm{q}. \tag{7.3}$$

Classically to take into account the Dirichlet boundary condition we take $\boldsymbol{v}_\alpha \in H^1_{0,\Gamma^D_\alpha}$, so from (7.3) we obtain

$$\sum_{\beta=1}^{m} \int_\Omega \mathcal{D}_{\mathcal{H}_{\alpha,\beta}}(\boldsymbol{u}_\beta, \boldsymbol{v}_\alpha) d\mathrm{q} - \int_{\Gamma^R_\alpha} \sum_{\beta=1}^{m} \frac{\partial \boldsymbol{u}_\beta}{\partial n_{\mathcal{H}_{\alpha,\beta}}} \boldsymbol{v}_\alpha d\sigma = \int_\Omega \boldsymbol{f}_\alpha \boldsymbol{v}_\alpha d\mathrm{q}.$$

From boundary conditions (2.12), we obtain $\forall \alpha \in [\![1, m]\!]$

$$\sum_{\beta=1}^{m} \int_\Omega \mathcal{D}_{\mathcal{H}_{\alpha,\beta}}(\boldsymbol{u}_\beta, \boldsymbol{v}_\alpha) d\mathrm{q} + \int_{\Gamma^R_\alpha} a^R_\alpha \boldsymbol{u}_\alpha \boldsymbol{v}_\alpha d\sigma = \int_{\Gamma^R_\alpha} g^R_\alpha \boldsymbol{v}_\alpha d\sigma + \int_\Omega \boldsymbol{f}_\alpha \boldsymbol{v}_\alpha d\mathrm{q}. \tag{7.4}$$

Summing these equations on α gives

$$\begin{aligned} \sum_{\alpha=1}^{m} \sum_{\beta=1}^{m} \int_\Omega \mathcal{D}_{\mathcal{H}_{\alpha,\beta}}(\boldsymbol{u}_\beta, \boldsymbol{v}_\alpha) d\mathrm{q} + \sum_{\alpha=1}^{m} \int_{\Gamma^R_\alpha} a^R_\alpha \boldsymbol{u}_\alpha \boldsymbol{v}_\alpha d\sigma \\ = \sum_{\alpha=1}^{m} \int_{\Gamma^R_\alpha} g^R_\alpha \boldsymbol{v}_\alpha d\sigma + \sum_{\alpha=1}^{m} \int_\Omega \boldsymbol{f}_\alpha \boldsymbol{v}_\alpha d\mathrm{q}. \end{aligned} \tag{7.5}$$

We can easily prove that (7.5) is equivalent to (7.4) : a simple sum on α of (7.4) gives (7.5) and, reciprocally, if we choose in (7.5) $\boldsymbol{v}$ such that $\boldsymbol{v}_\beta = 0$, $\forall \beta \in [\![1, m]\!] \backslash \alpha$ then we obtain (7.4).

Then from (7.5), we obtain the variational formulation for the *vector* BVP (2.10)-(2.12) given by

Variational formulation for the *vector* BVP

Find $\boldsymbol{u} \in H^1_{g_1^D,\Gamma_1^D} \times \ldots \times H^1_{g_m^D,\Gamma_m^D}$ such that

$$\boldsymbol{A}_{\mathcal{H}}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{\mathcal{F}}(\boldsymbol{v}) \ \forall \boldsymbol{v} \in H^1_{0,\Gamma_1^D} \times \ldots \times H^1_{0,\Gamma_m^D} \tag{7.6}$$

where

$$\boldsymbol{A}_{\mathcal{H}}(\boldsymbol{u}, \boldsymbol{v}) = \int_\Omega \boldsymbol{\mathcal{D}}_{\mathcal{H}}(\boldsymbol{u}, \boldsymbol{v}) d\mathrm{q} + \sum_{\alpha=1}^{m} \int_{\Gamma^R_\alpha} a^R_\alpha \boldsymbol{u}_\alpha \boldsymbol{v}_\alpha d\sigma \tag{7.7}$$

$$\boldsymbol{\mathcal{F}}(\boldsymbol{v}) = \int_\Omega \langle \boldsymbol{f}, \boldsymbol{v} \rangle d\mathrm{q} + \sum_{\alpha=1}^{m} \int_{\Gamma^R_\alpha} g^R_\alpha \boldsymbol{v}_\alpha d\sigma \tag{7.8}$$

where

$$\boldsymbol{\mathcal{D}}_{\mathcal{H}}(\boldsymbol{u}, \boldsymbol{v}) = \sum_{\alpha=1}^{m} \sum_{\beta=1}^{m} \mathcal{D}_{\mathcal{H}_{\alpha,\beta}}(\boldsymbol{u}_\beta, \boldsymbol{v}_\alpha) \tag{7.9}$$

By an extension theorem, $\forall \alpha \in [\![1, m]\!]$, if $g^D_\alpha \in \mathrm{H}^{1/2}(\Gamma^D_\alpha)$, there exists $R^D_\alpha \in \mathrm{H}^1(\Omega)$ such that $\gamma_{\Gamma^D_\alpha}(R^D_\alpha) = g^D_\alpha$. We set $\boldsymbol{R}^D = (R^D_1, \ldots, R^D_m) \in \left(\mathrm{H}^1(\Omega)\right)^m$ and $\boldsymbol{w} = \boldsymbol{u} - \boldsymbol{R}^D$. Then the previous variational formulation can be written equivalently as

Variational formulation for the *vector* BVP with an extension function

Find $\boldsymbol{w} \in H^1_{0,\Gamma_1^D} \times \ldots \times H^1_{0,\Gamma_m^D}$ such that

$$\boldsymbol{A}_{\mathcal{H}}(\boldsymbol{w},\boldsymbol{v}) = \boldsymbol{\mathcal{F}}(\boldsymbol{v}) - \boldsymbol{A}_{\mathcal{H}}(\boldsymbol{R}^D,\boldsymbol{v}) \ \forall \boldsymbol{v} \in H^1_{0,\Gamma_1^D} \times \ldots \times H^1_{0,\Gamma_m^D} \tag{7.10}$$

7.2 Discrete Variational formulation

Using the notations of Section 5.3, the discretisation of the variational formulation (7.6) is given by

Discrete variational formulation

Find $\boldsymbol{u}_h = (\boldsymbol{u}_{h,1},\ldots,\boldsymbol{u}_{h,m}) \in \mathrm{H}^{1,h}_{g_{h,1}^D,\Gamma_{h,1}^D} \times \ldots \times \mathrm{H}^{1,h}_{g_{h,m}^D,\Gamma_{h,m}^D}$ such that

$$\boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{u}_h,\boldsymbol{v}_h) = \boldsymbol{\mathcal{F}}^h(\boldsymbol{v}_h),\ \forall \boldsymbol{v}_h \in \mathrm{H}^{1,h}_{0,\Gamma_{h,1}^D} \times \ldots \times \mathrm{H}^{1,h}_{0,\Gamma_{h,m}^D} \tag{7.11}$$

where

$$\boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{u}_h,\boldsymbol{v}_h) = \int_{\Omega_h} \boldsymbol{\mathcal{D}}_{\mathcal{H}}(\boldsymbol{u}_h,\boldsymbol{v}_h)d\mathrm{q} + \sum_{\alpha=1}^m \int_{\Gamma_{h,\alpha}^R} a_\alpha^R \boldsymbol{u}_{h,\alpha}\boldsymbol{v}_{h,\alpha} d\sigma \tag{7.12}$$

$$\boldsymbol{\mathcal{F}}^h(\boldsymbol{v}_h) = \int_{\Omega_h} \langle \boldsymbol{f},\boldsymbol{v}_h\rangle d\mathrm{q} + \sum_{\alpha=1}^m \int_{\Gamma_{h,\alpha}^R} g_\alpha^R \boldsymbol{v}_{h,\alpha} d\sigma \tag{7.13}$$

Let $\boldsymbol{R}_h^D = (\boldsymbol{R}_{h,1}^D,\ldots,\boldsymbol{R}_{h,m}^D) \in (\mathrm{H}^{1,h}(\Omega_h))^m$ where, for all $\alpha \in [\![1,m]\!]$, $\boldsymbol{R}_{h,\alpha}^D$ is the discrete extension by zero of g_α^D defined in (6.15). Then, the discretisation of the variational formulation (7.10) is given by

Discrete variational formulation with an extension function

Find $\boldsymbol{w}_h \in \mathrm{H}^{1,h}_{0,\Gamma_{h,1}^D} \times \ldots \times \mathrm{H}^{1,h}_{0,\Gamma_{h,m}^D}$ such that

$$\boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{w}_h,\boldsymbol{v}_h) = \boldsymbol{\mathcal{F}}^h(\boldsymbol{v}_h) - \boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{R}_h^D,\boldsymbol{v}_h),\ \forall \boldsymbol{v}_h \in \mathrm{H}^{1,h}_{0,\Gamma_{h,1}^D} \times \ldots \times \mathrm{H}^{1,h}_{0,\Gamma_{h,m}^D} \tag{7.14}$$

Obviously, we have $\boldsymbol{u}_h = \boldsymbol{w}_h + \boldsymbol{R}_h^D$.

7.3 Matrix representation of the discrete variational formulation : the *vector* case

In this section, we set $\mathrm{n_q} = \mathcal{T}_h.\mathrm{n_q}$, $\mathrm{q}^i = \mathcal{T}_h.\mathrm{q}^i$, $\varphi_i = \mathcal{T}_h.\varphi_i$ and $\mathrm{n_{dof}} = m \times \mathrm{n_q}$.
For all $\alpha \in [\![1,m]\!]$, let $\mathcal{I}_{D,\alpha}$ be the set defined by

$$\mathcal{I}_{D,\alpha} = \left\{ i \in [\![1,\mathrm{n_q}]\!] \mid \mathrm{q}^i \in \overline{\Gamma_{h,\alpha}^D} \right\} \tag{7.15}$$

and $\mathcal{I}_{D,\alpha}^c$ its complement in $[\![1,\mathrm{n_q}]\!]$. We denote by N_α the cardinality of $\mathcal{I}_{D,\alpha}^c$. Using notations of (5.9) and (5.10) we have :

- Let $\boldsymbol{u}_h = (\boldsymbol{u}_{h,1},\ldots,\boldsymbol{u}_{h,m}) \in \mathrm{H}^{1,h}_{g_{h,1}^D,\Gamma_{h,1}^D} \times \ldots \times \mathrm{H}^{1,h}_{g_{h,m}^D,\Gamma_{h,m}^D}$ the solution of (7.11). We denote by $\boldsymbol{U} = (\boldsymbol{u}^1,\ldots,\boldsymbol{u}^m)^t$ the block vector in $\mathbb{R}^{\mathrm{n_{dof}}}$ such that, $\forall \mu \in [\![1,m]\!]$, $\boldsymbol{u}^\mu \in \mathbb{R}^{\mathrm{n_q}}$ with, $\forall i \in [\![1,\mathrm{n_q}]\!]$, $\boldsymbol{u}_i^\mu = \boldsymbol{u}_{h,\mu}(\mathrm{q}^i)$. From (5.9), we can write $\boldsymbol{u}_h$ using $\boldsymbol{U}$:

  $$\boldsymbol{u}_h = \sum_{\mu=1}^m \boldsymbol{u}_{h,\mu}\boldsymbol{e}_\mu = \sum_{\mu=1}^m \left(\sum_{i=1}^{\mathrm{n_q}} \boldsymbol{u}_i^\mu \varphi_i \right) \boldsymbol{e}_\mu$$

  One can also note that

  $$\forall \mu \in [\![1,m]\!],\ \forall i \in \mathcal{I}_{D,\mu},\ \boldsymbol{u}_i^\mu = g_{h,\mu}^D(\mathrm{q}^i)$$

and thus

$$\boldsymbol{u}_{h,\mu} = \sum_{i \in \mathcal{I}^c_{D,\mu}} \boldsymbol{u}^\mu_i \varphi_i + \sum_{i \in \mathcal{I}_{D,\mu}} g^D_{h,\mu}(\mathrm{q}^i)\varphi_i. \tag{7.16}$$

- Let $\boldsymbol{w}_h = (\boldsymbol{w}_{h,1}, \ldots, \boldsymbol{w}_{h,m}) \in \mathrm{H}^{1,h}_{0,\Gamma^D_{h,1}} \times \ldots \times \mathrm{H}^{1,h}_{0,\Gamma^D_{h,m}}$ be the solution of (7.14). We denote by $\boldsymbol{W} = (\boldsymbol{w}^1, \ldots, \boldsymbol{w}^m)^t$ the block vector in $\mathbb{R}^{\mathrm{n_{dof}}}$ such that, $\forall \mu \in [\![1,m]\!]$, $\boldsymbol{w}^\mu \in \mathbb{R}^{\mathrm{n_q}}$ with $\boldsymbol{w}^\mu_i = \boldsymbol{w}_{h,\mu}(\mathrm{q}^i)$, $\forall i \in [\![1,\mathrm{n_q}]\!]$. From (5.9), we can write $\boldsymbol{w}_h$ in function of $\boldsymbol{W}$:

$$\boldsymbol{w}_h = \sum_{\mu=1}^m \boldsymbol{w}_{h,\mu}\boldsymbol{e}_\mu = \sum_{\mu=1}^m \left(\sum_{i=1}^{\mathrm{n_q}} \boldsymbol{w}^\mu_i \varphi_i \right) \boldsymbol{e}_\mu.$$

One can also note that

$$\forall \mu \in [\![1,m]\!],\ \forall i \in \mathcal{I}_{D,\mu},\ \boldsymbol{w}^\mu_i = 0$$

and thus

$$\boldsymbol{w}_{h,\mu} = \sum_{i \in \mathcal{I}^c_{D,\mu}} \boldsymbol{w}^\mu_i \varphi_i. \tag{7.17}$$

We also obtain

$$\mathrm{Span}\left(\mathrm{H}^{1,h}_{0,\Gamma^D_{h,1}} \times \ldots \times \mathrm{H}^{1,h}_{0,\Gamma^D_{h,m}} \right) = \left\{ \varphi_i \boldsymbol{e}_\gamma \mid \gamma \in [\![1,m]\!],\ i \in \mathcal{I}^c_{D,\gamma} \right\} \tag{7.18}$$

- Let $\boldsymbol{R}^D_h = (\boldsymbol{R}^D_{h,1}, \ldots, \boldsymbol{R}^D_{h,m}) \in (\mathrm{H}^{1,h}(\Omega_h))^m$ where, for all $\mu \in [\![1,m]\!]$, $\boldsymbol{R}^D_{h,\mu}$ is the discrete extension by zero of g^D_μ defined in (6.15). We denote by $\boldsymbol{R} = (\boldsymbol{R}^1, \ldots, \boldsymbol{R}^m)^t$ the block vector in $\mathbb{R}^{\mathrm{n_{dof}}}$ such that, $\forall \mu \in [\![1,m]\!]$, $\boldsymbol{R}^\mu \in \mathbb{R}^{\mathrm{n_q}}$ with, for all $j \in [\![1,\mathrm{n_q}]\!]$,

$$\boldsymbol{R}^\mu_j = \boldsymbol{R}_{h,\mu}(\mathrm{q}^j) = \begin{cases} g^D_\mu(\mathrm{q}^j) & \text{if } j \in \mathcal{I}_{D,\mu}, \\ 0 & \text{otherwise} \end{cases}. \tag{7.19}$$

As we have $\boldsymbol{u}_h = \boldsymbol{w}_h + \boldsymbol{R}^D_h$, with these notations, one can write equivalently this equation in the vector form $\boldsymbol{U} = \boldsymbol{W} + \boldsymbol{R}$.

In the discrete variational formulation (7.14) $\boldsymbol{v}_h \in \mathrm{H}^{1,h}_{0,\Gamma^D_{h,1}} \times \ldots \times \mathrm{H}^{1,h}_{0,\Gamma^D_{h,m}}$ so, from (7.18), bilinearity of $\boldsymbol{A}^h_{\mathcal{H}}$ and linearity of $\boldsymbol{\mathcal{F}}^h$, the discrete variational formulations (7.11) and (7.14) are equivalent to

$$\boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{u}_h, \varphi_i \boldsymbol{e}_\gamma) = \boldsymbol{\mathcal{F}}^h(\varphi_i \boldsymbol{e}_\gamma),\ \forall \gamma \in [\![1,m]\!],\ \forall i \in \mathcal{I}^c_{D,\gamma} \tag{7.20}$$

and

$$\boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{w}_h, \varphi_i \boldsymbol{e}_\gamma) = \boldsymbol{\mathcal{F}}^h(\varphi_i \boldsymbol{e}_\gamma) - \boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{R}^D_h, \varphi_i \boldsymbol{e}_\gamma),\ \forall \gamma \in [\![1,m]\!],\ \forall i \in \mathcal{I}^c_{D,\gamma} \tag{7.21}$$

To explicitly write the linear system associated to (7.20) or (7.21), we introduce the $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ block matrix $\mathbb{A}$ partitioned in m^2 blocks defined by

$$\mathbb{A} := \left[\begin{array}{c|c|c} \mathbb{A}^{1,1} & \ldots & \mathbb{A}^{1,m} \\ \hline \vdots & \ddots & \vdots \\ \hline \mathbb{A}^{m,1} & \ldots & \mathbb{A}^{m,m} \end{array} \right] \tag{7.22}$$

where each block $\mathbb{A}^{\gamma,\mu}$ is the $\mathrm{n_q}$-by-$\mathrm{n_q}$ matrix given by

$$\mathbb{A}^{\gamma,\mu}_{i,j} = \boldsymbol{A}^h_{\mathcal{H}}(\varphi_j \boldsymbol{e}_\mu, \varphi_i \boldsymbol{e}_\gamma) \quad \forall (i,j) \in [\![1,\mathrm{n_q}]\!]^2. \tag{7.23}$$

Using (7.12) with (7.9) gives

$$\begin{aligned} \boldsymbol{A}^h_{\mathcal{H}}(\varphi_j \boldsymbol{e}_\mu, \varphi_i \boldsymbol{e}_\gamma) = & \sum_{\alpha=1}^m \sum_{\beta=1}^m \int_{\Omega_h} \mathcal{D}_{\mathcal{H}_{\alpha,\beta}}(\varphi_j \delta_{\mu,\beta}, \varphi_i \delta_{\gamma,\alpha}) d\mathrm{q} \\ & + \sum_{\alpha=1}^m \int_{\Gamma^R_{h,\alpha}} a^R_\alpha \varphi_j \delta_{\mu,\alpha} \varphi_i \delta_{\gamma,\alpha} d\sigma \end{aligned}$$

So we obtain $\forall (i,j) \in [\![1,\mathrm{n_q}]\!]^2$

$$\mathbb{A}^{\gamma,\mu}_{i,j} = \int_{\Omega_h} \mathcal{D}_{\mathcal{H}_{\gamma,\mu}}(\varphi_j, \varphi_i) d\mathrm{q} + \delta_{\mu,\gamma} \int_{\Gamma^R_{h,\gamma}} a^R_\gamma \varphi_j \varphi_i d\sigma \tag{7.24}$$

We also denote by $\boldsymbol{b} = (\boldsymbol{b}^1, \dots, \boldsymbol{b}^m)^t$ the block vector in $\mathbb{R}^{m \times \mathrm{n_q}}$ defined by

$$\boldsymbol{b}_{i+(\gamma-1)\mathrm{n_q}} = \boldsymbol{b}_i^\gamma = \boldsymbol{\mathcal{F}}^h(\varphi_i \boldsymbol{e}_\gamma) \tag{7.25}$$

Using (7.13), we have

$$\boldsymbol{\mathcal{F}}^h(\varphi_i \boldsymbol{e}_\gamma) = \sum_{\alpha=1}^m \int_{\Omega_h} \boldsymbol{f}_\alpha \varphi_i \delta_{\gamma,\alpha} d\mathrm{q} + \sum_{\alpha=1}^m \int_{\Gamma^R_{h,\alpha}} g^R_\alpha \varphi_i \delta_{\gamma,\alpha} d\sigma$$

and we obtain $\forall i \in [\![1, \mathrm{n_q}]\!]$

$$\boldsymbol{b}_i^\gamma = \int_{\Omega_h} \boldsymbol{f}_\gamma \varphi_i d\mathrm{q} + \int_{\Gamma^R_{h,\gamma}} g^R_\gamma \varphi_i d\sigma \tag{7.26}$$

By the bilinearity of $\boldsymbol{A}^h_{\mathcal{H}}$, we have $\forall \gamma \in [\![1, m]\!]$, $\forall i \in [\![1, \mathrm{n_q}]\!]$,

$$\boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{u}_h, \varphi_i \boldsymbol{e}_\gamma) = \sum_{\mu=1}^m \boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{u}_{h,\mu} \boldsymbol{e}_\mu, \varphi_i \boldsymbol{e}_\gamma)$$

and

$$\boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{u}_{h,\mu} \boldsymbol{e}_\mu, \varphi_i \boldsymbol{e}_\gamma) = \sum_{j=1}^{\mathrm{n_q}} \boldsymbol{u}_j^\mu \boldsymbol{A}^h_{\mathcal{H}}(\varphi_j \boldsymbol{e}_\mu, \varphi_i \boldsymbol{e}_\gamma) = \sum_{j=1}^{\mathrm{n_q}} \mathbb{A}^{\gamma,\mu}_{i,j} \boldsymbol{u}_j^\mu$$

So we have,

$$\boldsymbol{A}^h_{\mathcal{H}}(\boldsymbol{u}_h, \varphi_i \boldsymbol{e}_\gamma) = \left(\sum_{\mu=1}^m \mathbb{A}^{\gamma,\mu} \boldsymbol{u}^\mu \right)_i = (\mathbb{A}\boldsymbol{U})_{i+(\gamma-1)\mathrm{n_q}}.$$

Using (7.20), the matrix form of the discrete variational formulation (7.11) is

Matrix form of discrete variational formulation

Find $\boldsymbol{U} \in \mathbb{R}^{\mathrm{n_{dof}}}$ such that $\forall \gamma \in [\![1, m]\!]$

$$(\mathbb{A}\boldsymbol{U})_{i+(\gamma-1)\mathrm{n_q}} = \boldsymbol{b}_{i+(\gamma-1)\mathrm{n_q}}, \qquad \forall i \in \mathcal{I}^c_{D,\gamma} \tag{7.27}$$

$$(\boldsymbol{U})_{i+(\gamma-1)\mathrm{n_q}} = g^D_{h,\gamma}(\mathrm{q}^i), \qquad \forall i \in \mathcal{I}_{D,\gamma}. \tag{7.28}$$

As $\boldsymbol{u}_h = \boldsymbol{w}_h + \boldsymbol{R}^D_h$ and $\boldsymbol{U} = \boldsymbol{W} + \boldsymbol{R}$, using (7.21), the matrix form of the discrete variational formulation (7.14) is

Matrix form of discrete variational formulation with an extension function

Find $\boldsymbol{W} \in \mathbb{R}^{\mathrm{n_{dof}}}$ such that $\forall \gamma \in [\![1, m]\!]$

$$(\mathbb{A}\boldsymbol{W})_{i+(\gamma-1)\mathrm{n_q}} = \boldsymbol{b}_{i+(\gamma-1)\mathrm{n_q}} - (\mathbb{A}\boldsymbol{R})_{i+(\gamma-1)\mathrm{n_q}} \qquad \forall i \in \mathcal{I}^c_{D,\gamma} \tag{7.29}$$

$$(\boldsymbol{W})_{i+(\gamma-1)\mathrm{n_q}} = 0 \qquad \forall i \in \mathcal{I}_{D,\gamma}. \tag{7.30}$$

To solve this problem, we define $\boldsymbol{\mathcal{I}}_D$ as the subset of $[\![1, \mathrm{n_{dof}}]\!]$ such that

$$\boldsymbol{\mathcal{I}}_D = \bigcup_{\alpha=1}^m \{ l + (\alpha - 1)\mathrm{n_q},\ \forall l \in \mathcal{I}_{D,\alpha} \} \tag{7.31}$$

and $\boldsymbol{\mathcal{I}}^c_D$ its complement. We set $N = \#\boldsymbol{\mathcal{I}}^c_D = \sum_{\alpha=1}^m N_\alpha$.

We denote by $\boldsymbol{W}(\boldsymbol{\mathcal{I}}^c_D)$ the restriction of $\boldsymbol{W}$ to $\boldsymbol{\mathcal{I}}^c_D$ indices such that $\boldsymbol{W}(\boldsymbol{\mathcal{I}}^c_D) = (\boldsymbol{w}^1(\mathcal{I}^c_{D,1}), \dots, \boldsymbol{w}^m(\mathcal{I}^c_{D,m}))^t \in \mathbb{R}^N$ and $\mathbb{A}(\boldsymbol{\mathcal{I}}^c_D, \boldsymbol{\mathcal{I}}^c_D)$ the N-by-N block matrix, submatrix of $\mathbb{A}$ obtained by deleting $\boldsymbol{\mathcal{I}}_D$ rows and columns such that

$$\mathbb{A}(\boldsymbol{\mathcal{I}}^c_D, \boldsymbol{\mathcal{I}}^c_D) = \left[\begin{array}{c|c|c} \mathbb{A}^{1,1}(\mathcal{I}^c_{D,1}, \mathcal{I}^c_{D,1}) & \dots & \mathbb{A}^{1,m}(\mathcal{I}^c_{D,1}, \mathcal{I}^c_{D,m}) \\ \hline \vdots & \ddots & \vdots \\ \hline \mathbb{A}^{m,1}(\mathcal{I}^c_{D,m}, \mathcal{I}^c_{D,1}) & \dots & \mathbb{A}^{m,m}(\mathcal{I}^c_{D,m}, \mathcal{I}^c_{D,m}) \end{array}\right]$$

Each block $\mathbf{A}^{\alpha,\beta}(\mathcal{I}^c_{D,\alpha},\mathcal{I}^c_{D,\beta})$ is then a N_α-by-N_β matrix.

From (7.29)-(7.30), we deduce the linear system equivalent to the discrete variational formulation with an extension function (7.14) :

Linear system of discrete variational formulation with an extension function

Find $\boldsymbol{W} \in \mathbb{R}^{\mathrm{n_{dof}}}$ such that

$$\mathbf{A}(\boldsymbol{\mathcal{I}}^c_D,\boldsymbol{\mathcal{I}}^c_D)\,\boldsymbol{W}(\boldsymbol{\mathcal{I}}^c_D) = \boldsymbol{d}(\boldsymbol{\mathcal{I}}^c_D) \tag{7.32}$$

$$\boldsymbol{W}(\boldsymbol{\mathcal{I}}_D) = 0. \tag{7.33}$$

where $\boldsymbol{d} = \boldsymbol{b} - \mathbf{A}\boldsymbol{R} \in \mathbb{R}^{\mathrm{n_{dof}}}$.

We can note that $\boldsymbol{U}(\boldsymbol{\mathcal{I}}_D) = \boldsymbol{R}(\boldsymbol{\mathcal{I}}_D)$ and $\boldsymbol{U}(\boldsymbol{\mathcal{I}}^c_D) = \boldsymbol{W}(\boldsymbol{\mathcal{I}}^c_D)$.

As in the scalar case to ease the computation of $\mathbf{A}$ and $\boldsymbol{b}$, they are split using respectively (7.24) and (7.26). These decompositions come from the contribution without boundary condition and Robin boundary condition contribution.

For the terms without boundary conditions, we set $\mathbb{H}$ as the $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ block matrix where each block $\mathbb{H}^{\gamma,\mu}$ is of order $\mathrm{n_q}$ with $\forall (i,j) \in [\![1,\mathrm{n_q}]\!]^2$

$$\mathbb{H}^{\gamma,\mu}_{i,j} = \int_{\Omega_h} \mathcal{D}_{\mathcal{H}_{\gamma,\beta}}(\varphi_j,\varphi_i)d\mathrm{q} \tag{7.34}$$

$\boldsymbol{b}^f$ is the vector composed of m blocks where each block $\boldsymbol{b}^{f,\gamma}$ is of dimension $\mathrm{n_q}$ with $\forall i \in [\![1,\mathrm{n_q}]\!]$

$$\boldsymbol{b}^{f,\gamma}_i = \int_{\Omega_h} \boldsymbol{f}_\gamma \varphi_i d\mathrm{q}. \tag{7.35}$$

Finally for the terms corresponding to *Robin* boundary condition the block diagonal matrix $\mathbb{B}$ of dimension $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ is first defined with its γ-th diagonal block $\mathbb{B}^\gamma$ given by

$$\mathbb{B} = \mathrm{diag}(\mathbb{B}^1,\ldots,\mathbb{B}^m),\ \text{with}\ \mathbb{B}^\gamma_{i,j} = \int_{\Gamma^R_{h,\gamma}} a^R_\gamma \varphi_j\varphi_i d\sigma,\ \forall (i,j) \in [\![1,\mathcal{T}_h.\mathrm{n_q}]\!]^2 \tag{7.36}$$

where each diagonal block $\mathbb{B}^\gamma$, $\gamma \in [\![1,m]\!]$, is a $\mathcal{T}_h.\mathrm{n_q}$-by-$\mathcal{T}_h.\mathrm{n_q}$ matrix. The block vector $\boldsymbol{b}^R$ is defined as follows

$$\boldsymbol{b}^{R,\gamma}_i = \int_{\Gamma^R_{h,\gamma}} g^R_\gamma \varphi_i d\sigma \tag{7.37}$$

Using all these notations we obtain

$$\mathbf{A} = \mathbb{H} + \mathbb{B} \tag{7.38}$$

$$\boldsymbol{b} = \boldsymbol{b}^f + \boldsymbol{b}^R \tag{7.39}$$

We give in Algorithm 7.1 the successive steps to obtain vector $\boldsymbol{U} = \boldsymbol{W} + \boldsymbol{R}$.

Algorithm 7.1 Steps for solving the *Vector* Boundary Value Problem

1: Assembly of the matrix $\mathbb{H}$
2: Computation of $\boldsymbol{b}^f$
3: Computation of *genererlized Robin* contributions : $\mathbb{B}$ and $\boldsymbol{b}^R$
4: Computation of *Dirichlet* contributions : $\boldsymbol{\mathcal{I}}_D$, $\boldsymbol{\mathcal{I}}^c_D$ and $\boldsymbol{R}$.
5: $\boldsymbol{b} \leftarrow \boldsymbol{b}^f + \boldsymbol{b}^R$
6: $\mathbf{A} \leftarrow \mathbb{H} + \mathbb{B}$
7: $\boldsymbol{b} \leftarrow \boldsymbol{b} - \mathbf{A}\boldsymbol{R}$
8: $\boldsymbol{U}(\boldsymbol{\mathcal{I}}^c_D) \leftarrow$ Solve$(\mathbf{A}(\boldsymbol{\mathcal{I}}^c_D,\boldsymbol{\mathcal{I}}^c_D),\boldsymbol{b}(\boldsymbol{\mathcal{I}}^c_D))$
9: $\boldsymbol{U}(\boldsymbol{\mathcal{I}}_D) \leftarrow \boldsymbol{R}(\boldsymbol{\mathcal{I}_D})$

Lines 1 to 4 in Algorithm 7.1 are now detailed.

7.4 Matrix assembly

In this section we focus on the assembly of $\mathbb{H}$ defined in (7.34). As $\mathcal{H}_{\gamma,\beta}$ is an $\mathcal{L}$-operator we can compute the $\mathrm{n_q}$-by-$\mathrm{n_q}$ matrix $\mathbb{H}^{\gamma,\mu}$ using the DAssemblyP1_base function from Algorithm 6.11 :

$$\mathbb{H}^{\gamma,\mu} \leftarrow \text{DAssemblyP1\_base}(\mathcal{T}_h, \mathcal{H}_{\gamma,\mu})$$

So using the Hoperator structure from section 4.4.2, we obtain the simple Algorithm 7.2 to approximate $\mathbb{H}$.

Algorithm 7.2

Input :

$\mathcal{T}_h$: mesh structure associated to Ω_h (see Section 4.1)

Hop : `Hoperator` structure (see Section 4.4.2)

Output :

$\mathbb{H}$: $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ sparse matrix. $\mathrm{n_{dof}} = \mathrm{H.m} \times \mathcal{T}_h.\mathrm{n_q}$

```
1:  Function ℍ ← HAssemblyP1_base(𝒯_h, Hop)
2:    n_dof ← Hop.m * 𝒯_h.n_q
3:    ℍ ← Sparse(n_dof, n_dof)
4:    for α ← 1 to Hop.m do
5:      ℐ ← [[1, 𝒯_h.n_q]] + (α − 1)𝒯_h.n_q
6:      for β ← 1 to Hop.m do
7:        𝒥 ← [[1, 𝒯_h.n_q]] + (β − 1)𝒯_h.n_q
8:        ℍ(ℐ, 𝒥) ← DAssemblyP1_base(𝒯_h, Hop.H(α, β))
9:      end for
10:   end for
11: end Function
```

As the first line in Algorithm 7.1 is performed using the function HAssemblyP1_base, now we go to the next one : the computation of $\boldsymbol{b}^{\boldsymbol{f}}$ defined by (6.28)

7.5 Computation of the right-hand side $\boldsymbol{b}^{\boldsymbol{f}}$

The block vector $\boldsymbol{b}^{\boldsymbol{f}} \in \mathbb{R}^{\mathrm{n_{dof}}}$ is given in (7.35) where $\boldsymbol{f} = (\boldsymbol{f}_1, \ldots, \boldsymbol{f}_m) \in (L^2(\Omega))^m$ is the source term in the *vector* BVP (2.10). We choose to interpolate $\boldsymbol{f}_\gamma$ by its $\mathbb{P}^1$-Lagrange interpolate $\pi_h(\boldsymbol{f}_\gamma)$ (see (5.11)) and we denote by $\boldsymbol{F}^\gamma \in \mathbb{R}^{\mathrm{n_q}}$ the vector defined by $\boldsymbol{F}^\gamma_i = \boldsymbol{f}_\gamma(\mathrm{q}^i), \forall i \in [\![1, \mathrm{n_q}]\!]$. As in Section 6.5, we obtain

$$\boldsymbol{b}^{f,\gamma}_i = \int_{\Omega_h} \boldsymbol{f}_\gamma \varphi_i d\mathrm{q} \approx \sum_{j=1}^{\mathrm{n_q}} \boldsymbol{F}^\gamma_j \int_{\Omega_h} \varphi_j \varphi_i d\mathrm{q}$$

Using $\mathbb{M}$ as the $\mathrm{n_q}$-by-$\mathrm{n_q}$ Mass matrix on Ω_h defined in Section 6.5, we obtain

$$\boldsymbol{b}^{f,\gamma} \approx \mathbb{M}\boldsymbol{F}^\gamma$$

Thus we modify the function RHS of Algorithm 6.14 to take into account the vector case ($m > 1$). This new version, compatible with the scalar case ($m = 1$), is given in Algorithm 7.3.

Algorithm 7.3 function RHS : $\mathbb{P}^1$-Lagrange approximation of the right-hand side $\boldsymbol{b}^{\boldsymbol{f}}$ given by (7.35)

Input :

$\boldsymbol{f}$: 1-by-m array of functions from Ω to $\mathbb{R}$.
$\forall \alpha \in [\![1, m]\!]$, $\boldsymbol{f}(\alpha)$ is the function $\boldsymbol{f}_\alpha$.

$\mathcal{T}_h$: Ω_h mesh structure (see Section 4)

Output :

$\boldsymbol{b}^{\boldsymbol{f}}$: $\mathbb{R}^{\mathrm{n_{dof}}}$ vector defined in (7.35) with $\mathrm{n_{dof}} = m \times \mathcal{T}_h.\mathrm{n_q}$

Function $\boldsymbol{b}^{\boldsymbol{f}} \leftarrow \mathrm{RHS}(\mathcal{T}_h, \boldsymbol{f})$
  LMass $\leftarrow$ Loperator$(\mathcal{T}_h.\mathrm{d}, \mathbb{O}, \mathbf{0}, \mathbf{0}, 1)$
  $\mathbb{M} \leftarrow$ DAssemblyP1_base$(\mathcal{T}_h$, LMass$)$
  $\boldsymbol{b}^{\boldsymbol{f}} \leftarrow \mathbf{0}_{\mathrm{n_{dof}}}$
  $\mathcal{I} \leftarrow [\![1, \mathcal{T}_h.nq]\!]$
  for $\alpha \leftarrow 1$ **to** m **do**
    $\boldsymbol{b}^{\boldsymbol{f}}(\mathcal{I}) \leftarrow \mathbb{M} * \mathrm{setFdata}(\boldsymbol{f}(\alpha), \mathcal{T}_h)$
    $\mathcal{I} \leftarrow \mathcal{I} + \mathcal{T}_h.nq$
  end for
end Function

The first two lines of Algorithm 7.1 are done. Now we study the three following ones which correspond to the contributions of the different boundary conditions.

7.6 Boundary conditions

Now, as seen in Algorithm 7.1, we must compute the contributions of

- *Generalized Robin* boundary condition : vector $\boldsymbol{b}^R$ and matrix $\mathbb{B}$ respectively given by (7.37) and (7.36),
- *Dirichlet* boundary condition : the sets $\boldsymbol{\mathcal{I}}_D$, $\boldsymbol{\mathcal{I}}_D^{\mathrm{c}}$, defined by (7.31) and the vector $\boldsymbol{R}$ given in section 7.3.

7.6.1 Generalized vector Robin boundary conditions

In the equation (2.12) of the vector BVP, we have imposed the *generalized vector* Robin boundary conditions on Γ_α^R, $\forall \alpha \in [\![1, m]\!]$. We defined the corresponding contributions $\mathbb{B}$ and $\boldsymbol{b}^R$ respectively in (7.36) and (7.37).

We first explain how to compute the block vector $\boldsymbol{b}^R$. In (4.3), we have set $\Gamma_{h,\alpha}^R = \bigcup_{l \in \mathcal{I}_{labels}^{R,\alpha}} \Gamma_h^{labels(l)}$. Let $\alpha \in [\![1, m]\!]$, using notations of Sections 4.2 and 4.3, we set, for $l \in \mathcal{I}_{labels}^{R,\alpha}$, $g^{\alpha,l}$ as the restriction of g_α^R to $\Gamma_h^{labels(l)}$ also given by $g^{\alpha,l} = \mathrm{bclR}(\alpha, l).\mathrm{g}$. Then we have

$$\boldsymbol{b}_i^{R,\alpha} = \sum_{l \in \mathcal{I}_{labels}^{R,\alpha}} \int_{\Gamma_h^{labels(l)}} g^{\alpha,l} \varphi_i d\mathrm{q}, \ \forall i \in [\![1, \mathrm{n_q}]\!].$$

For all l in $\mathcal{I}_{labels}^{R,\alpha}$, we set $\boldsymbol{b}^{\alpha,l} \in \mathbb{R}^{\mathcal{T}_h.\mathrm{n_q}}$ as the vector defined by

$$\boldsymbol{b}_i^{\alpha,l} = \int_{\Gamma_h^{labels(l)}} g^{\alpha,l} \varphi_i d\mathrm{q}.$$

and then

$$\boldsymbol{b}^{R,\alpha} = \sum_{l \in \mathcal{I}_{labels}^{R,\alpha}} \boldsymbol{b}^{\alpha,l}.$$

As in the scalar case described in section 6.6.1, we denote by $\boldsymbol{G}^{\alpha,l} \in \mathbb{R}^{\mathrm{n_{q,}}_l}$ the vector defined by

$$\boldsymbol{G}_r^{\alpha,l} = g^{\alpha,l}(\mathcal{B}_h(l).\mathrm{q}^r), \quad \forall r \in [\![1, \mathrm{n_{q,}}_l]\!]$$

Using the *boundary mass* matrix defined in (6.51), we obtain the $\mathbb{P}^1$-Lagrange interpolation of $\boldsymbol{b}^{\alpha,l}$ given by

$$\left\{\begin{array}{lcl} \boldsymbol{b}^{\alpha,l}_{\mathcal{I}_l} & \approx & \mathbb{M}^l\boldsymbol{G}^{\alpha,l} \\ \boldsymbol{b}^{\alpha,l}_{\mathcal{I}_l^c} & = & 0. \end{array}\right. \Leftrightarrow \left\{\begin{array}{lcl} \boldsymbol{b}_{\mathcal{I}_l+(\alpha-1)\mathrm{n_q}} & \approx & \mathbb{M}^l\boldsymbol{G}^{\alpha,l} \\ \boldsymbol{b}_{\mathcal{I}_l^c+(\alpha-1)\mathrm{n_q}} & = & 0. \end{array}\right. \tag{7.40}$$

We can note that $\mathbb{M}^l$ matrix does not depend on α.

We now explain the computation of the $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ block diagonal matrix $\mathbb{B} = \mathrm{diag}(\mathbb{B}^1, \ldots, \mathbb{B}^m)$. Let $\alpha \in [\![1, m]\!]$, using notations of Sections 4.3 and 4.2, we set, for $l \in \mathcal{I}^{R,\alpha}_{labels}$, $a^{\alpha,l}$ as the restriction of a^R_α to $\Gamma_h^{labels(l)}$ also given by $a^{\alpha,l} = \mathrm{bcl}(\alpha, l).\mathrm{ar}$. Let $\mathbb{B}^{\alpha,l}$ be the $\mathrm{n_q}$-by-$\mathrm{n_q}$ matrix defined by

$$\mathbb{B}^{\alpha,l}_{i,j} = \int_{\Gamma_h^{labels(l)}} a^{\alpha,l}(\mathrm{q}) \times \mathcal{T}_h.\varphi_j(\mathrm{q}) \times \mathcal{T}_h.\varphi_i(\mathrm{q}) d\mathrm{q}, \ \forall (i,j) \in [\![1, \mathrm{n_q}]\!]^2.$$

So we obtain

$$\mathbb{B}^\alpha = \sum_{l \in \mathcal{I}^{R,\alpha}_{labels}} \mathbb{B}^{\alpha,l}$$

Using (6.50), if $i \notin \mathcal{I}_l$ and $j \notin \mathcal{I}_l$ we have $\mathbb{B}^{\alpha,l}_{i,j} = 0$. Otherwise, there exists $(r,s) \in [\![1, \mathrm{n_{q,}}_l]\!]^2$ such that $i = \mathcal{I}_l(r)$ and $j = \mathcal{I}_l(s)$ and we obtain

$$\mathbb{B}^{\alpha,l}_{i,j} = \int_{\Gamma_h^{labels(l)}} a^{\alpha,l}(\mathrm{q}) \times \mathcal{B}_h(l).\varphi_s(\mathrm{q}) \times \mathcal{B}_h(l).\varphi_r(\mathrm{q}) d\mathrm{q}$$

Let $l \in \mathcal{I}^{R,\alpha}_{labels}$ and $\mathbb{W}^{\alpha,l}$ the $\mathrm{n_{q,}}_l$-by-$\mathrm{n_{q,}}_l$ *boundary weighted Mass matrix* defined for all $(r,s) \in [\![1, \mathrm{n_{q,}}_l]\!]^2$ by

$$\mathbb{W}^{\alpha,l}_{r,s} = \int_{\Gamma_h^{labels(l)}} a^{\alpha,l}(\mathrm{q}) \times \mathcal{B}_h(l).\varphi_s(\mathrm{q}) \times \mathcal{B}_h(l).\varphi_r(\mathrm{q}) d\mathrm{q} \tag{7.41}$$

Then we obtain

$$\mathbb{B}^{\alpha,l}_{i,j} = \left\{\begin{array}{ll} 0 & \text{if } i \notin \mathcal{I}_l \text{ and } j \notin \mathcal{I}_l, \\ \mathbb{W}^{\alpha,l}_{r,s} & \text{otherwise, with } i = \mathcal{I}_l(r) \text{ and } j = \mathcal{I}_l(s). \end{array}\right. \tag{7.42}$$

Each matrix $\mathbb{W}^{\alpha,l}$ can be approximated using DAssemblyP1_base function (in Algorithm 6.11) and then a $\mathbb{P}^1$-Lagrange approximation of $\mathbb{B}$ is computed using Algorithm 7.4.

Algorithm 7.4 function RobinBC (vector version)

Input :

pde : a PDE structure.

Output :

$\boldsymbol{b}^R$: vector of dimension $\mathrm{n_{dof}} = \mathrm{pde}.m \times \mathrm{pde}.\mathcal{T}_h.\mathrm{n_q}$

$\mathbb{B}$: $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ matrix (see (7.36))

Function $[\mathbb{B}, \boldsymbol{b}^R] \leftarrow$ RobinBC(pde)
  $m \leftarrow \mathrm{pde}.m$, $\mathrm{n_q} \leftarrow \mathrm{pde}.\mathcal{T}_h.\mathrm{n_q}$, $d \leftarrow \mathrm{pde.d}$
  $\mathrm{n_{dof}} \leftarrow m \times \mathrm{n_q}$,
  $\boldsymbol{b}^R \leftarrow \mathbf{0}_{\mathrm{n_{dof}}}$
  $\mathbb{B} \leftarrow \mathbb{O}_{\mathrm{n_{dof}} \times \mathrm{n_{dof}}}$ ▷ sparse matrix
  LMass $\leftarrow$ Loperator$(d, \mathbb{O}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, 1)$
  for $l \leftarrow 1$ **to** $\mathrm{pde}.\mathrm{n_{lab}}$ **do**
    $\mathbb{M}^l \leftarrow$ DAssemblyP1_base$(\mathrm{pde}.\mathcal{B}_h(l), \mathrm{LMass})$
    for $\alpha \leftarrow 1$ **to** m **do** ▷ Contributions on $\Gamma^N_{h,\alpha}$
      $\mathcal{I}_l^\alpha \leftarrow \mathrm{pde}.\mathcal{B}_h(l).\mathrm{toGlobal} + (\alpha - 1)\mathrm{n_q}$
      if $\mathrm{pde.bclR}(\alpha, l).\mathrm{g} \neq \varnothing$ **then**
        $\boldsymbol{g}^l \leftarrow$ setFdata$(\mathrm{pde.bclR}(\alpha, l).\mathrm{g}, \mathrm{pde}.\mathcal{B}_h(l))$
        $\boldsymbol{b}^R(\mathcal{I}_l^\alpha) \leftarrow \boldsymbol{b}^R(\mathcal{I}_l^\alpha) + \mathbb{M}^{R,l} * \boldsymbol{g}^l$
      end if
      if $\mathrm{pde.bclR}(\alpha, l).\mathrm{ar} \neq \varnothing$ **then**
        LMassR $\leftarrow$ Loperator$(d, \mathbb{O}_{d\times d}, \mathbf{0}_d, \mathbf{0}_d, \mathrm{pde.bcl}(\alpha, l).\mathrm{ar})$
        $\mathbb{W}^{\alpha,l} \leftarrow$ DAssemblyP1_base$(\mathrm{pde}.\mathcal{B}_h(l), \mathrm{LMassR})$
        $\mathbb{B}(\mathcal{I}_l^\alpha, \mathcal{I}_l^\alpha) \leftarrow \mathbb{B}(\mathcal{I}_l^\alpha, \mathcal{I}_l^\alpha) + \mathbb{W}^{\alpha,l}$
      end if
    end for
  end for
end Function

7.6.2 Dirichlet boundary conditions

Finally for the Dirichlet boundary conditions, we modify the scalar version of the function DirichletBC - see Algorithm 6.18 - building the subset $\boldsymbol{\mathcal{I}}_D$ of $[\![1, \mathrm{n_{dof}}]\!]$ defined in (7.31), its complement $\boldsymbol{\mathcal{I}}_D^c$ and the block vector $\boldsymbol{R}$ defined in (7.19). The vector version of this function is given in Algorithm 7.5.

Algorithm 7.5 function DirichletBC (vector version)

Input :
pde : a PDE structure.
Output :
$\boldsymbol{R}^D$: vector of dimension $\mathrm{n_{dof}} = \mathrm{pde}.m \times \mathrm{pde}.\mathcal{T}_h.\mathrm{n_q}$ (see (7.19))
$\boldsymbol{\mathcal{I}}_D$: defined in (7.31)
$\boldsymbol{\mathcal{I}}_D^c$: complement of $\boldsymbol{\mathcal{I}}_D$ in $[\![1, \mathrm{n_{dof}}]\!]$.

Function $[\boldsymbol{\mathcal{I}}_D, \boldsymbol{\mathcal{I}}_D^c, \boldsymbol{R}^D] \leftarrow$ DirichletBC(pde)
  $\mathrm{n_{dof}} \leftarrow \mathrm{pde.m} \times \mathrm{pde.Th.n_q}$
  $\boldsymbol{R}^D \leftarrow \mathbf{0}_{\mathrm{n_{dof}}}$
  $\boldsymbol{\mathcal{I}}_D \leftarrow \varnothing$
  for $l \leftarrow 1$ **to** $\mathrm{pde.n_{lab}}$ **do**
    for $\alpha \leftarrow 1$ **to** pde.m **do**
      if $\mathrm{pde.bclD}(\alpha, l).\mathrm{g} \neq \varnothing$ **then** ▷ Dirichlet boundary condition
        $\mathcal{I} \leftarrow \mathrm{pde}.\mathcal{B}_h(l).\mathrm{toGlobal} + (\alpha - 1)\mathrm{n_q}$
        $\boldsymbol{R}^D(\mathcal{I}) \leftarrow$ setFData$(\mathrm{pde.bcl}(\alpha, l).\mathrm{g}, \mathrm{pde}.\mathcal{B}_h(l))$
        $\boldsymbol{\mathcal{I}}_D \leftarrow \boldsymbol{\mathcal{I}}_D \cup \mathcal{I}$
      end if
    end for
  end for
  $\boldsymbol{\mathcal{I}}_D^c \leftarrow [\![1, \mathrm{n_{dof}}]\!] \backslash \boldsymbol{\mathcal{I}}_D$
end Function

7.7 Construction and solution of the linear system

Using previous functions, we can easily modify the function SolvePDE acting on $\mathcal{L}$-operators from Algorithm 6.19 to obtain the function SolvePDE acting also on $\mathcal{H}$-operators given in Algorithm 7.6.

Algorithm 7.6 function SolvePDE : solution of a *scalar* or *vector* BVP

Input :
pde : a PDE structure.
Output :
$\boldsymbol{U}$: vector of dimension $\mathrm{n_{dof}} = m \times \mathcal{T}_h.\mathrm{n_q}$.

Function $\boldsymbol{U} \leftarrow$ SolvePDE(pde)
  $\mathrm{n_{dof}} \leftarrow \mathrm{pde.m} \times \mathrm{pde}.\mathcal{T}_h.\mathrm{n_q}$
  $\mathbb{A} \leftarrow$ HAssemblyP1_base$(\mathrm{pde}.\mathcal{T}_h, \mathrm{pde.Op})$
  $\boldsymbol{F}^\Omega \leftarrow \mathrm{RHS}(\mathrm{pde}.\mathcal{T}_h, \mathrm{pde}.\boldsymbol{f})$
  $[\mathbb{M}^R, \boldsymbol{F}^R] \leftarrow$ RobinBC(pde)
  $\mathbb{A} \leftarrow \mathbb{A} + \mathbb{M}^R$
  $\boldsymbol{F} \leftarrow \boldsymbol{F}^\Omega + \boldsymbol{F}^R$
  $[\boldsymbol{R}^D, \boldsymbol{I}_D, \boldsymbol{I}_D^c] \leftarrow$ DirichletBC(pde)
  $\boldsymbol{F} \leftarrow \boldsymbol{F} - \mathbb{A} * \boldsymbol{R}^D$
  $\boldsymbol{U} \leftarrow \mathbf{0}_{\mathrm{n_{dof}}}$
  $\boldsymbol{U}(\boldsymbol{I}_D^c) \leftarrow$ Solve$(\mathbb{A}(\boldsymbol{I}_D^c, \boldsymbol{I}_D^c), \boldsymbol{F}(\mathcal{I}_D^c))$
  $\boldsymbol{U}(\boldsymbol{I}_D) \leftarrow \boldsymbol{R}^D(\boldsymbol{I}_D)$
end Function

8 Performances of the classical or simplistic approach

We look at the performance of the algorithms implemented under Matlab[R2014b]/Octave[3.8.1] [2] and Python[3.4.0] [3] comparing with the one of FreeFEM++. Our reference machine is ....

More precisely, for a given BVP and for a given language, the time (in seconds) of construction of the linear system (assembly + right-hand side + boundary conditions - lines 3 to 8 in Algorithm 7.6) and the solution time (lines 9 to 12 in Algorithm 7.6) are given.

We may note that FreeFEM++ does not use an extension technique to take into account Dirichlet boundary conditions but a method enforcing on the Dirichlet rows of the matrix a very high value (e.g. $tgv = 1e+30$) on the diagonal term (which makes negligible the remaining terms of the row) and on the Dirichlet rows of the right-hand side $tgv \times g^D(\mathrm{q}^i)$.

The first result given in Table 1 corresponds to the computational times for the 2D stationary convection-diffusion problem given in Section 3.2.4 for different mesh sizes. The second one, given in Table 2, is for the 3D linear elasticity problem given in Section 3.3.1.

	Matlab		Octave		Python		FreeFEM++	
n_{dof}	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)
2448	23.132	0.022	44.350	0.010	7.643	0.012	0.082	0.002
6456	62.699	0.080	119.240	0.027	20.556	0.030	0.085	0.003
12205	121.701	0.079	229.142	0.056	38.997	0.061	0.162	0.004
25253	267.526	0.256	494.841	0.135	80.877	0.137	0.339	0.009
55996	642.744	0.576	1197.113	0.348	179.201	0.357	0.754	0.023
98712	1434.475	0.868	2555.707	0.725	317.546	0.742	1.351	0.064
118216	1602.288	1.424	3397.917	0.918	379.764	0.942	1.621	0.079

Table 1: Comparison of BVP solving : 2D stationary convection-diffusion problem - Matlab with `base` assembling and `classic` solve, Octave with `base` assembling and `classic` solve, Python with `base` assembling and `classic` solve, FreeFEM++ with `base` assembling and `sparsesolver` solve.

We see the poor performance of the codes for the construction of the linear system. This is mainly due to the current assembly algorithm which is not convenient for the sparse storage format (CSR) in the codes. We may note from Table 1 a *better* Python performance compared with Matlab/Octave because assembly is first made with LIL sparse matrices which are then converted to CSR format.

In the following, we explain how to write better assembly algorithms to obtain computational times similar to the ones of FreeFEM++.

9 Vectorization

The main drawback of the classical approach is the matrix assembly cost. In the following the assembly performance is enhanced while keeping the same code structure.

9.1 Non-vectorized *3d* algorithm : *3d* `OptV1` version

In this part we consider the finite element assembly of a generic $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ sparse matrix $\mathbb{M}$ with its corresponding $(d+1)$-by-$(d+1)$ local matrix $\mathbb{E}_k$ (also denoted by $\mathbb{E}(T_k)$ when referring to a specific element T_k). An element of $\mathbb{E}(T_k)$ is denoted by $e^k_{\alpha,\beta}$. We also define

	Matlab		Octave		Python		FreeFEM++	
n_{dof}	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)
208	20.352	0.018	39.974	0.006	16.447	0.008	0.067	0.023
756	91.260	0.070	182.602	0.046	92.118	0.049	0.134	0.068
1856	251.202	0.301	500.730	0.197	337.883	0.234	0.371	0.251
4961	734.156	1.063	1468.289	1.926	1640.140	1.959	1.180	1.034
10388	1594.150	3.480	3279.344	8.192	6032.708	11.114	2.380	2.982

Table 2: Comparison of BVP solving : 3D linear elasticity problem - Matlab with `base` assembling and `classic` solve, Octave with `base` assembling and `classic` solve, Python with `base` assembling and `classic` solve, FreeFEM++ with `base` assembling and `CG` solve,

the two $(d+1)$-by-$(d+1)$ matrices $\mathbb{I}_k$ and $\mathbb{J}_k$ such that $\forall(\alpha,\beta) \in [\![1, d+1]\!]^2$

$$\mathbb{I}_k(\alpha,\beta) = \mathrm{me}(\alpha,k) \text{ and } \mathbb{J}_k(\alpha,\beta) = \mathrm{me}(\beta,k).$$

In Algorithm 9.1[1], we recall the classical finite element assembly method to calculate $\mathbb{M}$ using the `ElemMat` function which returns $\mathbb{E}_k$, see also Section 6.4.

Algorithm 9.1 Classical assembly - `base` version

$\mathbb{M} \leftarrow \mathbb{O}_{\mathrm{n_{dof}} \times \mathrm{n_{dof}}}$ ▷ Sparse matrix
for $k \leftarrow 1$ **to** $\mathrm{n_{me}}$ **do**
  $\mathbb{E}_k \leftarrow$ `ElemMat`$(\mathrm{vol}(k), \ldots)$
  for $\alpha \leftarrow 1$ **to** $d+1$ **do**
    $i \leftarrow \mathrm{me}(\alpha, k)$
    for $\beta \leftarrow 1$ **to** $d+1$ **do**
      $j \leftarrow \mathrm{me}(\beta, k)$
      $\mathbb{M}_{i,j} \leftarrow \mathbb{M}_{i,j} + \mathbb{E}_k(\alpha,\beta)$
    end for
  end for
end for

In these algorithms the $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ sparse matrix $\mathbb{M}$ is first declared then each contribution of an element T_k is added to the matrix $\mathbb{M}$. The inefficiency of the `base` version is mainly due to repeated element insertions in the sparse structure and to some dynamic reallocation troubles that may also occur. Due to the sparse storage of $\mathbb{M}$ these successive operations are very expensive.

To enhance the performance of the assembly algorithms in vector languages we compute and store all elementary contributions and use them to generate the sparse matrix $\mathbb{M}$. This is the purpose of the first optimized algorithm using *3d* arrays shown in Algorithm 9.2. This optimized version (named *3d* - `OptV1`) is non-vectorized and based on the use of the `sparse` function as follows:

$$\mathtt{M} \leftarrow \textsc{sparse}(\mathtt{Ig}, \mathtt{Jg}, \mathtt{Kg}, \mathtt{m}, \mathtt{n})$$

This command returns a `m` $\times$`n` sparse matrix `M` such that `M`$(\mathtt{Ig}(k), \mathtt{Jg}(k)) \leftarrow \mathtt{Kg}(k)$. The vectors `Ig`, `Jg` and `Kg` have the same length. The zero elements of `Kg` are not taken into account and the elements of `Kg` having the same indices in `Ig` and `Jg` are summed.
Here are examples of the sparse function in vector languages

[1] The functions and operators used in the algorithms are given in Appendix 12.

- Python (*scipy.sparse* module) :
  `M=sparse.<format>_matrix((Kg,(Ig,Jg)),shape=(m,n))`
  where `<format>` is the sparse matrix format (e.g. `csc`, `csr`, `lil`, ...),
- Matlab : `M=sparse(Ig,Jg,Kg,m,n)`, only `csc` format,
- Octave : `M=sparse(Ig,Jg,Kg,m,n)`, only `csc` format,
- Scilab : `M=sparse([Ig,Jg],Kg,[m,n])`, only `row-by-row` format.

In compiled languages, there are some libraries with such functions. For example, in C language one can use the *SuiteSparse* [4] developed by T. Davis. With a Nvidia GPU, one can use the *Thrust* [10] and *Cusp* [9] libraries for vectorization and sparse computations respectively.

The idea is to store all the element matrices $\mathbb{E}_k$ and the global indices $\mathbb{I}_k$, $\mathbb{J}_k$ into tridimensional arrays for creating $\mathbb{M}$ using the SPARSE function.

So to build the *3d* `OptV1` algorithm, we define the n_{me}-by-$(d+1)$-by-$(d+1)$ *3d* arrays $\mathbb{K}_g$, $\mathbb{I}_g$ and $\mathbb{J}_g$ by, $\forall k \in [\![1, n_{me}]\!]$,

$$\mathbb{K}_g(k,:,:) = \mathbb{E}_k, \quad \mathbb{I}_g(k,:,:) = \mathbb{I}_k \text{ and } \mathbb{J}_g(k,:,:) = \mathbb{J}_k. \tag{9.1}$$

We also denote by ElemMatComp the function which returns the (α, β) entry of the matrix $\mathbb{E}_k$. With these notations, we obtain a preliminary version of a *3d* - `OptV1` given in Algorithm 9.2. The operator (:) in the algorithm allows to convert *3d* arrays to *1d* arrays.

Algorithm 9.2 *3d*-`OptV1` algorithm : preliminary version

$\mathbb{I}_g \leftarrow \mathbb{O}_{n_{me}\times(d+1)\times(d+1)}$ ▷ 3d array contiguous in memory
$\mathbb{J}_g \leftarrow \mathbb{O}_{n_{me}\times(d+1)\times(d+1)}$ ▷ 3d array contiguous in memory
$\mathbb{K}_g \leftarrow \mathbb{O}_{n_{me}\times(d+1)\times(d+1)}$ ▷ 3d array contiguous in memory
for $k \leftarrow 1$ **to** n_{me} **do** ▷ Loop over mesh elements
  for $\alpha \leftarrow 1$ **to** $d+1$ **do** ▷ Loop over local nodes
    for $\beta \leftarrow 1$ **to** $d+1$ **do** ▷ Loop over local nodes
      $\mathbb{K}_g(k,\alpha,\beta) \leftarrow$ ElemMatComp$(\alpha, \beta, \text{vol}(k), \ldots)$
      $\mathbb{I}_g(k,\alpha,\beta) \leftarrow \text{me}(\alpha, k)$
      $\mathbb{J}_g(k,\alpha,\beta) \leftarrow \text{me}(\beta, k)$
    end for
  end for
end for
$\mathbb{M} \leftarrow$ SPARSE$(\mathbb{I}_g(:), \mathbb{J}_g(:), \mathbb{K}_g(:), n_{dof}, n_{dof})$

The computations of the 3d-arrays $\mathbb{I}_g$, $\mathbb{J}_g$ and $\mathbb{K}_g$ are represented on Figure 1.

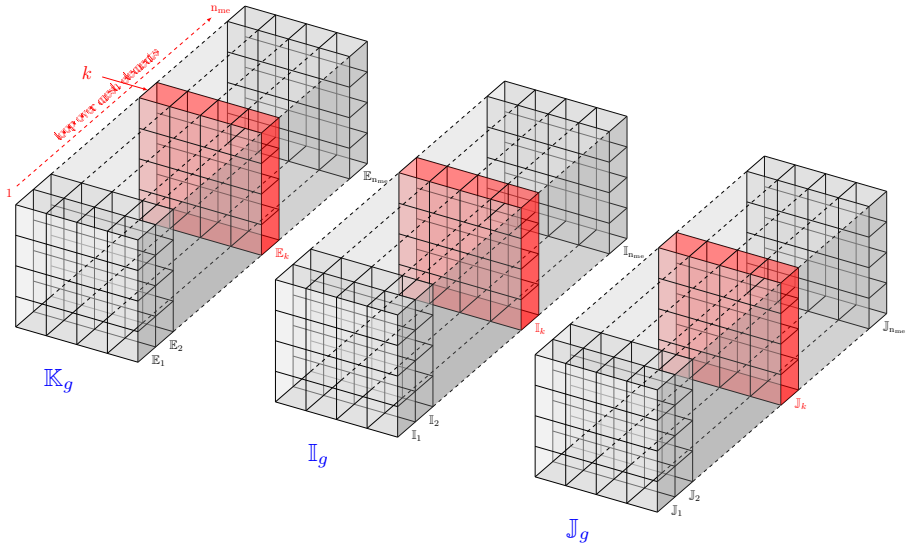


Figure 1: *3d* OptV1 algorithm

In [1] we show that a similar `OptV1` algorithm is more efficient than the classical algorithm. However, the *3d* - `OptV1` algorithm still uses a loop over elements. To improve the efficiency of this algorithm on vector languages, we propose in Section 9.2 a *3d* optimized version, in a fully vectorized form called `OptV3` version.

9.2 *3d* vectorized algorithm : `OptV3` version

In a first time we note in Algorithm 9.2 that it is very easy to permute local loops (α and β) with the loop over mesh elements. This is the goal of Algorithm 9.3. The principle is represented on Figure 2.

Algorithm 9.3 *3d* `OptV1` version : loop interchange

1: $\mathbb{I}_g \leftarrow \mathbb{O}_{\mathrm{n_{me}}\times(d+1)\times(d+1)}$ ▷ 3d array contiguous in memory
2: $\mathbb{J}_g \leftarrow \mathbb{O}_{\mathrm{n_{me}}\times(d+1)\times(d+1)}$ ▷ 3d array contiguous in memory
3: $\mathbb{K}_g \leftarrow \mathbb{O}_{\mathrm{n_{me}}\times(d+1)\times(d+1)}$ ▷ 3d array contiguous in memory
4: **for** $\alpha \leftarrow 1$ **to** $d+1$ **do** ▷ Loop over local nodes
5: **for** $\beta \leftarrow 1$ **to** $d+1$ **do** ▷ Loop over local nodes
6: **for** $k \leftarrow 1$ **to** $\mathrm{n_{me}}$ **do** ▷ Loop over mesh elements
7: $\mathbb{K}_g(k,\alpha,\beta) \leftarrow$ ElemMatComp$(\alpha,\beta,\mathrm{vol}(k),\ldots)$
8: $\mathbb{I}_g(k,\alpha,\beta) \leftarrow \mathrm{me}(\alpha,k)$
9: $\mathbb{J}_g(k,\alpha,\beta) \leftarrow \mathrm{me}(\beta,k)$
10: **end for**
11: **end for**
12: **end for**
13: $\mathbb{M} \leftarrow$ sparse$(\mathbb{I}_g(:),\mathbb{J}_g(:),\mathbb{K}_g(:),\mathrm{n_{dof}},\mathrm{n_{dof}})$

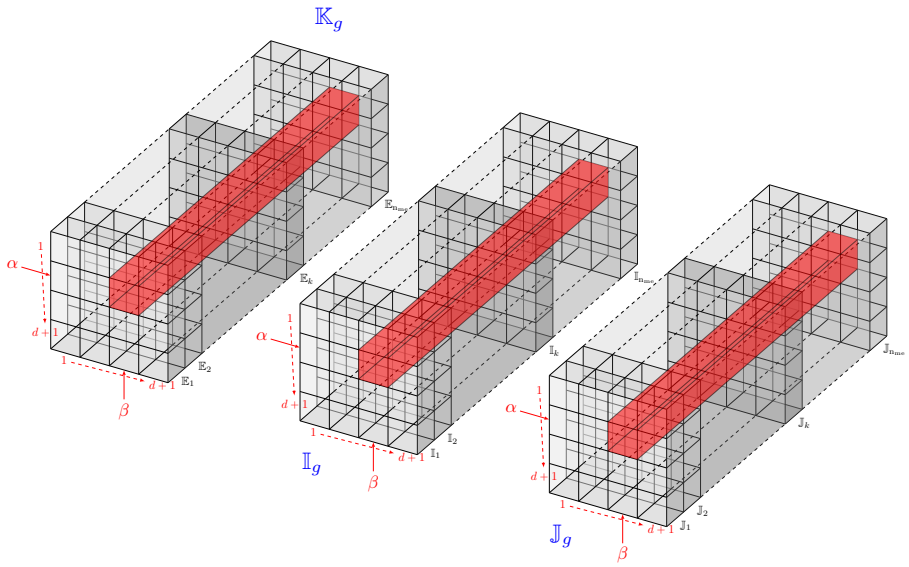


Figure 2: Insertion of (α, β) entry of all element matrices contributions in global arrays

The loop over the mesh elements is to be vectorized. It is easier for the 3d-arrays $\mathbb{I}_g$ and $\mathbb{J}_g$:

$$\mathbb{I}_g(:,\alpha,\beta) \leftarrow \mathrm{me}(\alpha,:) \text{ and } \mathbb{J}_g(:,\alpha,\beta) \leftarrow \mathrm{me}(\beta,:)$$

All the difficulty lies in the vectorization of the computation of the 3d-arrays $\mathbb{K}_g$. We will see that there exists a solution and their vectorization will be described in Section 9.3. For the moment we suppose that function ElemMatVec already exists and allows for a given (α, β) to return the whole contributions of all element matrices. Then Algorithm 9.4 is obtained.

Algorithm 9.4 Generic OptV3 algorithm (memory consuming)

1: $\mathbb{I}_g \leftarrow \mathbb{O}_{\mathrm{n_{me}}\times(d+1)\times(d+1)}$,
2: $\mathbb{J}_g \leftarrow \mathbb{O}_{\mathrm{n_{me}}\times(d+1)\times(d+1)}$
3: $\mathbb{K}_g \leftarrow \mathbb{O}_{\mathrm{n_{me}}\times(d+1)\times(d+1)}$
4: **for** $\alpha \leftarrow 1$ **to** $d+1$ **do** ▷ Loop over local nodes
5: **for** $\beta \leftarrow 1$ **to** $d+1$ **do** ▷ Loop over local nodes
6: $\mathbb{K}_g(:,\alpha,\beta) \leftarrow$ ElemMatVec$(\alpha, \beta, \mathrm{vol}, \ldots)$ ▷ Vectorized element matrix
7: $\mathbb{I}_g(:,\alpha,\beta) \leftarrow \mathrm{me}(\alpha,:)$
8: $\mathbb{J}_g(:,\alpha,\beta) \leftarrow \mathrm{me}(\beta,:)$
9: **end for**
10: **end for**
11: $\mathbb{M} \leftarrow$ sparse$(\mathbb{I}_g(:), \mathbb{J}_g(:), \mathbb{K}_g(:), \mathrm{n_{dof}}, \mathrm{n_{dof}})$

A lower memory consuming algorithm (but mostly less efficient) can be deduced from the previous one by summing $\mathrm{n_{dfe}^2}$ sparse matrices of dimension $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ generated for each α and β. This is the goal of Algorithm 9.5.

Algorithm 9.5 Generic `OptV3` algorithm (less memory consuming)

$\mathbb{M} \leftarrow \mathbb{O}_{\mathrm{n_{dof}} \times \mathrm{n_{dof}}}$ ▷ zeros sparse matrix
for $\alpha \leftarrow 1$ **to** $d+1$ **do** ▷ Loop over local nodes
  for $\beta \leftarrow 1$ **to** $d+1$ **do** ▷ Loop over local nodes
    $\mathbb{M} \leftarrow \mathbb{M} + \textsc{sparse}(\mathrm{me}(\alpha,:), \mathrm{me}(\beta,:), \textsc{ElemMatVec}(\alpha, \beta, \mathrm{vol}, \ldots), \mathrm{n_{dof}}, \mathrm{n_{dof}})$
  end for
end for

Now we apply the technique and adapt it to assemble the matrices associated to operators $\mathcal{L}$ and $\mathcal{H}$.

9.3 OptV3 algorithm for $\mathcal{L}$ operator

As we have just seen the vectorized computation of $\mathbb{I}_g$ and $\mathbb{J}_g$ 3d-arrays is very simple. In Algorithm 9.6 the function IgJgP1_OptV3 is given for computing these two arrays.

Algorithm 9.6 Function IgJgP1_OptV3

Input :
d :
me :
Output :
$\mathbb{I}_g$: $\mathrm{n_{me}}$-by-$(d+1)$-by-$(d+1)$ 3d-array,
$\mathbb{J}_g$: $\mathrm{n_{me}}$-by-$(d+1)$-by-$(d+1)$ 3d-array,

Function $[\mathbb{I}_g, \mathbb{J}_g] \leftarrow$ IgJgP1_OptV3$(\ d, \mathrm{me})$
  for $\alpha \leftarrow 1$ **to** $d+1$ **do**
    for $\beta \leftarrow 1$ **to** $d+1$ **do**
      $\mathbb{I}_g(:,\alpha,\beta) \leftarrow \mathrm{me}(\alpha,:)$
      $\mathbb{J}_g(:,\alpha,\beta) \leftarrow \mathrm{me}(\beta,:)$
    end for
  end for
end Function

The 3d-array $\mathbb{K}_g$ associated to the $\mathcal{L}$ operator is defined by

$$(\mathbb{K}_g)_k = \mathbb{D}^{e,\mathcal{L}}(T_k), \ \forall k \in [\![1, \mathrm{n_{me}}]\!]$$

where $\mathbb{D}^{e,\mathcal{L}}$ is the $(d+1)$-by-$(d+1)$ matrix defined in (6.35). To vectorize the computation of $\mathbb{K}_g$ we use the splitting of $\mathcal{D}_{\mathcal{L}}$ operator seen in Section 6.4.1. Let f be a function defined in Ω, $(i,j) \in [\![1,d]\!]$ and $\mathbb{K}_{uv}(f)$, $\mathbb{K}_{dudv}(f,i,j)$, $\mathbb{K}_{udv}(f,i)$ and $\mathbb{K}_{duv}(f,i)$ the $\mathrm{n_{me}} \times (d+1) \times (d+1)$ 3d-arrays defined with tensor notation and for all $k \in [\![1, \mathrm{n_{me}}]\!]$ by

$$(\mathbb{K}_{uv}(f))_k = \mathbb{D}^e_{uv}(T_k, f), \qquad (\mathbb{K}_{dudv}(f,i,j))_k = \mathbb{D}^e_{dudv}(T_k, f, i, j), \tag{9.2}$$

$$(\mathbb{K}_{udv}(f))_k = \mathbb{D}^e_{udv}(T_k, f, i), \qquad (\mathbb{K}_{duv}(f,i))_k = \mathbb{D}^e_{duv}(T_k, f, i). \tag{9.3}$$

Then using 6.41, we obtain

$$(\mathbb{K}_g)_k = \sum_{i=1}^{d}\sum_{j=1}^{d} (\mathbb{K}_{dudv}(\mathbb{A}_{i,j}, i, j))_k - \sum_{i=1}^{d} (\mathbb{K}_{udv}(\boldsymbol{b}_i, i))_k + \sum_{i=1}^{d} (\mathbb{K}_{duv}(\boldsymbol{c}_i, i))_k + (\mathbb{K}_{uv}(a_0))_k$$

and so

$$\mathbb{K}_g = \sum_{i=1}^{d}\sum_{j=1}^{d} \mathbb{K}_{dudv}(\mathbb{A}_{i,j}, i, j) - \sum_{i=1}^{d} \mathbb{K}_{udv}(\boldsymbol{b}_i, i) + \sum_{i=1}^{d} \mathbb{K}_{duv}(\boldsymbol{c}_i, i) + \mathbb{K}_{uv}(a_0) \tag{9.4}$$

We obtain the Algorithm 9.7 in which the functions KgP1_OptV3_guv, KgP1_OptV3_gdudv, KgP1_OptV3_gduv and KgP1_OptV3_gudv need to be described. This is the object of the following sections.

Algorithm 9.7 Function KgP1_OptV3

Input :
$\mathcal{T}_h$: mesh structure associated to Ω_h (see section 4)
$\mathcal{L}$: `Loperator` structure (see section 4)
$\mathbb{G}$: gradients array ($\mathrm{n_{me}}$-by-$(d+1)$-by-d)
$\mathbb{G}(k,\alpha,:) = \nabla\varphi^k_\alpha(\mathrm{q})$, $\forall\alpha\in[\![1,d+1]\!]$
Output :
$\mathbb{K}_g$: $\mathrm{n_{me}}$-by-$(d+1)$-by-$(d+1)$ 3d-array

Function $\mathbb{K}_g \leftarrow$ KgP1_OptV3($\mathcal{T}_h, \mathcal{L}, \mathbb{G}$)
  $\mathbb{K}_g \leftarrow$ KgP1_OptV3_guv($\mathcal{T}_h, \mathcal{L}.a0$)
  for $i \leftarrow 1$ **to** $\mathcal{T}_h.d$ **do**
    for $j \leftarrow 1$ **to** $\mathcal{T}_h.d$ **do**
      $\mathbb{K}_g \leftarrow \mathbb{K}_g +$ KgP1_OptV3_gdudv($\mathcal{T}_h, \mathcal{L}.A(i,j), \mathbb{G}, i, j$)
    end for
    $\mathbb{K}_g \leftarrow \mathbb{K}_g -$ KgP1_OptV3_gudv($\mathcal{T}_h, \mathcal{L}.b(i), \mathbb{G}, i$)
    $\mathbb{K}_g \leftarrow \mathbb{K}_g +$ KgP1_OptV3_gduv($\mathcal{T}_h, \mathcal{L}.c(i), \mathbb{G}, i$)
  end for
end Function

Then we define the vectorized function GradientVec, given in Algorithm 11.3. It returns the 3d-array $\mathbb{G}$ of dimension $\mathrm{n_{me}}$-by-$(d+1)$-by-d containing the gradients of all $\mathbb{P}^1$-Lagrange local basis functions. Using the previous functions the Algorithm 9.4 is converted to the Algorithm 9.8.

Algorithm 9.8 Function DAssemblyP1_OptV3

Input :
$\mathcal{T}_h$: mesh structure associated to Ω_h (see section 4)
$\mathcal{L}$: `Loperator` structure (see section 4)
$\mathbb{G}$: 3d-array given by function GradientsVec or empty (useful for $\mathcal{H}$ operator)
Output :
$\mathbb{D}^{\mathcal{L}}$: $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ sparse matrix with $\mathrm{n_{dof}} = \mathcal{T}_\mathrm{h}.\mathrm{nq}$.

Function $\mathbb{D} \leftarrow$ DAssemblyP1_OptV3($\mathcal{T}_h, \mathcal{L}, \mathbb{G}$)
  $[\mathbb{I}_g, \mathbb{J}_g] \leftarrow$ IgJgP1_OptV3($d, \mathcal{T}_h.\mathrm{n_{me}}, \mathcal{T}_h.\mathrm{me}$)
  if $\mathbb{G} = \varnothing$ **then**
    $\mathbb{G} \leftarrow$ GradientsVec($\mathcal{T}_h.\mathrm{q}, \mathcal{T}_h.\mathrm{me}, \mathcal{T}_h.\mathrm{vols}$)
  end if
  $\mathbb{K}_g \leftarrow$ KgP1_OptV3($\mathcal{T}_h, \mathcal{L}, \mathbb{G}$)
  $\mathbb{D}^{\mathcal{L}} \leftarrow$ Sparse($\mathbb{I}_g(:), \mathbb{J}_g(:), \mathbb{K}_g(:), \mathrm{n_{dof}}, \mathrm{n_{dof}}$)
end Function

To end this section, it only remains to vectorize the splitting functions of the operator.

Let f be a function from Ω to $\mathbb{R}$. In the following sections, we denote by $\boldsymbol{f}$ the 1-by-$\mathrm{n_q}$ array containing function f values on mesh vertices such that $\boldsymbol{f}(i) = f(\mathrm{q}^i)$. We denote by $\mathbb{f}$ the $(d+1)$-by-$\mathrm{n_{me}}$ array such that $\mathbb{f}_{\mathrm{me}}(\alpha,k) = \boldsymbol{f}(\mathrm{me}(\alpha,k))$ and $\boldsymbol{f}^s$ the 1-by-$\mathrm{n_{me}}$ array such that $\boldsymbol{f}^s(k) = \sum_{\alpha=1}^{d+1} \mathbb{f}_{\mathrm{me}}(\alpha,k)$. We can easily compute these arrays in a vectorized form :

$$\mathbb{f}_{\mathrm{me}} \leftarrow \boldsymbol{f}(\mathrm{me}), \quad \boldsymbol{f}^s \leftarrow \mathrm{sum}(\mathbb{f}_{\mathrm{me}}, 1). \tag{9.5}$$

We also denote by $\mathbb{G}$ the $\mathrm{n_{me}}$-by-$(d+1)$-by-d 3d-array defined, $\forall k\in[\![1,\mathrm{n_{me}}]\!]$, $\forall\alpha\in[\![1,d+1]\!]$, by

$$\mathbb{G}(k,\alpha,:) = \nabla\varphi_{\mathrm{me}(\alpha,k)}, \text{ on } T_k. \tag{9.6}$$

9.3.1 Vectorized calculation of $\mathbb{K}_{uv}(f)$

From Section 6.4.2, Formula (6.43) and from notations (9.5), we have on a mesh element T_k and $\forall(\alpha,\beta) \in [\![1, d+1]\!]^2$

$$(\mathbb{D}^e_{uv}(T_k, f))_{\alpha,\beta} \approx \frac{d!|T_k|}{(d+3)!}(1+\delta_{\alpha,\beta})(\boldsymbol{f}^s(k) + \mathbb{f}_{\mathrm{me}}(\alpha,k) + \mathbb{f}_{\mathrm{me}}(\beta,k)).$$

From (9.2), we have

$$(\mathbb{K}_{uv}(f))_{k,\alpha,\beta} = (\mathbb{D}^e_{uv}(T_k, f))_{\alpha,\beta}\,, \qquad \forall k \in [\![1, \mathrm{n_{me}}]\!],\ \forall(\alpha,\beta) \in [\![1, d+1]\!]^2.$$

We represent in Figure 3 the *3d* `OptV1` and `OptV3` versions for the computation of $\mathbb{K}_g$.

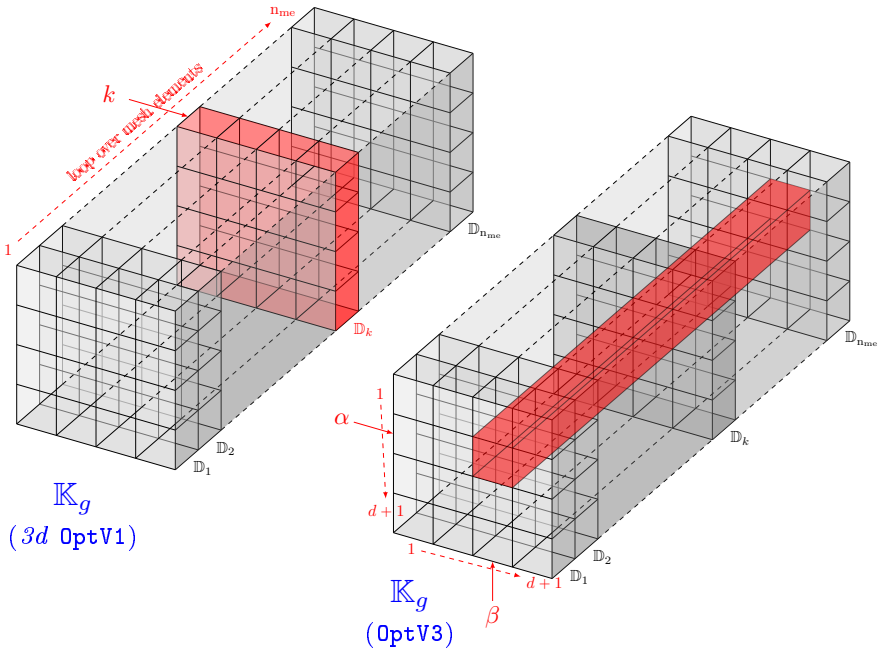


Figure 3: Left : classical insertion of a local matrix in the 3d-array $\mathbb{K}_g$. Right : `OptV3` version

We obtain the vectorized computation of $(\mathbb{K}_{uv}(f))_{:,\beta,\alpha}$:

$$(\mathbb{K}_{uv}(f))_{:,\alpha,\beta} \approx \frac{d!(1+(\alpha==\beta))}{(d+3)!} \times \mathcal{T}_h.\mathrm{vol} \times (\boldsymbol{f}^s + \mathbb{f}_{me}(\alpha,:) + \mathbb{f}_{me}(\beta,:))$$

and we write the complete function KgP1_OptV3_guv in Algorithm 9.9.

Algorithm 9.9 Function KgP1_OptV3_guv : computation of $\mathbb{K}_{uv}(f)$

Input :
$\mathcal{T}_h$: mesh structure associated to Ω_h (see section 4.1)
f : function from Ω to $\mathbb{R}$.
Output :
$\mathbb{K}$: $(d+1)$-by-$(d+1)$-by-$\mathrm{n_{me}}$ array corresponding to $\mathbb{K}_{uv}(f)$

Function $\mathbb{K} \leftarrow$ KgP1_OptV3_guv$(\mathcal{T}_h, f)$
  $\boldsymbol{f} \leftarrow f(\mathcal{T}_h.q)$ ▷ $\boldsymbol{f}$: 1-by-$\mathrm{n_q}$ array
  $\mathbb{f}_{\mathrm{me}} \leftarrow \boldsymbol{f}(\mathcal{T}_h.\mathrm{me})$ ▷ $\mathbb{f}_{\mathrm{me}}$: $(d+1)$-by-$\mathrm{n_{me}}$ array
  $\boldsymbol{f}^s \leftarrow$ sum$(\mathbb{f}_{\mathrm{me}}, 1)$ ▷ $\boldsymbol{f}^s$: 1-by-$\mathrm{n_{me}}$ array
  for $\alpha \leftarrow 1$ **to** $d+1$ **do**
    for $\beta \leftarrow 1$ **to** $d+1$ **do**
      $\mathbb{K}(:,\alpha,\beta) \leftarrow \dfrac{d!(1+(\alpha==\beta))}{(d+3)!} \times \mathcal{T}_h.\mathrm{vol} \times (\boldsymbol{f}^s + \mathbb{f}_{me}(\alpha,:) + \mathbb{f}_{me}(\beta,:))$
    end for
  end for
end Function

9.3.2 Vectorized calculation of $\mathbb{K}_{dudv}(f,i,j)$

From Section 6.4.3, Formula (6.44), we have on a mesh element T_k and $\forall(\alpha,\beta) \in [\![1,d+1]\!]^2$

$$(\mathbb{D}^e_{dudv}(T_k,g,i,j))_{\alpha,\beta} \approx \frac{d!|T_k|}{(d+1)!} \boldsymbol{f}^s(k) \times \mathbb{G}(k,\alpha,i) \times \mathbb{G}(k,\beta,j)$$

From (9.2), we have

$$(\mathbb{K}_{dudv}(f,i,j))_{k,\alpha,\beta} = (\mathbb{D}^e_{dudv}(T_k,f,i,j))_{\alpha,\beta}\,, \qquad \forall k \in [\![1,\mathrm{n_{me}}]\!],\ \forall(\alpha,\beta) \in [\![1,d+1]\!]^2.$$

One can vectorize this formula by

$$(\mathbb{K}_{dudv}(f,i,j))_{:,\alpha,\beta} \approx \frac{d!}{(d+1)!} \times \mathcal{T}_h.\mathrm{vols} \times \mathbb{G}(:,\alpha,i) \times \mathbb{G}(:,\beta,j) \times \boldsymbol{f}^s \tag{9.7}$$

and we write the complete function KgP1_OptV3_gdudv given in Algorithm 9.10.

Algorithm 9.10 Function KgP1_OptV3_gdudv : : computation of $\mathbb{K}_{dudv}(f,i,j)$

Input :
$\mathcal{T}_h$: mesh structure associated to Ω_h (see section 4.1)
f : function from Ω to $\mathbb{R}$.
$\mathbb{G}$: gradients arrays ($\mathrm{n_{me}}$-by-$(d+1)$-by-d)
  $\mathbb{G}(k,\alpha,:) = \nabla\varphi_{\mathrm{me}(\alpha,k)}(\mathrm{q})$, $\forall \mathrm{q} \in T_k$, $\forall\alpha \in [\![1,d+1]\!]$
i : integer in $[\![1,d]\!]$
j : integer in $[\![1,d]\!]$
Output :
$\mathbb{K}$: $(d+1)$-by-$(d+1)$-by-$\mathrm{n_{me}}$ array corresponding to $\mathbb{K}_{dudv}(f,i,j)$.

Function $\mathbb{K} \leftarrow$ KgP1_OptV3_gdudv$(\mathcal{T}_h, g, \mathbb{G}, i, j)$
  $\boldsymbol{f} \leftarrow f(\mathcal{T}_h.q)$ ▷ $\boldsymbol{f}$: 1-by-$\mathrm{n_q}$ array
  $\mathbb{f}_{\mathrm{me}} \leftarrow \boldsymbol{f}(\mathcal{T}_h.\mathrm{me})$ ▷ $\mathbb{f}_{\mathrm{me}}$: $(d+1)$-by-$\mathrm{n_{me}}$ array
  $\boldsymbol{f}^s \leftarrow$ sum$(\mathbb{f}_{\mathrm{me}}, 1)$ ▷ $\boldsymbol{f}^s$: 1-by-$\mathrm{n_{me}}$ array
  for $\alpha \leftarrow 1$ **to** $d+1$ **do**
    for $\beta \leftarrow 1$ **to** $d+1$ **do**
      $\mathbb{K}(:,\alpha,\beta) \leftarrow \dfrac{d!}{(d+1)!} \times \mathcal{T}_h.\mathrm{vols} \times \mathbb{G}(:,\alpha,i) \times \mathbb{G}(:,\beta,j) \times \boldsymbol{f}^s$
    end for
  end for
end Function

9.3.3 Vectorized calculation of $\mathbb{K}_{duv}(f,i)$

From Section 6.4.4, formula (6.45), we have on a mesh element T_k $\forall(\alpha,\beta)\in[\![1,d+1]\!]^2$

$$(\mathbb{D}^e_{duv}(T_k,g,i))_{\alpha,\beta}\approx\frac{d!|T_k|}{(d+2)!}(\boldsymbol{f}^s(k)+\mathbb{f}_{me}(\alpha,k))\times\mathbb{G}(k,\beta,i)$$

From (9.3) we have

$$(\mathbb{K}_{duv}(f,i))_{k,\alpha,\beta}=(\mathbb{D}^e_{duv}(T_k,f,i))_{\alpha,\beta}\,,\qquad\forall k\in[\![1,\mathrm{n_{me}}]\!],\ \forall(\alpha,\beta)\in[\![1,d+1]\!]^2.$$

One can vectorize this formula by

$$(\mathbb{K}_{duv}(f,i))_{:,\alpha,\beta}\approx\frac{d!}{(d+2)!}\times\mathcal{T}_h.\mathrm{vols}\times\mathbb{G}(:,\beta,i)\times(\boldsymbol{f}^s+\mathbb{f}_{me}(\alpha,:)) \tag{9.8}$$

and we write the complete function KgP1_OptV3_gduv given in Algorithm 9.11.

Algorithm 9.11 Function KgP1_OptV3_gduv : add computation of $\mathcal{D}(u,v)=f\frac{\partial u}{\partial x_i}v$ to $\mathbb{K}_g$ 3d array

Input :
- $\mathcal{T}_h$: mesh structure associated to Ω_h (see section 4)
- f : function
- $\mathbb{G}$: gradients arrays ($\mathrm{n_{me}}$-by-$(d+1)$-by-d) $\mathbb{G}(k,\alpha,:)=\nabla\varphi^k_\alpha(\mathrm{q})$, $\forall\alpha\in[\![1,d+1]\!]$
- i : integer in $[\![1,d]\!]$

Output :
- $\mathbb{K}_g$: $d+1$-by-$(d+1)$-by-$\mathrm{n_{me}}$ array.

Function $\mathbb{K}_g\leftarrow$ KgP1_OptV3_gduv($\mathcal{T}_h,f,\mathbb{G},i$)
  $\boldsymbol{f}_h\leftarrow f(\mathcal{T}_h.q)$, $\mathbb{f}_{\mathrm{me}}\leftarrow\boldsymbol{f}_h(\mathcal{T}_h.\mathrm{me})$
  $\boldsymbol{f}^s\leftarrow$ sum$(\mathbb{f}_{\mathrm{me}},1)$
  for $\alpha\leftarrow1$ **to** $d+1$ **do**
    for $\beta\leftarrow1$ **to** $d+1$ **do**
      $\mathbb{K}_g(:,\alpha,\beta)\leftarrow\frac{d!}{(d+2)!}\times\mathcal{T}_h.\mathrm{vols}\times\boldsymbol{G}(:,\beta,i)\times(\boldsymbol{f}^s+\mathbb{f}_{\mathrm{me}}(\alpha,:))$
    end for
  end for
end Function

9.3.4 Vectorized calculation of $\mathbb{K}_{udv}(f,i)$

From Section 6.4.5, Formula (6.46), we have on a mesh element T_k and $\forall(\alpha,\beta)\in[\![1,d+1]\!]^2$

$$(\mathbb{D}^e_{udv}(T_k,f,i))_{\alpha,\beta}\approx\frac{d!|T_k|}{(d+2)!}(\boldsymbol{f}^s(k)+\mathbb{f}_{me}(\beta,k))\times\mathbb{G}(k,\alpha,i)$$

From (9.3), we have

$$(\mathbb{K}_{duv}(f,i))_{k,\alpha,\beta}=(\mathbb{D}^e_{udv}(T_k,f,i))_{\alpha,\beta}\,,\qquad\forall k\in[\![1,\mathrm{n_{me}}]\!],\ \forall(\alpha,\beta)\in[\![1,d+1]\!]^2.$$

One can vectorize this formula by

$$(\mathbb{K}_{udv}(f,i))_{:,\alpha,\beta}\approx\frac{d!}{(d+2)!}\times\mathcal{T}_h.\mathrm{vols}\times\mathbb{G}(:,\alpha,i)\times(\boldsymbol{f}^s+\mathbb{f}_{me}(\beta,:)) \tag{9.9}$$

and we write the complete function KgP1_OptV3_gudv given in Algorithm 9.12.

Algorithm 9.12 Function KgP1_OptV3_gudv : add computation of $\mathcal{D}(u,v) = fu\frac{\partial v}{\partial x_i}$ to $\mathbb{K}_g$ 3d array

Input :

$\mathcal{T}_h$	:	mesh structure associated to Ω_h (see section 4)
f	:	function
$\mathbb{G}$	:	gradients arrays ($\mathrm{n_{me}}$-by-$(d+1)$-by-d) $\mathbb{G}(k,\alpha,:) = \nabla\varphi_\alpha^k(\mathrm{q})$, $\forall\alpha\in[\![1,d+1]\!]$
i	:	integer in $[\![1,d]\!]$

Output :

$\mathbb{K}_g$	:	$(d+1)$-by-$(d+1)$-by-$\mathrm{n_{me}}$ array.

1: **Function** $\mathbb{K}_g \leftarrow$ KgP1_OptV3_gudv($\mathcal{T}_h, f, \mathbb{G}, i$)
2: $\boldsymbol{f}_h \leftarrow f(\mathcal{T}_h.q)$, $\mathbb{f}_{\mathrm{me}} \leftarrow \boldsymbol{f}_h(\mathcal{T}_h.\mathrm{me})$
3: $\boldsymbol{f}^s \leftarrow \mathrm{sum}(\mathbb{f}_{\mathrm{me}}, 1)$
4: **for** $\alpha \leftarrow 1$ **to** $d+1$ **do**
5: **for** $\beta \leftarrow 1$ **to** $d+1$ **do**
6: $\mathbb{K}_g(:,\alpha,\beta) \leftarrow \frac{d!}{(d+2)!} \times \mathcal{T}_h.\mathrm{vols} \times \boldsymbol{G}(:,\alpha,i) \times (\boldsymbol{f}^s + \mathbb{f}_{\mathrm{me}}(\beta,:))$
7: **end for**
8: **end for**
9: **end Function**

9.4 OptV3 algorithm for $\mathcal{H}$ operator

We just have to replace in HAssemblyP1_base (see Algorithm 7.2) the call to DAssemblyP1_base by a call to DAssemblyP1_OptV3 to obtain the function HAssemblyP1_OptV3 given in Algorithm 9.13.

Algorithm 9.13 Function HAssemblyP1_OptV3 - Matrix $\mathbb{H}$ assembly

Input :

$\mathcal{T}_h$	:	mesh structure associated to Ω_h (see section 4)
Hop	:	`Hoperator` structure (see section 4)

Output :

$\mathbb{H}$	:	$\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ sparse matrix. $\mathrm{n_{dof}} = \mathrm{H.m} \times \mathcal{T}_h.\mathrm{n_q}$

1: **Function** $\mathbb{H} \leftarrow$ HAssemblyP1_OptV3($\mathcal{T}_h$, Hop)
2: $\mathrm{n_{dof}} \leftarrow \mathrm{Hop.m} * \mathcal{T}_h.\mathrm{n_q}$
3: $\mathbb{H} \leftarrow$ Sparse($\mathrm{n_{dof}}, \mathrm{n_{dof}}$)
4: $\mathbb{G} \leftarrow$ GradientsVec($\mathcal{T}_h$.q, $\mathcal{T}_h$.me, $\mathcal{T}_h$.vols)
5: **for** $\alpha \leftarrow 1$ **to** Hop.m **do**
6: $\mathcal{I} \leftarrow [\![1, \mathcal{T}_h.\mathrm{n_q}]\!] + (\alpha-1)\mathcal{T}_h.\mathrm{n_q}$
7: **for** $\beta \leftarrow 1$ **to** Hop.m **do** ▷ Set block matrix (α,β)
8: $\mathcal{J} \leftarrow [\![1, \mathcal{T}_h.\mathrm{n_q}]\!] + (\beta-1)\mathcal{T}_h.\mathrm{n_q}$
9: $\mathbb{H}(\mathcal{I},\mathcal{J}) \leftarrow$ DAssemblyP1_OptV3($\mathcal{T}_h$, Hop.H(α,β))
10: **end for**
11: **end for**
12: **end Function**

We propose in Algorithm 9.14, a modified version less memory consuming.

Algorithm 9.14 Function HAssemblyP1_OptV3 - Matrix $\mathbb{H}$ assembly *(less memory consuming version)*

Input :

$\mathcal{T}_h$: mesh structure associated to Ω_h (see section 4)

Hop : `Hoperator` structure (see section 4)

Output :

$\mathbb{H}$: $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ sparse matrix. $\mathrm{n_{dof}} = \mathrm{Hop.m} \times \mathcal{T}_h.\mathrm{n_q}$

Function $\mathbb{H} \leftarrow$ HAssemblyP1_OptV3$(\mathcal{T}_h, \mathrm{Hop})$
  $\mathrm{n_{dof}} \leftarrow \mathrm{Hop.m} * \mathcal{T}_h.\mathrm{n_q}$
  $\mathbb{H} \leftarrow$ Sparse$(\mathrm{n_{dof}}, \mathrm{n_{dof}})$
  $\mathbb{G} \leftarrow$ GradientsVec$(\mathcal{T}_h.\mathrm{q}, \mathcal{T}_h.\mathrm{me}, \mathcal{T}_h.\mathrm{vols})$
  $[\mathbb{I}_g, \mathbb{J}_g] \leftarrow$ IgJgP1_OptV3$(d, \mathrm{n_{me}}, \mathrm{me})$
  for $\alpha \leftarrow 1$ **to** Hop.m **do**
    $\mathbb{I} \leftarrow \mathbb{I}_g + (\alpha - 1)\mathcal{T}_h.\mathrm{n_q}$
    for $\beta \leftarrow 1$ **to** Hop.m **do**
      $\mathbb{J} \leftarrow \mathbb{J}_g + (\beta - 1)\mathcal{T}_h.\mathrm{n_q}$
      $\mathbb{K} \leftarrow$ KgP1_OptV3$(\mathcal{T}_h, \mathrm{Hop.H}(\alpha, \beta), \mathbb{G})$
      $\mathbb{H} \leftarrow \mathbb{H} +$ Sparse$(\mathbb{I}(:), \mathbb{J}(:), \mathbb{K}(:), \mathrm{n_{dof}}, \mathrm{n_{dof}})$
    end for
  end for
end Function

10 Algorithm performance

The different benchmarks have been achieved on our reference machine gpucreos1.

Techniques and algorithms presented in this report have been implemented under Matlab/Octave and Python and are referenced under the name *vecFEMP1Light*.

10.1 Comparison with other assembly codes

Here we compare `vecFEMP1` codes with those given in [1] and referenced under the name `OptFEMP1`. The latter are dedicated to the calculation of specific matrices : *mass* , *weighted mass*, *stiffness* and *elastic stiffness* matrices in space dimension 2 or 3. For each one of these matrices, we will use three different techniques implemented in `OptFEMP1` to carry out the assembly. Those will be referred as the versions `OptV1`, `OptV2` and `OptV3`. They are described in details in [1].

We now compare times calculations of these three versions of `OptFEMP1` with the `OptV3` version of `vecFEMP1` for the *stiffness* and *elastic stiffness* matrices in space dimension 2 and 3.

10.1.1 Assembly of the stiffness matrix in dimension $d = 2$ or 3

The $\mathrm{n_q}$-by-$\mathrm{n_q}$ stiffness matrix $\mathbb{S}$ is defined by

$$\mathbb{S}_{i,j} = \int_{\Omega_h} \langle \nabla \varphi_j, \nabla \varphi_i \rangle d\mathrm{q}, \quad \forall (i,j) \in \{1, ..., \mathrm{n_q}\}^2. \tag{10.1}$$

To assemble this matrix with `OptFEMP1` codes we use the dedicated functions

$$\mathbb{S} \leftarrow \text{AssemblyStiffP1<version>}(\mathcal{T}_h)$$

where <version> is either the version `OptV1`, `OptV2` or `OptV3`. With `vecFEMP1`, we use the generic function

$$\mathbb{S} \leftarrow \text{DAssemblyP1\_OptV3}(\mathcal{T}_h, \mathcal{L})$$

where $\mathcal{L} = \mathcal{L}_{\mathbb{I}_d, \mathbf{0}_d, \mathbf{0}_d, 0}$.

The computational costs to assemble this matrix compared to $\mathrm{n_{dof}} = \mathrm{n_q}$ for each previous function are represented in Figure 1 and Figure 2 respectively for the dimension $d = 2$ and $d = 3$. The detailed results are given in Table 1 and 2.

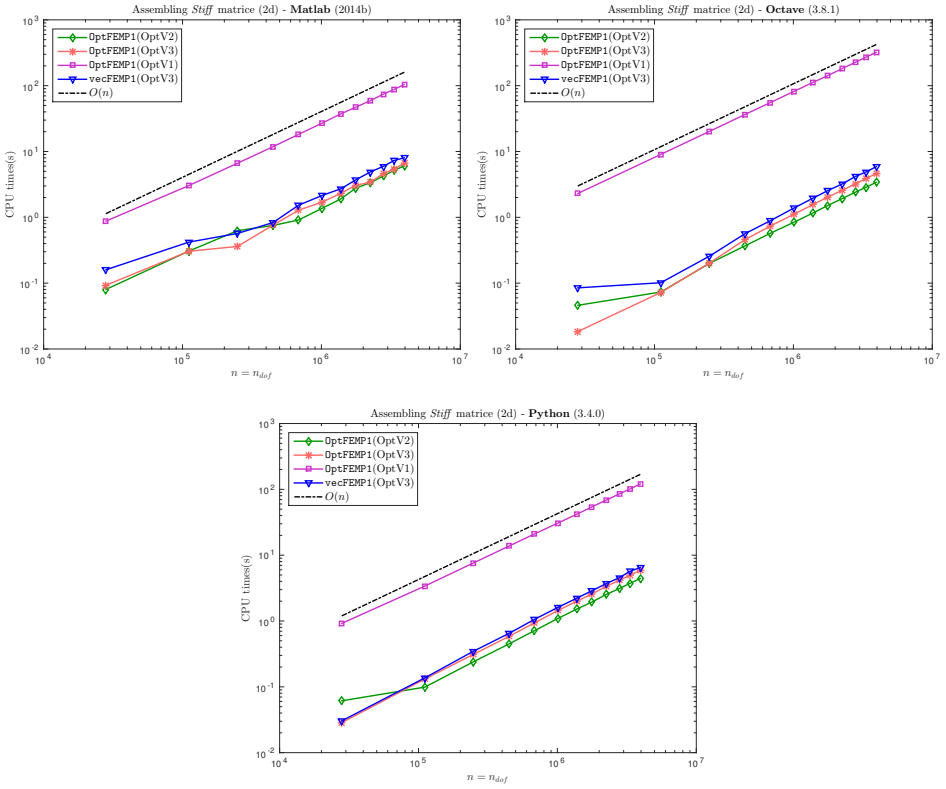


Figure 1: 2d stiffness matrix : comparison between `OptFEMP1` (OptV1, OptV2 and OptV3 versions) and `vecFEMP1` (OptV3 version) toolboxes in Matlab (top left), Octave (top right) and Python (bottom).

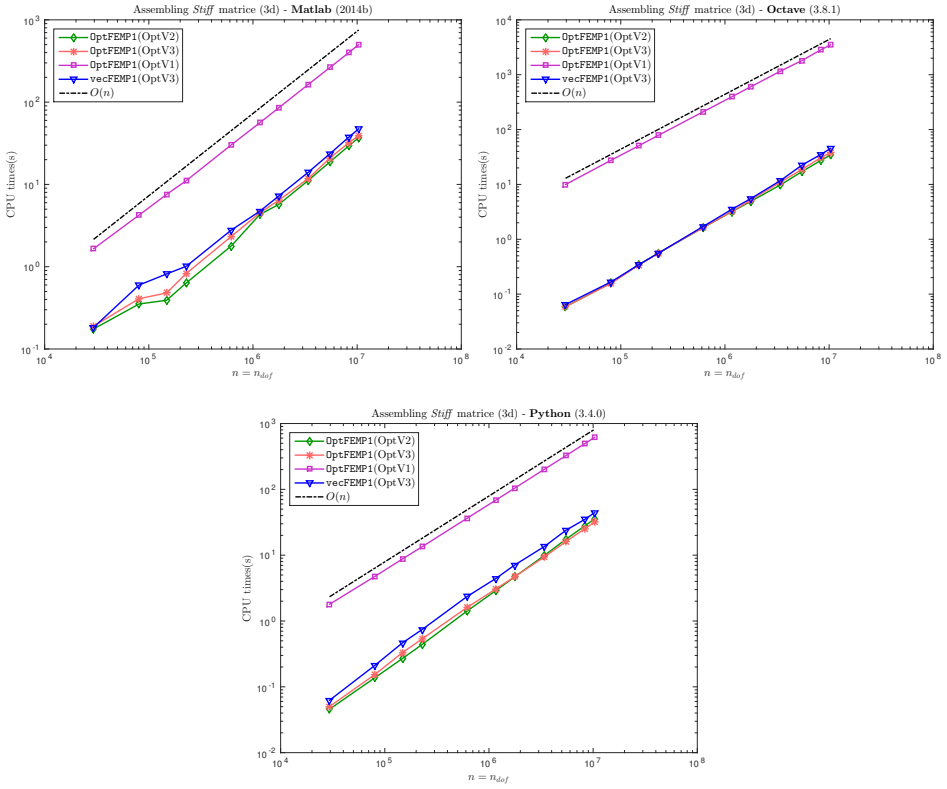


Figure 2: 3d stiffness matrix : comparison between `OptFEMP1` (OptV1, OptV2 and OptV3 versions) and `vecFEMP1` (OptV3 version) toolboxes in Matlab (top left), Octave (top right) and Python (bottom).

10.1.2 Assembly of the stiffness elasticity matrix in dimension $d = 2$ or 3

Here we consider sufficiently regular vector fields $\boldsymbol{u} = (u_1, \ldots, u_d) : \Omega \to \mathbb{R}^d$, with the associated discrete space $(X_h^1)^d$, $d = 2$ or 3 (i.e. $m = d$ in that case).

We consider the elastic stiffness matrix arising in linear elasticity when Hooke's law is used and the material is isotropic, under small strain hypothesis (see for example [5]). This sparse matrix $\mathbb{K}$ is defined by

$$\mathbb{K}_{l,n} = \int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\psi}_n)\mathbb{C}\underline{\boldsymbol{\epsilon}}(\boldsymbol{\psi}_l)d\mathrm{q}, \quad \forall (l,n) \in [\![1, \mathrm{n_{dof}}]\!]^2, \tag{10.2}$$

where $\boldsymbol{\psi}_{d(\alpha-1)+i} = \varphi_i \boldsymbol{e}_\alpha$, $\forall (\alpha, i) \in [\![1,d]\!] \times [\![1, \mathrm{n_q}]\!]$, and $\underline{\boldsymbol{\epsilon}}$ is the linearized strain tensor given by

$$\underline{\boldsymbol{\epsilon}}(\boldsymbol{u}) = \frac{1}{2}\left(\boldsymbol{\nabla}(\boldsymbol{u}) + \boldsymbol{\nabla}^t(\boldsymbol{u})\right),$$

with $\underline{\boldsymbol{\epsilon}} = (\epsilon_{11}, \epsilon_{22}, 2\epsilon_{12})^t$ in 2D and $\underline{\boldsymbol{\epsilon}} = (\epsilon_{11}, \epsilon_{22}, \epsilon_{33}, 2\epsilon_{12}, 2\epsilon_{23}, 2\epsilon_{13})^t$ in 3D, with $\epsilon_{ij}(\boldsymbol{u}) = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$. The elasticity tensor $\mathbb{C}$ depends on the Lamé parameters λ and μ satisfying $\lambda + \mu > 0$, and possibly variable in Ω. For $d = 2$ or $d = 3$, the matrix $\mathbb{C}$ is given by

$$\mathbb{C} = \begin{pmatrix} \lambda \mathbb{1}_2 + 2\mu \mathbb{I}_2 & \mathbb{O}_{2\times 1} \\ \mathbb{O}_{1\times 2} & \mu \end{pmatrix}_{3\times 3}, \qquad \mathbb{C} = \begin{pmatrix} \lambda \mathbb{1}_3 + 2\mu \mathbb{I}_3 & \mathbb{O}_{3\times 3} \\ \mathbb{O}_{3\times 3} & \mu \mathbb{I}_3 \end{pmatrix}_{6\times 6}.$$

Formula (10.2) is related to the Hooke's law

$$\underline{\boldsymbol{\sigma}} = \mathbb{C}\underline{\boldsymbol{\epsilon}},$$

StiffAssembling2DP1 - Matlab

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
28042	.080 (s) x 1	.093 (s) x 1.16	.874 (s) x 10.9	.160 (s) x 1.99
111036	.307 (s) x 1	.305 (s) x .993	3.04 (s) x 9.90	.420 (s) x 1.37
248818	.627 (s) x 1	.361 (s) x .576	6.64 (s) x 10.6	.569 (s) x .908
449492	.753 (s) x 1	.764 (s) x 1.01	11.9 (s) x 15.7	.834 (s) x 1.11
684162	.910 (s) x 1	1.28 (s) x 1.41	18.2 (s) x 20.0	1.53 (s) x 1.68
1011010	1.37 (s) x 1	1.70 (s) x 1.24	27.0 (s) x 19.7	2.15 (s) x 1.57
1376630	1.9 (s) x 1	2.31 (s) x 1.22	37.3 (s) x 19.6	2.71 (s) x 1.42
1767840	2.77 (s) x 1	3.06 (s) x 1.11	47.1 (s) x 17.0	3.71 (s) x 1.34
2250578	3.36 (s) x 1	3.42 (s) x 1.02	59.2 (s) x 17.6	4.89 (s) x 1.46
2798256	4.26 (s) x 1	4.65 (s) x 1.09	74.5 (s) x 17.5	5.97 (s) x 1.40
3337702	5.17 (s) x 1	5.38 (s) x 1.04	87.8 (s) x 17.0	7.31 (s) x 1.41
3952402	6.06 (s) x 1	6.68 (s) x 1.10	104 (s) x 17.2	8.03 (s) x 1.32

StiffAssembling2DP1 - Octave

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
28042	.046 (s) x 1	.018 (s) x .399	2.30 (s) x 50.0	.085 (s) x 1.84
111036	.073 (s) x 1	.072 (s) x .985	9.04 (s) x 123	.102 (s) x 1.38
248818	.200 (s) x 1	.202 (s) x 1.01	20.2 (s) x 101	.257 (s) x 1.29
449492	.370 (s) x 1	.460 (s) x 1.24	36.6 (s) x 98.9	.566 (s) x 1.53
684162	.574 (s) x 1	.739 (s) x 1.29	55.5 (s) x 96.8	.891 (s) x 1.55
1011010	.844 (s) x 1	1.12 (s) x 1.32	82.1 (s) x 97.2	1.39 (s) x 1.64
1376630	1.16 (s) x 1	1.56 (s) x 1.34	112 (s) x 96.3	1.94 (s) x 1.68
1767840	1.51 (s) x 1	2.03 (s) x 1.35	142 (s) x 94.0	2.54 (s) x 1.68
2250578	1.93 (s) x 1	2.58 (s) x 1.33	182 (s) x 94.4	3.22 (s) x 1.67
2798256	2.42 (s) x 1	3.23 (s) x 1.34	227 (s) x 94.0	4.18 (s) x 1.73
3337702	2.86 (s) x 1	3.88 (s) x 1.36	272 (s) x 95.0	4.90 (s) x 1.71
3952402	3.41 (s) x 1	4.63 (s) x 1.36	321 (s) x 94.2	5.88 (s) x 1.73

StiffAssembling2DP1 - Python

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
28042	.061 (s) x 1	.029 (s) x .467	.921 (s) x 15.0	.030 (s) x .496
111036	.099 (s) x 1	.131 (s) x 1.33	3.40 (s) x 34.3	.137 (s) x 1.38
248818	.240 (s) x 1	.312 (s) x 1.30	7.62 (s) x 31.7	.347 (s) x 1.45
449492	0.45 (s) x 1	.585 (s) x 1.30	13.8 (s) x 30.6	.650 (s) x 1.44
684162	.713 (s) x 1	.937 (s) x 1.31	21 (s) x 29.4	1.06 (s) x 1.48
1011010	1.10 (s) x 1	1.45 (s) x 1.32	30.8 (s) x 28.1	1.62 (s) x 1.47
1376630	1.52 (s) x 1	2.00 (s) x 1.32	41.8 (s) x 27.5	2.22 (s) x 1.46
1767840	1.94 (s) x 1	2.56 (s) x 1.32	54.1 (s) x 27.9	2.87 (s) x 1.48
2250578	2.54 (s) x 1	3.38 (s) x 1.33	68.7 (s) x 27.1	3.71 (s) x 1.46
2798256	3.12 (s) x 1	4.16 (s) x 1.33	85.5 (s) x 27.5	4.56 (s) x 1.46
3337702	3.74 (s) x 1	4.98 (s) x 1.33	102 (s) x 27.1	5.70 (s) x 1.52
3952402	4.41 (s) x 1	5.86 (s) x 1.33	121 (s) x 27.3	6.48 (s) x 1.47

Table 1: 2d stiffness matrix : comparison between `OptFEMP1` (OptV1, OptV2 and OptV3 versions) and `vecFEMP1` (OptV3 version) toolboxes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and speedup (bottom value). The speedup reference is `OptFEMP1 OptV2` version.

StiffAssembling3DP1 - Matlab

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
29497	.175 (s) x 1	.190 (s) x 1.08	1.65 (s) x 9.42	.183 (s) x 1.04
80607	.352 (s) x 1	.408 (s) x 1.16	4.24 (s) x 12.0	.598 (s) x 1.70
148076	.391 (s) x 1	.480 (s) x 1.23	7.55 (s) x 19.3	.816 (s) x 2.09
228933	.635 (s) x 1	.827 (s) x 1.30	11.1 (s) x 17.5	1.01 (s) x 1.59
617073	1.78 (s) x 1	2.32 (s) x 1.31	30.1 (s) x 16.9	2.78 (s) x 1.56
1163783	4.31 (s) x 1	4.50 (s) x 1.04	56.2 (s) x 13.0	4.72 (s) x 1.09
1768111	5.71 (s) x 1	6.41 (s) x 1.12	85.4 (s) x 15.0	7.23 (s) x 1.27
3411040	11.2 (s) x 1	11.8 (s) x 1.05	165 (s) x 14.6	14.1 (s) x 1.26
5493207	18.8 (s) x 1	21.0 (s) x 1.11	264 (s) x 14.0	23.7 (s) x 1.26
8309931	29.4 (s) x 1	32.1 (s) x 1.09	399 (s) x 13.6	37.5 (s) x 1.28
1.035E7	37.0 (s) x 1	39.2 (s) x 1.06	500 (s) x 13.5	47.5 (s) x 1.28

StiffAssembling3DP1 - Octave

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
29497	.059 (s) x 1	.059 (s) x 1.00	9.95 (s) x 169	.065 (s) x 1.10
80607	0.16 (s) x 1	.157 (s) x .981	27.5 (s) x 172	.167 (s) x 1.04
148076	.342 (s) x 1	.337 (s) x .985	50.5 (s) x 148	.342 (s) x .999
228933	.558 (s) x 1	.554 (s) x .993	78.3 (s) x 140	.556 (s) x .996
617073	1.63 (s) x 1	1.63 (s) x 1.00	211 (s) x 129	1.71 (s) x 1.05
1163783	3.18 (s) x 1	3.20 (s) x 1.01	396 (s) x 125	3.51 (s) x 1.11
1768111	4.91 (s) x 1	5.20 (s) x 1.06	603 (s) x 123	5.51 (s) x 1.12
3411040	9.79 (s) x 1	11.0 (s) x 1.12	1159 (s) x 118	11.8 (s) x 1.2
5493207	17.2 (s) x 1	18.8 (s) x 1.10	1818 (s) x 106	22.5 (s) x 1.31
8309931	27.3 (s) x 1	30.6 (s) x 1.12	2829 (s) x 104	35.1 (s) x 1.29
1.035E7	34.4 (s) x 1	38.1 (s) x 1.11	3523 (s) x 102	45.4 (s) x 1.32

StiffAssembling3DP1 - Python

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
29497	.046 (s) x 1	.050 (s) x 1.10	1.80 (s) x 39.2	.063 (s) x 1.37
80607	.138 (s) x 1	.155 (s) x 1.12	4.77 (s) x 34.6	.213 (s) x 1.54
148076	0.27 (s) x 1	.332 (s) x 1.23	8.78 (s) x 32.5	.466 (s) x 1.73
228933	.441 (s) x 1	.536 (s) x 1.22	13.5 (s) x 30.6	.745 (s) x 1.69
617073	1.42 (s) x 1	1.61 (s) x 1.13	36.4 (s) x 25.7	2.36 (s) x 1.66
1163783	2.90 (s) x 1	3.06 (s) x 1.06	68.5 (s) x 23.7	4.42 (s) x 1.53
1768111	4.69 (s) x 1	4.71 (s) x 1.01	104 (s) x 22.3	7.09 (s) x 1.51
3411040	9.95 (s) x 1	9.41 (s) x .946	202 (s) x 20.3	13.6 (s) x 1.36
5493207	17.5 (s) x 1	16.1 (s) x .922	326 (s) x 18.6	24.0 (s) x 1.37
8309931	27.8 (s) x 1	25.2 (s) x .907	494 (s) x 17.8	35.0 (s) x 1.26
1.035E7	35.7 (s) x 1	32.1 (s) x .898	617 (s) x 17.3	44.3 (s) x 1.24

Table 2: 3d stiffness matrix : comparison between OptFEMP1 (OptV1, OptV2 and OptV3 versions) and vecFEMP1Light (OptV3 version) toolboxes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and speedup (bottom value). The speedup reference is OptFEMP1 `OptV2` version.

where $\underline{\underline{\boldsymbol{\sigma}}}$ is the elastic stress tensor.

To assemble elastic stiffness matrix $\mathbb{K}$ with `OptFEMP1` codes we use the dedicated functions

$$\mathbb{K} \leftarrow \text{AssemblyStiffElasP1<version>}(\mathcal{T}_h)$$

where <version> is either the version `OptV1`, `OptV2` or `OptV3`. With `vecFEMP1`, we use the generic function

$$\mathbb{K} \leftarrow \text{HAssemblyP1\_OptV3}(\mathcal{T}_h, \mathcal{H})$$

where $\mathcal{H}$ is given in Section 3.3.1 Lemma 1.

The computational costs to assemble this matrix compared to $\mathrm{n_{dof}} = d\mathrm{n_q}$ for each previous function are represented in Figure 3 and Figure 4 respectively for the dimension $d = 2$ and $d = 3$. The detailed results are given in Table 3 and 4.

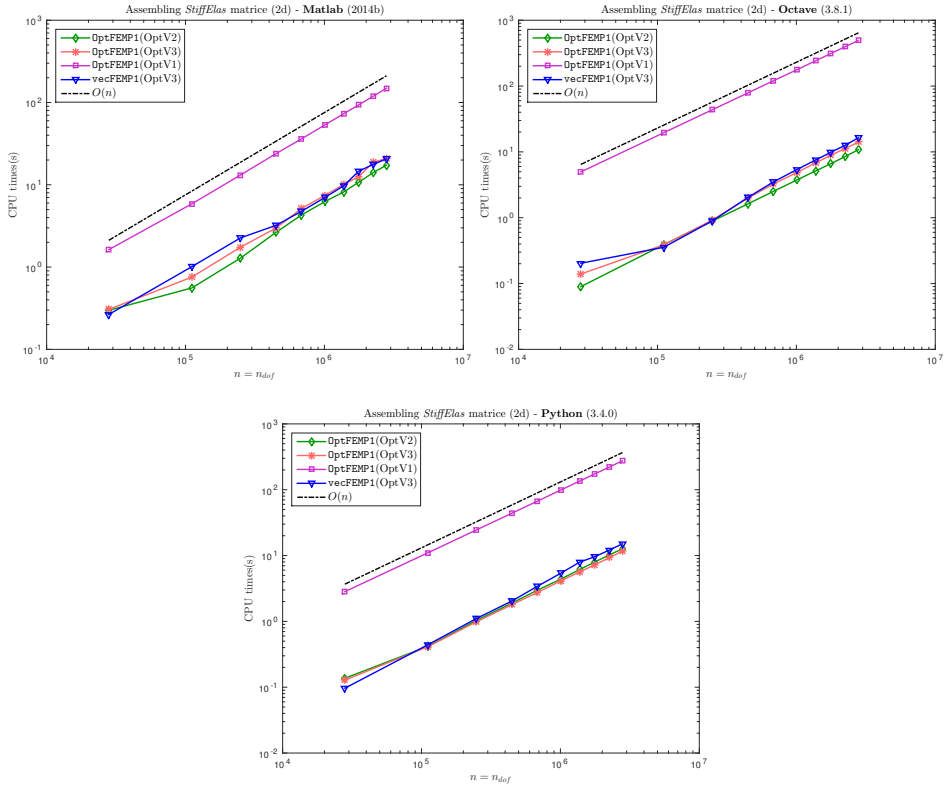


Figure 3: 2d stiffness elasticity matrix : comparison between `OptFEMP1` (OptV1, OptV2 and OptV3 versions) and `vecFEMP1` (OptV3 version) toolboxes in Matlab (top left), Octave (top right) and Python (bottom).

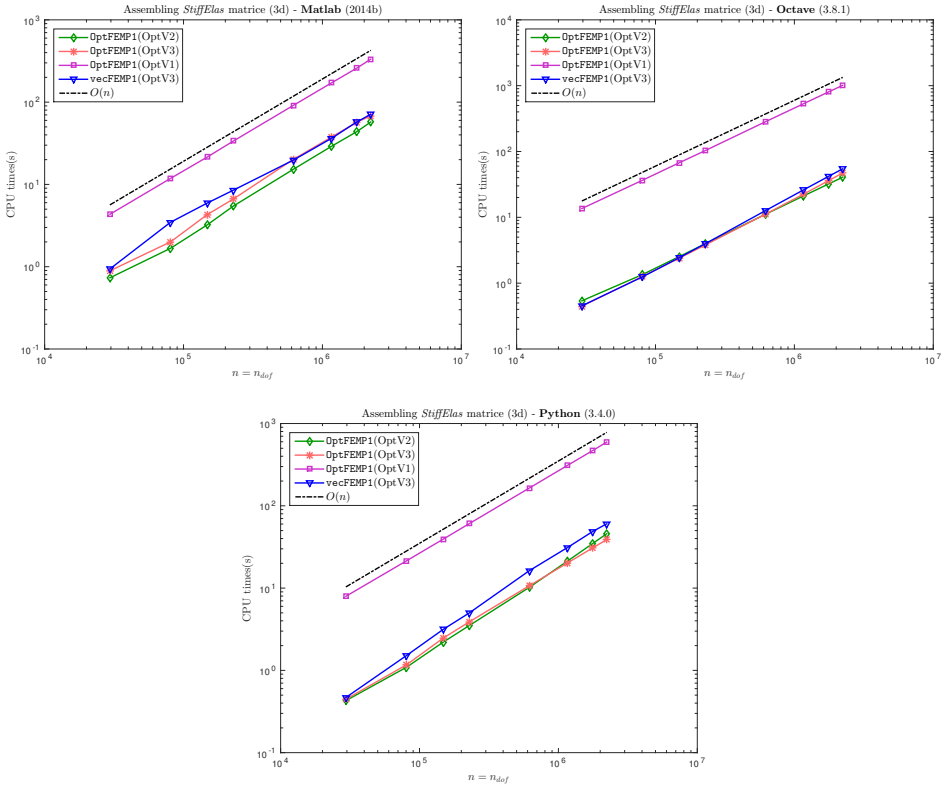


Figure 4: 3d stiffness elasticity matrix : comparison between `OptFEMP1` (OptV1, OptV2 and OptV3 versions) and `vecFEMP1` (OptV3 version) toolboxes in Matlab (top left), Octave (top right) and Python (bottom).

StiffElasAssembling2DP1 - Matlab

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
28042	.299 (s) x 1	.306 (s) x 1.03	1.63 (s) x 5.45	.266 (s) x .890
111036	.557 (s) x 1	.755 (s) x 1.35	5.87 (s) x 10.5	1.01 (s) x 1.82
248818	1.29 (s) x 1	1.75 (s) x 1.36	13.1 (s) x 10.2	2.26 (s) x 1.76
449492	2.65 (s) x 1	2.99 (s) x 1.13	24.0 (s) x 9.05	3.22 (s) x 1.22
684162	4.30 (s) x 1	5.16 (s) x 1.20	36.2 (s) x 8.41	4.77 (s) x 1.11
1011010	6.27 (s) x 1	7.44 (s) x 1.19	53.6 (s) x 8.55	7.03 (s) x 1.12
1376630	8.18 (s) x 1	10.1 (s) x 1.23	73.1 (s) x 8.93	9.73 (s) x 1.19
1767840	10.7 (s) x 1	12.5 (s) x 1.17	93.8 (s) x 8.73	14.6 (s) x 1.36
2250578	14.1 (s) x 1	18.9 (s) x 1.34	119 (s) x 8.47	17.7 (s) x 1.26
2798256	17.3 (s) x 1	20.8 (s) x 1.20	149 (s) x 8.61	20.7 (s) x 1.19

StiffElasAssembling2DP1 - Octave

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
28042	.090 (s) x 1	.139 (s) x 1.56	4.96 (s) x 55.4	.203 (s) x 2.27
111036	0.39 (s) x 1	.388 (s) x .994	19.6 (s) x 50.2	.356 (s) x .912
248818	.892 (s) x 1	.922 (s) x 1.03	43.8 (s) x 49.1	.900 (s) x 1.01
449492	1.63 (s) x 1	1.99 (s) x 1.22	79 (s) x 48.5	2.04 (s) x 1.25
684162	2.51 (s) x 1	3.26 (s) x 1.30	120 (s) x 48.0	3.5 (s) x 1.39
1011010	3.76 (s) x 1	4.87 (s) x 1.30	178 (s) x 47.3	5.41 (s) x 1.44
1376630	5.14 (s) x 1	6.83 (s) x 1.33	243 (s) x 47.2	7.53 (s) x 1.47
1767840	6.61 (s) x 1	8.85 (s) x 1.34	312 (s) x 47.2	9.84 (s) x 1.49
2250578	8.51 (s) x 1	11.3 (s) x 1.32	397 (s) x 46.7	12.6 (s) x 1.48
2798256	10.8 (s) x 1	14.1 (s) x 1.30	492 (s) x 45.5	16.4 (s) x 1.52

StiffElasAssembling2DP1 - Python

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
28042	.137 (s) x 1	.128 (s) x .938	2.83 (s) x 20.7	.097 (s) x .705
111036	.418 (s) x 1	.412 (s) x .986	11 (s) x 26.4	.441 (s) x 1.06
248818	1.03 (s) x 1	.988 (s) x .954	24.5 (s) x 23.7	1.11 (s) x 1.07
449492	1.90 (s) x 1	1.80 (s) x .948	44.2 (s) x 23.2	2.05 (s) x 1.08
684162	2.99 (s) x 1	2.75 (s) x .922	67.2 (s) x 22.5	3.42 (s) x 1.14
1011010	4.40 (s) x 1	4.14 (s) x .940	99.4 (s) x 22.6	5.43 (s) x 1.23
1376630	6.13 (s) x 1	5.65 (s) x .921	135 (s) x 22.1	7.94 (s) x 1.29
1767840	7.98 (s) x 1	7.22 (s) x .905	174 (s) x 21.8	9.61 (s) x 1.20
2250578	10.2 (s) x 1	9.31 (s) x .913	221 (s) x 21.7	12.1 (s) x 1.19
2798256	12.7 (s) x 1	11.7 (s) x .920	275 (s) x 21.7	15.1 (s) x 1.19

Table 3: 2d stiffness elasticity matrix : comparison between `OptFEMP1` (OptV1, OptV2 and OptV3 versions) and `vecFEMP1` (OptV3 version) toolboxes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and speedup (bottom value). The speedup reference is `OptFEMP1 OptV2` version.

StiffElasAssembling3DP1 - Matlab

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
29497	.732 (s) x 1	.891 (s) x 1.22	4.33 (s) x 5.92	.939 (s) x 1.28
80607	1.67 (s) x 1	2.00 (s) x 1.20	11.8 (s) x 7.07	3.47 (s) x 2.07
148076	3.22 (s) x 1	4.29 (s) x 1.33	21.5 (s) x 6.67	5.93 (s) x 1.84
228933	5.47 (s) x 1	6.67 (s) x 1.22	33.7 (s) x 6.17	8.53 (s) x 1.56
617073	15.3 (s) x 1	20.1 (s) x 1.32	91.2 (s) x 5.98	19.7 (s) x 1.29
1163783	29.1 (s) x 1	37.6 (s) x 1.29	172 (s) x 5.90	36.3 (s) x 1.25
1768111	43.8 (s) x 1	57.0 (s) x 1.30	259 (s) x 5.91	57.4 (s) x 1.31
2229968	57.3 (s) x 1	68.1 (s) x 1.19	333 (s) x 5.81	71.2 (s) x 1.24

StiffElasAssembling3DP1 - Octave

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
29497	.539 (s) x 1	.446 (s) x .826	13.6 (s) x 25.2	.453 (s) x .840
80607	1.35 (s) x 1	1.27 (s) x .935	36.6 (s) x 27.1	1.25 (s) x .922
148076	2.53 (s) x 1	2.39 (s) x .946	67.4 (s) x 26.6	2.44 (s) x .963
228933	3.98 (s) x 1	3.82 (s) x .958	104 (s) x 26.0	3.99 (s) x 1.00
617073	11.1 (s) x 1	11.1 (s) x 1.00	282 (s) x 25.5	12.6 (s) x 1.14
1163783	21.1 (s) x 1	22.5 (s) x 1.06	532 (s) x 25.2	26.1 (s) x 1.24
1768111	32.2 (s) x 1	36.2 (s) x 1.12	806 (s) x 25	42.0 (s) x 1.30
2229968	40.7 (s) x 1	47.3 (s) x 1.16	1014 (s) x 24.9	54.7 (s) x 1.34

StiffElasAssembling3DP1 - Python

n_{dof}	OptFEMP1 OptV2	OptFEMP1 OptV3	OptFEMP1 OptV1	vecFEMP1 OptV3
29497	.429 (s) x 1	.446 (s) x 1.04	7.94 (s) x 18.5	.468 (s) x 1.09
80607	1.09 (s) x 1	1.17 (s) x 1.07	21.5 (s) x 19.7	1.52 (s) x 1.39
148076	2.20 (s) x 1	2.47 (s) x 1.12	39.3 (s) x 17.9	3.15 (s) x 1.43
228933	3.52 (s) x 1	3.90 (s) x 1.11	60.9 (s) x 17.3	5.01 (s) x 1.42
617073	10.2 (s) x 1	10.7 (s) x 1.05	164 (s) x 16.1	16.3 (s) x 1.59
1163783	21.3 (s) x 1	20.2 (s) x .949	310 (s) x 14.5	31.1 (s) x 1.46
1768111	34.9 (s) x 1	30.8 (s) x .883	470 (s) x 13.5	48.8 (s) x 1.40
2229968	45.8 (s) x 1	38.9 (s) x .849	595 (s) x 13.0	60.5 (s) x 1.32

Table 4: 3d stiffness elasticity matrix : comparison between `OptFEMP1` (OptV1, OptV2 and OptV3 versions) and `vecFEMP1` (OptV3 version) toolboxes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and speedup (bottom value). The speedup reference is `OptFEMP1 OptV2` version.

10.2 BVP benchmarking

In this section we evaluate the performance of the toolbox `vecFEMP1` in Matlab (Release 2014b), Octave (3.8.1) and Python (3.4.0) on our reference machine gpucreos1. In each language we will give times in seconds to assemble the linear system and to solve it. We use the most generic solution function in each language : \ in Matlab/Octave and the function `spsolve` in SciPy for Python.

10.2.1 2D stationary convection-diffusion benchmark

We solve the BVP given in Section 3.2.4. given in Table 5.

	Matlab		Octave		Python		FreeFEM++	
n_{dof}	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)
25253	.375	.376	.157	.161	.183	.133	.375	.009
98712	1.14	1.09	.469	.714	0.82	0.72	1.35	.063
226226	2.25	2.25	1.34	2.08	1.95	2.08	3.15	.161
393202	3.28	3.98	2.55	4.49	3.52	4.56	5.46	.292
605547	5.36	6.27	4.10	8.65	5.52	8.90	8.45	.463
878642	7.95	9.74	5.97	15.4	8.02	15.9	12.2	.691
1190916	10.6	15.1	8.21	26.8	11	27.6	16.8	.961

Table 5: Comparison of BVP solving : 2D stationary convection-diffusion - Matlab with `OptV3` assembling and `classic` solve, Octave with `OptV3` assembling and `classic` solve, Python with `OptV3` assembling and `classic` solve, FreeFEM++ with `OptV3` assembling and `sparsesolver` solve,

10.2.2 3D stationary convection-diffusion benchmark

We solve the BVP given in Section 3.2.5. given in Table 6.

10.2.3 2D linear elasticity benchmark

We solve the BVP given in Section 3.3.1 using Algorithm 3.8. The computational times are given in Table 7.

10.2.4 3D linear elasticity benchmark

We solve the BVP given in Section 3.3.1 using Algorithm 3.9. The computational times are given in Table 8.

11 Appendix

11.1 Example of mesh data structure for a ring mesh

For example, we give in Figure 1 a ring mesh with 36 vertices, 48 mesh elements and 24 boundary elements. The data structure associated to the mesh is denoted by $\mathcal{T}_h$ and verifies

	Matlab		Octave		Python		FreeFEM++	
n_{dof}	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)
9610	.534	.807	.218	.988	.253	.508	1.09	.610
68137	2.63	8.71	1.92	10.6	2.66	5.42	8.41	7.63
222768	9.75	45.7	6.91	59.7	9.60	31.1	29.0	35.7
497673	25.2	111	16.9	138	21.4	77.4	64.1	81.1
974856	46.8	243	39.5	347	42.8	184	142	210

Table 6: Comparison of BVP solving : 3D stationary convection-diffusion - Matlab with `OptV3` assembling and `bicgstab` solve, Octave with `OptV3` assembling and `bicgstab` solve, Python with `OptV3` assembling and `bicgstab` solve, FreeFEM++ with `OptV3` assembling and `GMRES` solve,

	Matlab		Octave		Python		FreeFEM++	
n_{dof}	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)
51102	.547	0.3	.192	.237	.233	.676	.740	.040
202202	2.18	1.02	.695	1.28	1.07	4.56	2.96	.181
453302	4.08	2.19	1.76	3.41	2.47	15.1	6.79	.447
804402	6.79	3.83	3.45	6.84	4.70	45.0	11.9	.809
1255502	10.8	5.85	5.61	12.3	7.52	87.7	18.5	1.61

Table 7: Comparison of BVP solving : 2D linear elasticity - Matlab with `OptV3` assembling and `classic` solve, Octave with `OptV3` assembling and `classic` solve, Python with `OptV3` assembling and `spLU` solve, FreeFEM++ with `OptV3` assembling and `sparsesolver` solve,

n_{dof}	Matlab		Octave		Python		FreeFEM++	
	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)	System(s)	Solve(s)
14883	.892	1.63	.363	2.93	.386	1.63	1.08	1.03
107163	7.33	32.0	2.34	43.5	4.06	22.7	8.58	17.3
348843	19.2	113	10.4	210	16.1	114	29.5	84.9
811923	47.1	342	25.9	663	39.2	372	69.3	266

Table 8: Comparison of BVP solving : 3D linear elasticity - Matlab with `OptV3` assembling and `pcg` solve, Octave with `OptV3` assembling and `pcg` solve, Python with `OptV3` assembling and `minres` solve, FreeFEM++ with `OptV3` assembling and `CG` solve.

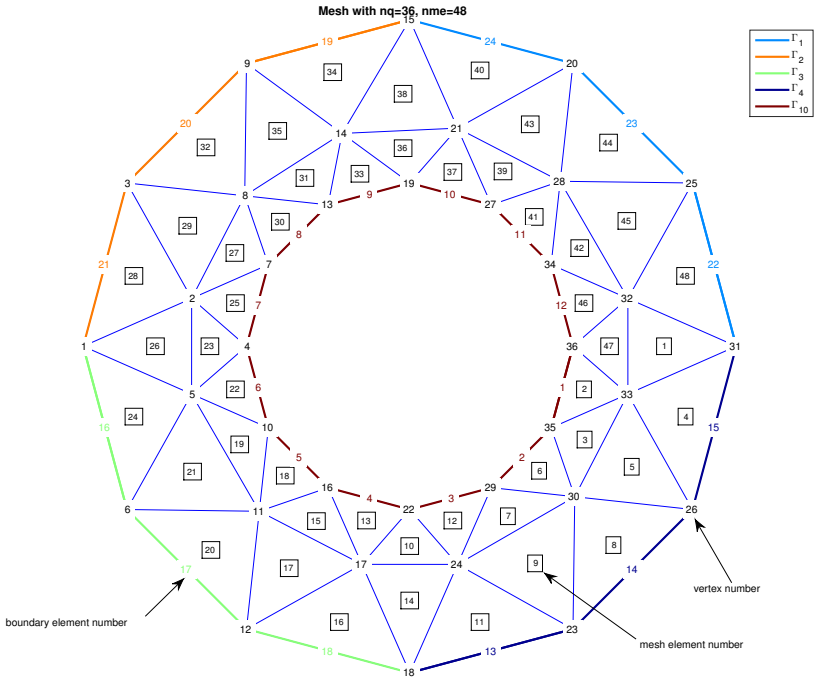


Figure 1: Coarse ring mesh

$\mathcal{T}_h.\mathrm{n_q} = 36,\ \mathcal{T}_h.\mathrm{n_{me}} = 48,\ \mathcal{T}_h.\mathrm{n_{be}} = 24,$

$$\mathcal{T}_h.\mathrm{q} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & & 34 & 35 & 36 \end{matrix} \\ \begin{pmatrix} -1.000 & -0.668 & -0.866 & \dots & 0.433 & 0.433 & 0.500 \\ 0.000 & 0.146 & 0.500 & \dots & 0.250 & -0.250 & -0.000 \end{pmatrix} \end{array}$$

$$\mathcal{T}_h.\mathrm{me} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & & 44 & 45 & 46 & 47 & 48 \end{matrix} \\ \begin{pmatrix} 33 & 35 & 35 & 31 & 30 & \dots & 25 & 32 & 32 & 33 & 31 \\ 31 & 33 & 30 & 33 & 26 & \dots & 20 & 25 & 34 & 32 & 25 \\ 32 & 36 & 33 & 26 & 33 & \dots & 28 & 28 & 36 & 36 & 32 \end{pmatrix} \end{array}$$

$$\mathcal{T}_h.\mathrm{be} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & & 20 & 21 & 22 & 23 & 24 \end{matrix} \\ \begin{pmatrix} 36 & 35 & 29 & 22 & 16 & \dots & 9 & 3 & 31 & 25 & 20 \\ 35 & 29 & 22 & 16 & 10 & \dots & 3 & 1 & 25 & 20 & 15 \end{pmatrix} \end{array}$$

$$\mathcal{T}_h.\mathrm{bel} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & & 20 & 21 & 22 & 23 & 24 \end{matrix} \\ \begin{pmatrix} 10 & 10 & 10 & 10 & 10 & \dots & 2 & 2 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

11.2 Example of boundary mesh data structure

On the example of Figure 1 we have $lab = [1, 2, 3, 4, 10]$ and for example if we consider the fifth boundary of label 10 ($lab(5) = 10$), we have $\mathcal{B}_\mathrm{h}(5).\mathrm{nq} = 12$, $\mathcal{B}_\mathrm{h}(5).\mathrm{nme} = 12$, $\mathcal{B}_\mathrm{h}(5).\mathrm{label} = 10$.

$$\mathcal{B}_\mathrm{h}(5).\mathrm{q} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & & 10 & 11 & 12 \end{matrix} \\ \begin{pmatrix} -0.500 & -0.433 & -0.433 & \dots & 0.433 & 0.433 & 0.500 \\ -0.000 & 0.250 & -0.250 & \dots & 0.250 & -0.250 & -0.000 \end{pmatrix} \end{array}$$

$$\mathcal{B}_\mathrm{h}(5).\mathrm{me} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{matrix} \\ \begin{pmatrix} 12 & 11 & 9 & 7 & 5 & 3 & 1 & 2 & 4 & 6 & 8 & 10 \\ 11 & 9 & 7 & 5 & 3 & 1 & 2 & 4 & 6 & 8 & 10 & 12 \end{pmatrix} \end{array}$$

$$\mathcal{B}_\mathrm{h}(5).\mathrm{toGlobal} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{matrix} \\ \begin{pmatrix} 4 & 7 & 10 & 13 & 16 & 19 & 22 & 27 & 29 & 34 & 35 & 36 \end{pmatrix} \end{array}$$

11.3 BuildBoundaryMeshes function

Algorithm 11.1 function BuildBoundaryMeshes

Input :

$\mathcal{T}_h$: a mesh structure of Ω

Output :

$\mathcal{B}_h$: 1-by-$\mathrm{n_{lab}}$ array of boundary mesh structures.

```
1: Function Bh ← BuildBoundaryMeshes(Th)
2:    labels ← Unique(Th.bel)
3:    nlab ← Length(labels)
4:    for l ← 1 to nlab do
5:       Bh(l) ← BuildBoundaryMesh(Th, labels(l))
6:    end for
7: end Function
```

11.4 Elasticity in $\mathbb{R}^d$

11.4.1 Mathematical notations

We want to prove (3.34) of Lemma 1. We can write (3.30) as

$$-(\operatorname{div}\sigma(\boldsymbol{u}))_i = f_i,\ \forall i \in [\![1,d]\!],\ \text{in } \Omega \tag{11.1}$$

We have

$$\begin{aligned}(\operatorname{div}\sigma(\boldsymbol{u}))_i &= (2\mu\epsilon_{ij}(\boldsymbol{u}))_{,j} + (\lambda\epsilon_{kk}(\boldsymbol{u}))_{,i}\\ &= \sum_{j=1}^{d}\frac{\partial}{\partial x_j}\left(2\mu\epsilon_{ij}(\boldsymbol{u})\right) + \frac{\partial}{\partial x_i}\left(\lambda\sum_{k=1}^{d}\epsilon_{kk}(\boldsymbol{u})\right)\end{aligned}$$

and

$$\epsilon_{ij}(\boldsymbol{u}) = \frac{1}{2}\left(\frac{\partial \boldsymbol{u}_i}{\partial x_j} + \frac{\partial \boldsymbol{u}_j}{\partial x_i}\right)$$

So we obtain

$$\begin{aligned}(\operatorname{div}\sigma(\boldsymbol{u}))_i &= \sum_{j=1}^{d}\frac{\partial}{\partial x_j}\left(\mu\left(\frac{\partial \boldsymbol{u}_i}{\partial x_j} + \frac{\partial \boldsymbol{u}_j}{\partial x_i}\right)\right) + \frac{\partial}{\partial x_i}\left(\lambda\sum_{j=1}^{d}\frac{\partial u_j}{\partial x_j}\right)\\ &= \sum_{j=1}^{d}\left\{\frac{\partial}{\partial x_j}\left(\mu\frac{\partial \boldsymbol{u}_i}{\partial x_j}\right) + \frac{\partial}{\partial x_j}\left(\mu\frac{\partial \boldsymbol{u}_j}{\partial x_i}\right)\right\} + \sum_{j=1}^{d}\frac{\partial}{\partial x_i}\left(\lambda\frac{\partial \boldsymbol{u}_j}{\partial x_j}\right)\\ &= \sum_{k=1}^{d}\frac{\partial}{\partial x_k}\left(\mu\frac{\partial \boldsymbol{u}_i}{\partial x_k}\right) + \sum_{j=1}^{d}\frac{\partial}{\partial x_j}\left(\mu\frac{\partial \boldsymbol{u}_j}{\partial x_i}\right) + \sum_{j=1}^{d}\frac{\partial}{\partial x_i}\left(\lambda\frac{\partial \boldsymbol{u}_j}{\partial x_j}\right)\\ &= \sum_{j=1}^{d}\left\{\sum_{k=1}^{d}\frac{\partial}{\partial x_k}\left(\mu\frac{\partial \boldsymbol{u}_j}{\partial x_k}\right)\delta_{ij} + \frac{\partial}{\partial x_j}\left(\mu\frac{\partial \boldsymbol{u}_j}{\partial x_i}\right)\right\} + \sum_{j=1}^{d}\frac{\partial}{\partial x_i}\left(\lambda\frac{\partial \boldsymbol{u}_j}{\partial x_j}\right)\end{aligned}$$

So, from (2.10) and (11.1) we want $\forall i \in [\![1,d]\!]$

$$(\mathcal{H}(\boldsymbol{u}))_i = \sum_{j=1}^{d}\mathcal{H}_{i,j}(\boldsymbol{u}_j) = (\operatorname{div}\sigma(\boldsymbol{u}))_i.$$

and by identification, we obtain $\forall i \in [\![1,d]\!]$

$$\mathcal{H}_{i,j}(\boldsymbol{u}_j) = \frac{\partial}{\partial x_j}\left(\mu\frac{\partial \boldsymbol{u}_j}{\partial x_i}\right) + \frac{\partial}{\partial x_i}\left(\lambda\frac{\partial \boldsymbol{u}_j}{\partial x_j}\right) + \sum_{k=1}^{d}\frac{\partial}{\partial x_k}\left(\mu\frac{\partial \boldsymbol{u}_j}{\partial x_k}\right)\delta_{ij}$$

and then $\mathcal{H}_{i,j} := \mathcal{L}_{\mathbb{A}^{i,j},\mathbf{0},\mathbf{0},0}$ with

$$(\mathbb{A}^{i,j})_{k,l} = \mu\delta_{k,l}\delta_{i,j} + \mu\delta_{k,j}\delta_{l,i} + \lambda\delta_{k,i}\delta_{l,j},\ \forall (k,l) \in [\![1,d]\!]^2,$$

With these notations, we can rewrite the elasticity problem (11.1) as

$$\sum_{j=1}^{d}\operatorname{div}(\mathbb{A}^{i,j}\nabla \boldsymbol{u}_j) + f_i = 0,\ \forall i \in [\![1,d]\!],\ \text{in } \Omega \tag{11.2}$$

In dimension $d = 2$, (11.2) becomes

$$\operatorname{div}\left(\begin{pmatrix}\gamma & 0\\ 0 & \mu\end{pmatrix}\nabla \boldsymbol{u}_1\right) + \operatorname{div}\left(\begin{pmatrix}0 & \lambda\\ \mu & 0\end{pmatrix}\nabla \boldsymbol{u}_2\right) + f_1 = 0, \tag{11.3}$$

$$\operatorname{div}\left(\begin{pmatrix}0 & \mu\\ \lambda & 0\end{pmatrix}\nabla \boldsymbol{u}_1\right) + \operatorname{div}\left(\begin{pmatrix}\mu & 0\\ 0 & \gamma\end{pmatrix}\nabla \boldsymbol{u}_2\right) + f_2 = 0, \tag{11.4}$$

and in dimension $d = 3$

$$\operatorname{div}\left(\begin{pmatrix}\gamma & 0 & 0\\ 0 & \mu & 0\\ 0 & 0 & \mu\end{pmatrix}\nabla \boldsymbol{u}_1\right) + \operatorname{div}\left(\begin{pmatrix}0 & \lambda & 0\\ \mu & 0 & 0\\ 0 & 0 & 0\end{pmatrix}\nabla \boldsymbol{u}_2\right) + \operatorname{div}\left(\begin{pmatrix}0 & 0 & \lambda\\ 0 & 0 & 0\\ \mu & 0 & 0\end{pmatrix}\nabla \boldsymbol{u}_3\right) + f_1 = 0, \tag{11.5}$$

$$\operatorname{div}\left(\begin{pmatrix}0 & \mu & 0\\ \lambda & 0 & 0\\ 0 & 0 & 0\end{pmatrix}\nabla \boldsymbol{u}_1\right) + \operatorname{div}\left(\begin{pmatrix}\mu & 0 & 0\\ 0 & \gamma & 0\\ 0 & 0 & \mu\end{pmatrix}\nabla \boldsymbol{u}_2\right) + \operatorname{div}\left(\begin{pmatrix}0 & 0 & 0\\ 0 & 0 & \lambda\\ 0 & \mu & 0\end{pmatrix}\nabla \boldsymbol{u}_3\right) + f_2 = 0, \tag{11.6}$$

$$\operatorname{div}\left(\begin{pmatrix}0 & 0 & \mu\\ 0 & 0 & 0\\ \lambda & 0 & 0\end{pmatrix}\nabla \boldsymbol{u}_1\right) + \operatorname{div}\left(\begin{pmatrix}0 & 0 & 0\\ 0 & 0 & \mu\\ 0 & \lambda & 0\end{pmatrix}\nabla \boldsymbol{u}_2\right) + \operatorname{div}\left(\begin{pmatrix}\mu & 0 & 0\\ 0 & \mu & 0\\ 0 & 0 & \gamma\end{pmatrix}\nabla \boldsymbol{u}_3\right) + f_3 = 0, \tag{11.7}$$

11.4.2 Boundary conditions

We want to prove (3.35) of Lemma 1. We set $\frac{\partial \boldsymbol{u}_j}{\partial n_{\mathcal{H}_{i,j}}} := \frac{\partial \boldsymbol{u}_j}{\partial n_{i,j}}$ and, by definition of $\mathcal{H}_{i,j}$ operators, we obtain on Γ

$$\sum_{j=1}^{d} \frac{\partial \boldsymbol{u}_j}{\partial n_{i,j}} = \left\langle \mathbb{A}^{i,i} \nabla \boldsymbol{u}_i, \boldsymbol{n} \right\rangle + \sum_{\substack{j=1 \\ j\neq i}}^{d} \left\langle \mathbb{A}^{i,j} \nabla \boldsymbol{u}_j, \boldsymbol{n} \right\rangle$$

But we have

$$\begin{aligned} \left\langle \mathbb{A}^{i,i} \nabla \boldsymbol{u}_i, \boldsymbol{n} \right\rangle &= \sum_{k=1}^{d} \sum_{l=1}^{d} \mathbb{A}^{i,i}_{k,l} \frac{\partial \boldsymbol{u}_j}{\partial x_l} \boldsymbol{n}_k \\ &= \sum_{k=1}^{d} \sum_{l=1}^{d} \left(\mu \delta_{k,l} + (\lambda+\mu) \delta_{k,i} \delta_{l,i} \right) \frac{\partial \boldsymbol{u}_i}{\partial x_l} \boldsymbol{n}_k \\ &= \mu \sum_{k=1}^{d} \frac{\partial \boldsymbol{u}_i}{\partial x_k} \boldsymbol{n}_k + (\lambda+\mu) \frac{\partial \boldsymbol{u}_i}{\partial x_i} \end{aligned}$$

and, for $j \neq i$

$$\begin{aligned} \left\langle \mathbb{A}^{i,j} \nabla \boldsymbol{u}_j, \boldsymbol{n} \right\rangle &= \sum_{k=1}^{d} \sum_{l=1}^{d} \mathbb{A}^{i,j}_{k,l} \frac{\partial \boldsymbol{u}_j}{\partial x_l} \boldsymbol{n}_k \\ &= \sum_{k=1}^{d} \sum_{l=1}^{d} \left(\lambda \delta_{k,i} \delta_{l,j} + \mu \delta_{k,j} \delta_{l,i} \right) \frac{\partial \boldsymbol{u}_j}{\partial x_l} \boldsymbol{n}_k \\ &= \lambda \frac{\partial \boldsymbol{u}_j}{\partial x_j} \boldsymbol{n}_i + \mu \frac{\partial \boldsymbol{u}_j}{\partial x_i} \boldsymbol{n}_j. \end{aligned}$$

So we obtain

$$\sum_{j=1}^{d} \frac{\partial \boldsymbol{u}_j}{\partial n_{i,j}} = \sum_{\substack{j=1 \\ j\neq i}}^{d} \left(\lambda \frac{\partial \boldsymbol{u}_j}{\partial x_j} \boldsymbol{n}_i + \mu \frac{\partial \boldsymbol{u}_j}{\partial x_i} \boldsymbol{n}_j \right) + \mu \sum_{k=1}^{d} \frac{\partial \boldsymbol{u}_i}{\partial x_k} \boldsymbol{n}_k + (\lambda+\mu) \frac{\partial \boldsymbol{u}_i}{\partial x_i} \boldsymbol{n}_i \tag{11.8}$$

In linear elasticity boundary conditions can be expressed as *Dirichlet* or as $(\sigma(\vec{u})\boldsymbol{n})_i = g$ for example. We have

$$(\sigma(\vec{u})\boldsymbol{n})_i = \sum_{j=1}^{d} \sigma_{i,j}(\boldsymbol{u}) \boldsymbol{n}_j$$

with

$$\sigma_{i,j}(\boldsymbol{u}) = 2\mu \epsilon_{i,j}(\boldsymbol{u}) + \lambda \delta_{i,j} \sum_{k=1}^{d} \epsilon_{k,k}(\boldsymbol{u}) \quad \text{and} \quad \epsilon_{i,j}(\boldsymbol{u}) = \frac{1}{2} \left(\frac{\partial \boldsymbol{u}_j}{\partial x_i} + \frac{\partial \boldsymbol{u}_i}{\partial x_j} \right)$$

So we have with $\sigma_{i,j}(\boldsymbol{u}) = 2\mu \epsilon_{i,j}(\boldsymbol{u}) + \lambda \delta_{i,j} \sum_{k=1}^{d} \epsilon_{k,k}(\boldsymbol{u})$

$$\begin{aligned} (\sigma(\vec{u})\boldsymbol{n})_i &= \sigma_{i,i}(\boldsymbol{u}) \boldsymbol{n}_i + \sum_{\substack{j=1 \\ j\neq i}}^{d} \sigma_{i,j}(\boldsymbol{u}) \boldsymbol{n}_j \\ &= \left(2\mu \epsilon_{i,i}(\boldsymbol{u}) + \lambda \sum_{k=1}^{d} \epsilon_{k,k}(\boldsymbol{u}) \right) \boldsymbol{n}_i + \sum_{\substack{j=1 \\ j\neq i}}^{d} 2\mu \epsilon_{i,j}(\boldsymbol{u}) \boldsymbol{n}_j \end{aligned}$$

We also have $\epsilon_{i,j}(\boldsymbol{u}) = \frac{1}{2} \left(\frac{\partial \boldsymbol{u}_j}{\partial x_i} + \frac{\partial \boldsymbol{u}_i}{\partial x_j} \right)$ and then

$$\begin{aligned} (\sigma(\vec{u})\boldsymbol{n})_i &= \sum_{\substack{j=1 \\ j\neq i}}^{d} \mu \left(\frac{\partial \boldsymbol{u}_j}{\partial x_i} + \frac{\partial \boldsymbol{u}_i}{\partial x_j} \right) \boldsymbol{n}_j + \left(2\mu \frac{\partial \boldsymbol{u}_i}{\partial x_i} + \lambda \sum_{k=1}^{d} \frac{\partial \boldsymbol{u}_k}{\partial x_k} \right) \boldsymbol{n}_i \\ &= \sum_{\substack{j=1 \\ j\neq i}}^{d} \mu \frac{\partial \boldsymbol{u}_j}{\partial x_i} \boldsymbol{n}_j + \sum_{\substack{j=1 \\ j\neq i}}^{d} \mu \frac{\partial \boldsymbol{u}_i}{\partial x_j} \boldsymbol{n}_j + \lambda \sum_{\substack{k=1 \\ k\neq i}}^{d} \frac{\partial \boldsymbol{u}_k}{\partial x_k} \boldsymbol{n}_i + (\lambda+2\mu) \frac{\partial \boldsymbol{u}_i}{\partial x_i} \boldsymbol{n}_i \\ &= \sum_{\substack{j=1 \\ j\neq i}}^{d} \left(\mu \frac{\partial \boldsymbol{u}_j}{\partial x_i} \boldsymbol{n}_j + \lambda \frac{\partial \boldsymbol{u}_j}{\partial x_j} \boldsymbol{n}_i \right) + \sum_{j=1}^{d} \mu \frac{\partial \boldsymbol{u}_i}{\partial x_j} \boldsymbol{n}_j + (\lambda+\mu) \frac{\partial \boldsymbol{u}_i}{\partial x_i} \boldsymbol{n}_i. \end{aligned}$$

So we have proved that

$$(\sigma(\vec{u})\boldsymbol{n})_i = \sum_{j=1}^{d} \frac{\partial \boldsymbol{u}_j}{\partial n_{i,j}}, \ \forall i \in [\![1,d]\!] \tag{11.9}$$

11.5 Computation of the barycentric coordinates

Let T_k be a d-simplex in $\mathbb{R}^d$ of vertices $\mathrm{q}^0, \ldots, \mathrm{q}^d$ and $\hat{T}$ be the reference d-simplex with $\hat{\mathrm{q}}^0, \ldots, \hat{\mathrm{q}}^d$ such that $\hat{\mathrm{q}}^0 = \mathbf{0}_d$ and $\hat{\mathrm{q}}^i = \boldsymbol{e}_i$, $\forall i \in [\![1,d]\!]$.

We denote by $\mathcal{F}_k$ the bijection from $\hat{T}$ to T_k defined by $\mathrm{q} = \mathcal{F}_k(\hat{\mathrm{q}}) = \mathbb{B}_k\hat{\mathrm{q}} + \mathrm{q}^0$ where $\mathbb{B}_k \in \mathcal{M}_d(\mathbb{R})$ is such that, for all $i \in [\![1,d]\!]$, the i-th column is equal to $\mathrm{q}^i - \mathrm{q}^0$.

The barycentric coordinates of $\hat{\mathrm{q}} = (\hat{x}_1, \ldots, \hat{x}_d) \in \hat{T}$ are given by $\hat{\lambda}_0 = 1 - \sum_{i=1}^{d} \hat{x}_i$ and $\hat{\lambda}_k = \hat{x}_i$, $\forall i \in [\![1,d]\!]$.

The barycentric coordinates of $\mathrm{q} = (x_1, \ldots, x_d) \in T_k$ are given by $\lambda_{k,i}(\mathrm{q}) = \hat{\lambda}_i \circ \mathcal{F}_k^{-1}(\mathrm{q})$ and we have

$$\nabla \lambda_{k,i}(\mathrm{q}) = \mathbb{B}_k^{-t} \hat{\nabla} \hat{\lambda}_i(\hat{\mathrm{q}}), \ \forall i \in [\![0,d]\!] \tag{11.10}$$

with $\hat{\nabla}\hat{\lambda}_0(\hat{\mathrm{q}}) = \begin{pmatrix} -1 \\ \ldots \\ -1 \end{pmatrix}$, $\hat{\nabla}\hat{\lambda}_i = \boldsymbol{e}_i$, $\forall i \in [\![1,d]\!]$. We may note that gradients are constant.

By setting

$$\hat{\mathbb{G}} = \left(\hat{\nabla}\hat{\lambda}_0, \quad \ldots, \quad \hat{\nabla}\hat{\lambda}_d\right) = \begin{pmatrix} -1 & 1 & 0 & \ldots & 0 \\ -1 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ -1 & 0 & \ldots & 0 & 1 \end{pmatrix}$$

computing the gradients of barycentric coordinates is equivalent to solve $(d+1)$ linear systems which writes under matrix form

$$\mathbb{B}_k^t \mathbb{G}_k = \hat{\mathbb{G}} \tag{11.11}$$

where $\mathbb{G}_k = \left(\nabla \lambda_{k,0}(\mathrm{q}), \quad \ldots, \quad \nabla \lambda_{k,d}(\mathrm{q})\right) \in \mathcal{M}_{d,d+1}(\mathbb{R})$.

Algorithm 11.2 Gradient of basis functions on a d-simplex in dimension d.

Input :
`q` : d-by-$(d+1)$ array, `q`$(:,i)$ is the i-th vertex of the d-simplex

Output :
$\boldsymbol{G}$: gradient array (d-by-$(d+1)$)
$\boldsymbol{G}(:,\alpha) = \nabla \lambda_\alpha(\mathrm{q})$, $\forall \alpha \in [\![1,d+1]\!]$

Function $\boldsymbol{G} \leftarrow$ GRADIENTS(`q`)
  `Grad` $\leftarrow [-\mathbf{1}_d, \mathbb{I}_{d\times d}]$ ▷ Contains gradients of $\hat{\lambda}_i$
  $\mathbb{B} \leftarrow$ `q`$(:, 2:d+1)^t - \mathbf{1}_d *$ `q`$(:,1)^t$
  $\boldsymbol{G} \leftarrow$ INV($\mathbb{B}$) $*$ `Grad`
end Function

For each d-simplex in the mesh, there are $(d+1)$ gradients to be computed which gives $(d+1)\mathrm{n_{me}}$ vectors of dimension d to be computed.

To vectorize the computation of all the gradients, we need to rewrite in an equivalent form the $\mathrm{n_{me}}$ equations (11.11) in an only one big sparse block-diagonal linear system where each diagonal block is of dimension d-by-d :

$$\begin{pmatrix} \mathbb{B}_1^t & \mathbb{O} & \ldots & \mathbb{O} \\ \mathbb{O} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathbb{O} \\ \mathbb{O} & \ldots & \mathbb{O} & \mathbb{B}_{\mathrm{n_{me}}}^t \end{pmatrix}_{N\times N} \begin{pmatrix} \mathbb{G}_1 \\ \mathbb{G}_2 \\ \vdots \\ \mathbb{G}_{\mathrm{n_{me}}} \end{pmatrix}_{N\times(d+1)} = \begin{pmatrix} \hat{\mathbb{G}} \\ \hat{\mathbb{G}} \\ \vdots \\ \hat{\mathbb{G}} \end{pmatrix}_{N\times(d+1)} \tag{11.12}$$

with $N = d \times \mathrm{n_{me}}$.

Algorithm 11.3 Vectorized computation of the gradients of basis functions in dimension d

Input :
q : d-by-$\mathrm{n_q}$ array
me : $(d+1)$-by-$\mathrm{n_{me}}$ connectivity array
Output :
$\mathbb{G}$: gradient array ($\mathrm{n_{me}}$-by-$(d+1)$-by-d)
$\mathbb{G}(k,\alpha,:) = \nabla\,\varphi_\alpha^k(\mathrm{q}),\ \forall \alpha \in [\![1, d+1]\!]$

Function $\boldsymbol{G} \leftarrow$ GradientVec(q, me)
  $\mathbb{K} \leftarrow$ zeros$(d, d, \mathrm{n_{me}})$, $\mathbb{I} \leftarrow$ zeros$(d, d, \mathrm{n_{me}})$, $\mathbb{J} \leftarrow$ zeros$(d, d, \mathrm{n_{me}})$
  $\mathtt{ii} \leftarrow d * [0 : (\mathrm{n_{me}} - 1)]$
  for $i \leftarrow 1$ **to** d **do**
    for $j \leftarrow 1$ **to** d **do**
      $\mathbb{K}(i,j,:) \leftarrow \mathrm{q}(i, \mathrm{me}(j+1,:)) - \mathrm{q}(i, \mathrm{me}(1,:))$
      $\mathbb{I}(i,j,:) \leftarrow \mathtt{ii} + j$, $\mathbb{J}(i,j,:) \leftarrow \mathtt{ii} + i$
    end for
  end for
  $\mathbb{S} \leftarrow$ sparse$(I(:), J(:), K(:), d * \mathrm{n_{me}}, d * \mathrm{n_{me}})$
  $\mathbb{R} \leftarrow$ zeros$(d\mathrm{n_{me}}, d+1)$ ▷ Build RHS
  $\hat{\mathbb{G}} \leftarrow [-$ones$(d, 1),$ eye$(d)]$
  $\mathtt{ii} \leftarrow 1 : d$
  for $k \leftarrow 1$ **to** $\mathrm{n_{me}}$ **do**
    $\mathbb{R}(ii,:) \leftarrow \hat{\mathbb{G}}$, $\mathtt{ii} \leftarrow \mathtt{ii} + d$
  end for
  $\mathbb{G} \leftarrow$ solve$(\mathbb{S}, \mathbb{R})$
  $\mathbb{G} \leftarrow$ reshape() ▷ A finir...
end Function

12 Vectorized algorithmic language

In computer science, vector languages (also known as array programming or multidimensional languages) are often used in scientific and engineering computations. They generalize the operations on scalars to higher dimensional arrays, matrices and vectors : such operations are named vectorized operations. We provide below some common functions and operators of the vectorized algorithmic language used in this article, which is close to Matlab/Octave.

$\mathbb{A} \leftarrow \mathbb{B}$	Assignment
$\mathbb{A} * \mathbb{B}$	matrix multiplication,
$\mathbb{A}. * \mathbb{B}$	element-wise multiplication,
$\mathbb{A}./\mathbb{B}$	element-wise division,
$\mathbb{A}(:)$	all the elements of $\mathbb{A}$, regarded as a single column.
[,]	Horizontal concatenation,
[;]	Vertical concatenation,
$\mathbb{A}(:, J)$	J-th column of $\mathbb{A}$
$\mathbb{A}(I, :)$	I-th row of $\mathbb{A}$
$\mathtt{sum}(\mathbb{A}, dim)$	sums along the dimension dim.
$\mathbb{1}_{m\times n}$	m-by-n array or sparse matrix of ones.
$\mathbb{O}_{m\times n}$	m-by-n array or sparse matrix of zeros.
$a \equiv b$	boolean for testing if a is equal to b.
Unique(I)	returns the unique values of I after sorting.
Length$(\mathbb{A})$	returns the maximum size of A over each dimension.
Size$(\mathbb{A})$	size of A in each dimension.
Find(I)	

List of Algorithms

Liste des bclogo

Index

References

[1] F. Cuvelier, C. Japhet, and G. Scarella. An efficient way to perform the assembly of finite element matrices in vector languages. `http://hal.archives-ouvertes.fr/hal-00931066v1`, 2014.

[2] F. Cuvelier and G. Scarella. mvecfemp1: a matlab/octave toolbox to solve boundary value and eigenvalues problems by a p1-lagrange finite element method in any space dimension. `http://www.math.univ-paris13.fr/~cuvelier/software/`, 2015.

[3] F. Cuvelier and G. Scarella. pyvecfemp1: a python package to solve boundary value and eigenvalues problems by a p1-lagrange finite element method in any space dimension. `http://www.math.univ-paris13.fr/~cuvelier/software/`, 2015.

[4] T. A. Davis. `SuiteSparse` packages, released 4.2.1. `http://faculty.cse.tamu.edu/davis/suitesparse.html`, 2013.

[5] G. Dhatt, E. Lefrançois, and G. Touzot. *Finite Element Method*. Wiley, 2012.

[6] T. Gerasimov. *The clamped elastic grid, a fourth order equation on a domain with corner*. PhD thesis, Delft University of Technology, the Netherlands, 2009.

[7] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[8] F. Hecht. Freefem++. `www.freefem.org/ff++/index.htm`, 2014.

[9] NVIDIA. Cusp, a C++ Templated Sparse Matrix Library. `https://github.com/cusplibrary/cusplibrary`, 2015.

[10] NVIDIA. Thrust, a C++ template library for CUDA based on the Standard Template Library (STL). `https://github.com/thrust/thrust`, 2015.

[11] A. Quarteroni. *Numerical Models for Differential Problems*, volume 8 of *MS&A. Modeling, Simulation and Applications*. Springer, Milan, second edition, 2014. Translated from the fifth (2012) Italian edition by Silvia Quarteroni.

[12] A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*, volume 23 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 1994.