Politecnico di Milano

Advanced Programming for Scientific Computing

A finite element method for scalar and vectorial interface problems.

Comparison of a symmetric and non symmetric implementation with `GetFEM++`.

Alessandra Arrigoni
alessandra.arrigoni@mail.polimi.it

Sara Francesca Pichierri
sarafrancesca.pichierri@mail.polimi.it

Supervisor:

Prof. Luca Formaggia
luca.formaggia@polimi.it

Prof.ssa Anna Scotti
anna.scotti@polimi.it

February 16, 2020

POLITECNICO
MILANO 1863

Contents

1 Introduction

Interface problems regularly arise in the context of numerical analysis of PDEs applied to geophysics, with the aim to model and study the formation of earthquakes starting from the analysis of the slip conditions between two different subduction zones.
In particular, an accurate solution of the inverse problem, that is, the determination of cracks from boundary measurements and the reconstruction of the geometry of a fault, has become more and more important in order to anticipate and possibly prevent the occurrence of large earthquakes.
Another field that involves interface problems and the coupling of different equations is the study of the flow in fractured porous media, whose applications span from water resource management to the isolation of radioactive waste to prevent water contamination, to name just a few. In the case of fractures with very small width compared both to their length and to the dimensions of the whole computational domain, indeed, they can be modelled as $(d-1)$ dimensional interfaces between two d-dimensional domains.

The first main issue arising in this context is the construction of a suitable computational grid, which can be either conforming (matching and aligned) or not to the interface. The first choice is suitable for simple networks of fractures and allows the usual assembly of the matrices describing the differential operators associated to the PDE of interest; in more complex cases, however, it may imply the creation of badly shaped elements and very fine grids, dramatically increasing the computational cost of the meshing process, especially for 3D problems.
The second choice, instead, allows the use of regular grids on the bulk which are arbitrarily cut by the cracks; as a result, the integration on the elements that are far from the interface can be performed efficiently and accurately, but some *ad hoc* integration techniques are needed on those elements that are trimmed by the fracture.

The second point to be addressed is the choice of the numerical method applied to solve the given differential problem. In particular, since the interface usually determines a discontinuity in the physical solution, the method should be able to reproduce this same property on the numerical one.
Since their introduction in the early 1970s, the family of discontinuous Galerkin methods has represented the most natural choice to describe this type of solutions, given the intrinsic discontinuity of their basis functions (see [2] for an application of DG methods to flow in fractured porous media). As a main drawback, however, we must mention the large number of degrees of freedom they introduce, which increases both the computational cost of solving the system associated to the problem and the memory usage.
In the most common case, where the solution is regular except at the interface, another possibility is to employ a Petrov-Galerkin finite element method. As explained and proven in [1] for a particular case, the choice of two different discrete spaces to represent the solution and the test functions let us build a sequence of approximations that converge to the exact discontinuous solution with optimal order. From the computational point of view, this approach produces a smaller system with respect to the one coming from a DG method, for a given mesh; indeed there are less degrees of freedom since the trial functions can be chosen to be continuous everywhere except at the interface.

The aim of this project is to study the convergence of the finite element method described in [1] when applied to a slightly different configuration than the one proposed in the paper; here indeed we consider an interface that splits the 2-dimensional domain in two halves, intersecting the external boundaries in two points.
We provide two different implementations of the linear system and compare the performances in the two cases with respect to the matrix condition number. The first approach derives directly from the method formulation, while the second one involves an algebraic rearrangement of the matrix in order to reduce its dimension and preserve its symmetry, when present.
To perform the numerical tests we chose a regular triangle mesh of the unit square conforming to the fixed interface and studied both the elliptic problem (scalar solution) and the linear elasticity equation (vectorial solution). In both cases we obtained optimal convergence order for the errors in L^2 and H^1 norms for regular solutions regardless of which implementation we used.

The report is organized as follows: in Section 2 we introduce the differential problems, in their strong and weak forms, together with the discrete formulations of the FE method; then in Section 3 we describe the structure of the two different matrices used to write the algebraic formulation of the problems. In the following Section 4 we describe the C++ code based on the open source finite element library `GetFEM++`: an overview of the general structure and of some auxiliary classes are given in Subsection 4.1, while the main classes describing the differential problems are presented in Subsection 4.2. To conclude, in Section 5 we explain how to compile and run the code, in Section 6 we show the results of the numerical tests and in Section 7 we draw our conclusions and provide some ideas to improve and extend the present work.

2 Problem and method definition

First of all, we introduce the notation and the assumptions used throughout the report.
Let $\Omega \subset \mathbf{R}^2$ be a bounded Lipschitz domain, which is split by a Lipschitz curve Γ, representing the interface, in two disjoint connected subdomains Ω_1 and Ω_2 such that $\Omega \setminus \Gamma = \Omega_1 \cup \Omega_2$ and $\Omega_1 \cap \Omega_2 = 0$. In particular, we will always consider the unit square with a vertical interface at $x = 0.5$, as shown in Figure 1 and denote by $\boldsymbol{\nu}_1$ the outward unit vector on Γ with respect Ω_1; moreover, let $\Gamma_1 = \partial\Omega_1 \setminus \Gamma$ and $\Gamma_2 = \partial\Omega_2 \setminus \Gamma$ be the induced partition of $\partial\Omega$.
In order to write the weak formulations of the differential problems, we must define suitable functional spaces: for a bounded domain $D \in \mathbf{R}^2$ we introduce the usual Sobolev spaces of integer order $k \geq 0$: $H^k(D)$ with norm $\|\cdot\|_{k,D}$, seminorm $|\cdot|_{k,D}$ and inner product $(\cdot,\cdot)_D$; $H^k(\partial D)$ with norm $|\cdot|_{k,\partial D}$ and inner product $\langle\cdot,\cdot\rangle_{\partial D}$. For any real $k \geq 0$ the corresponding spaces are defined by interpolation.
Moreover, for a bounded open set $G = \bigcup_{j=1}^m D_j$ with D_j open and mutually disjoint, we denote by $H^k(G)$ the Sobolev space of functions w with restrictions $w|_{D_j} \in H^k(D_j)$; we define the norm and seminorm on $H^k(G)$ as follows:

$$\|w\|_{k,G} = \Big(\sum_{j=1}^m \|w\|^2_{k,D_j}\Big)^{1/2} \quad \text{and} \quad |w|_{k,G} = \Big(\sum_{j=1}^m |w|^2_{k,D_j}\Big)^{1/2}.$$

We denote by $\mathbf{H}^m(D) = [H^m(D)]^2$ and $\mathbf{H}^m(G) = [H^m(G)]^2$, $m \geq 0$ their usual vectorial counterparts.

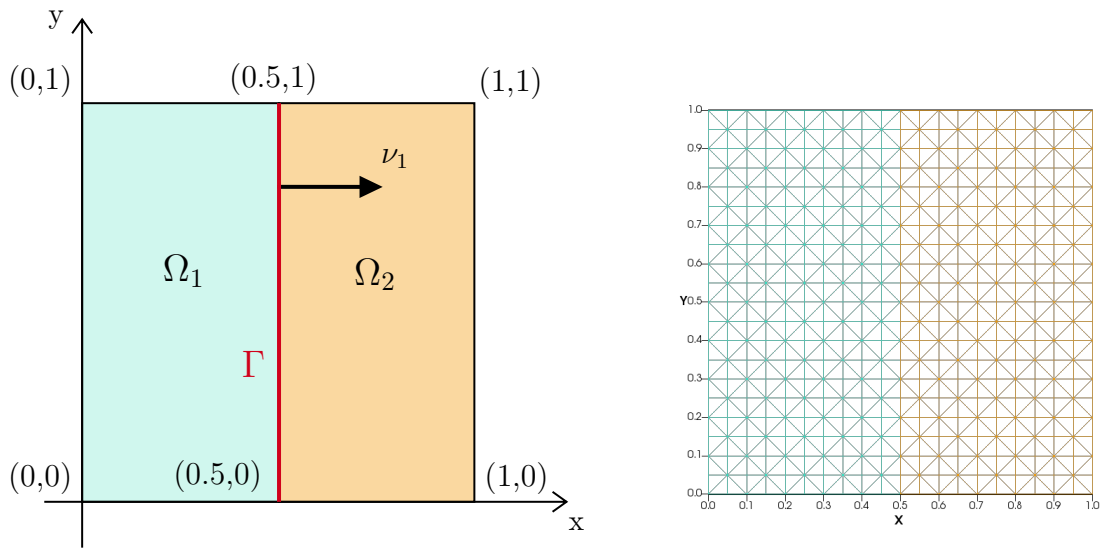


Figure 1: Computational domain $\Omega = [0,1] \times [0,1]$: unit square with vertical interface at $x = 0.5$. On the right the coarsest mesh used in the project (20 subdivisions in each dimension) is shown.

The second step to describe the finite element method is the definition of a discretization of the domain. Let $\{\mathcal{T}_h\}$ be a family of triangulations of Ω associated to the discretizing parameter h such that:

$$\mathcal{T}_h = \mathcal{T}_h^1 \cup \mathcal{T}_h^2 \quad \text{with} \quad \mathcal{T}_h^1 = \Omega_1, \ \mathcal{T}_h^2 = \Omega_2 \quad \text{and} \quad \mathcal{T}_h^1 \cap \mathcal{T}_h^2 = \emptyset.$$

As usual, we assume that the triangulation is conforming, quasiuniform, i.e. $\max_{\tau_k \in \mathcal{T}_h} h_k \lesssim \min_{\tau_k \in \mathcal{T}_h} h_k$, where h_k denotes the measure of each triangle, and that every element τ_k is affine equivalent to a reference triangle. Furthermore, we assume that it is aligned with Γ and conforming, implying that no hanging node is allowed on the interface. As a result we can construct a one-dimensional discretization of Γ containing only those vertices of the triangles $\tau_k \in \mathcal{T}_h$ that lie on the polygonal curve; the same procedure is applied to generate the discretization of the boundary $\Gamma_1 \cup \Gamma_2$.

We now introduce the discrete spaces of piecewise polynomials employed in the finite element method. Given the definition of $V_h(\Omega_k)$, $k = 1, 2$ as the space of continuous piecewise linear functions on Ω_k relative to the triangulation $\mathcal{T}_h$, let us denote by V_h the set of functions $\phi \in H^1(\Omega_1 \cup \Omega_2)$ such that $\phi|_{\Omega_k} \in V_h(\Omega_k)$. Moreover we denote by $V_h(\Gamma)$ and $V_h(\Gamma_1 \cup \Gamma_2)$ the spaces of piecewise linear functions on these curves, relative to the discretization induced by $\mathcal{T}_h$ previously introduced; notice that, for the second space, discontinuities are allowed at the intersection points of Γ_1 and Γ_2.
As in the continuous case, we use the notation $\mathbf{V}_h, \mathbf{V}_h(\Omega_k), \mathbf{V}_h(\Gamma)$ and $\mathbf{V}_h(\Gamma_1 \cup \Gamma_2)$ to indicate their vectorial counterparts.
The last preliminary step consists in defining the L^2 projectors onto the boundary spaces; let us denote by $Q_{h,\Gamma} \colon L^2(\Gamma) \to V_h(\Gamma)$ and by $Q_h \colon L^2(\Gamma_1 \cup \Gamma_2) \to V_h(\Gamma_1 \cup \Gamma_2)$ the usual orthogonal

projectors satisfying, respectively,

$$\begin{aligned} \langle Q_{h,\Gamma}\,\psi, \chi\rangle_\Gamma &= \langle \psi, \chi\rangle_\Gamma && \forall\ \chi \in V_h(\Gamma), \\ \langle Q_h\,\phi, \xi\rangle_{\Gamma_1\cup\Gamma_2} &= \langle \phi, \xi\rangle_{\Gamma_1\cup\Gamma_2} && \forall\ \xi \in V_h(\Gamma_1\cup\Gamma_2). \end{aligned}$$

Once again, the projectors linked to the vectorial spaces are defined in the same way and denoted by $\mathbf{Q}_{h,\Gamma}$ and $\mathbf{Q}_h$.

Notice that in the more general case described in [1] the boundary and the interface may be smooth curves, requiring additional care in approximating them with polygonal lines and in transferring the boundary and interface data.

2.1 Elliptic equation

The first interface problem we consider on the domain Ω in Figure 1 is the classic scalar elliptic problem, whose strong form is the following:

$$\begin{aligned} A_1 u_1 = f \text{ in } \Omega_1, \quad A_2 u_2 = f \quad \text{in } \Omega_2 \\ u_2 - u_1 = q_0 \quad \text{on } \Gamma \\ \frac{\partial u_2}{\partial \nu_{A_2}} - \frac{\partial u_1}{\partial \nu_{A_1}} = q_1 \quad \text{on } \Gamma \\ u_1 = g_1 \quad \text{on } \Gamma_1, \quad u_2 = g_2 \quad \text{on } \Gamma_2 \end{aligned} \tag{1}$$

where A_1 and A_2 are the second order uniformly elliptic operators with smooth coefficients $a_{ij}^k \in C^2(\Omega_k')$, where $\overline{\Omega}_k \subset \Omega_k'$:

$$A_k = -\sum_{i,j=1}^{2} \frac{\partial}{\partial x_i}(a_{ij}^k(x)\frac{\partial}{\partial x_j}), \quad x \in \Omega_k,\ k = 1,2$$

Moreover $\partial w/\partial \nu_{A_k}$ denotes the conormal derivative:

$$\frac{\partial w}{\partial \nu_{A_k}}(x) = \sum_{i,j=1}^{2} a_{ij}^k(x)\frac{\partial w}{\partial x_i}(x)\,\nu_{1,j}, \quad x \in \Gamma.$$

The **weak form** of problem (1) reads: for $f \in L^2(\Omega)$, $q_0 \in H^{r-1/2}(\Gamma)$, $q_1 \in H^{r-3/2}(\Gamma)$ and $g \in H^{r-1/2}(\Gamma_1 \cup \Gamma_2)$ with $1 \le r \le 2$, find $u \in H^1(\Omega_1 \cup \Omega_2)$ such that:

$$\begin{aligned} a(u,v) = (f,v)_\Omega - \langle q_1, v\rangle_\Gamma, \quad \forall\ v \in H_0^1(\Omega) \\ [u] = q_0 \text{ on } \Gamma, \quad u = g \text{ on } \partial\Omega \end{aligned} \tag{2}$$

where $[u] = u_2 - u_1$ and $u_i = u|_{\Omega_i}$.

The continuous bilinear form $a(\cdot,\cdot) : H^1(\Omega_1 \cup \Omega_2) \times H^1(\Omega_1 \cup \Omega_2) \to \mathbf{R}$ associated with the elliptic operator is defined as $a(u,v) = a^1(u,v) + a^2(u,v)$, where for $k = 1,2$

$$a^k(u,v) = \sum_{i,j=1}^{2} \int_{\Omega_k} a_{ij}^k(x)\frac{\partial u}{\partial x_i}\frac{\partial v}{\partial x_j}dx, \qquad u, v \in H^1(\Omega_k).$$

Notice that the Sobolev spaces associated to the test function v and to the solution u are different; in particular, the choice for the latter allows for discontinuities in the solution across the smooth interface, which would be banned for functions belonging to $H^1(\Omega)$ on the whole domain.

At this point we can write the finite element method analyzed in this project, starting by defining the discrete boundary data g_h and discrete interface jump $q_{0,h}$ as the projections on the finite dimensional spaces $V_h(\Gamma_1 \cup \Gamma_2)$ and $V_h(\Gamma)$ of the continuous data:

$$g_h := Q_h\, g \qquad q_{0,h} := Q_{h,\Gamma}\, q_0.$$

The **approximate problem** reads as follows: in Ω find $u_h \in V_h$ such that

$$\begin{aligned} a(u_h, \phi) &= (f, \phi)_\Omega - \langle q_1, \phi \rangle_\Gamma \qquad \forall\ \phi \in V_h^0 \\ \text{with } [u_h] = q_{0,h} \text{ on } \Gamma \quad &\text{and} \quad u_h = g_h \text{ on } \partial\Omega, \end{aligned} \tag{3}$$

where $V_h^0 = V_h \cap H_0^1(\Omega)$, $[u_h] = u_{h2} - u_{h1}$ and $f = 0$ outside Ω. On the right hand side there are the usual inner products in $L^2(\Omega)$ and $L^2(\Gamma)$, respectively.

Notice that the jump condition and the Dirichlet boundary condition are assigned in a direct way to the approximate solution u_h and that the interface condition on the conormal derivative q_1 does not need a projection onto the discrete space $V_h(\Gamma)$ since it only appears inside a L^2 inner product on Γ.
As already mentioned, this method can be ascribed to the Petrov-Galerkin class of finite element methods, since the spaces for the test (ϕ) and the trial functions (u_h) are different. It is nevertheless a conforming method, as the discrete spaces are subspaces of the continuous ones and reflect their asymmetric definition.
We remark that here we can write the same bilinear form defined for 2 since we restrict our discussion to polygonal domains; in the more general case treated in [1] it must be modified by computing the integrals on the polygonal approximations of Ω_1 and Ω_2 and the additional error must be controlled.

2.2 Linear elasticity equation

The second problem analyzed in this project is the classic linear compressible elasticity equation, governing the displacement $\mathbf{u} \in \mathbf{R}^2$ of a bidimensional plate Ω.
We will formulate the problem using the two Lamé coefficients $\lambda, \mu \in \mathbf{R}$, but an equivalent description can be obtained with the coefficients' pair made of the Young modulus (E) and the Poisson ratio (ν); in principle we allow the two coefficients to be different constants on the two subdomains Ω_1 and Ω_2, even if in some applications we are interested in studying a crack inside a single material, i.e. a problem where these physical coefficients are uniform on Ω.

The differential problem reads as follows:

$$\begin{aligned} -\nabla \cdot \underline{\sigma}_1(\mathbf{u}_1) = \mathbf{f} \quad \text{in } \Omega_1, \quad -\nabla \cdot \underline{\sigma}_2(\mathbf{u}_2) = \mathbf{f} \quad \text{in } \Omega_2 \\ [\![\mathbf{u}]\!] = \mathbf{q}_0 \quad \text{on } \Gamma \\ [\![\underline{\sigma}(\mathbf{u}) \cdot \boldsymbol{\nu}_1]\!] = \mathbf{q}_1 \quad \text{on } \Gamma \\ \mathbf{u}_1 = \mathbf{g}_1 \quad \text{on } \Gamma_1, \quad \mathbf{u}_2 = \mathbf{g}_2 \quad \text{on } \Gamma_2 \end{aligned} \tag{4}$$

For $k = 1, 2$ $\mathbf{u}_k = \mathbf{u}|_{\Omega_k}$ and $\underline{\sigma}_k(\mathbf{u}_k)$ is the stress tensor associated to the plate's section Ω_k defined as:

$$\underline{\sigma}_k(\mathbf{u}_k) = 2\mu_k \, \underline{\epsilon}(\mathbf{u}_k) + \lambda_k \nabla \cdot \mathbf{u}_k \, I \qquad \text{where} \quad \underline{\epsilon}(\mathbf{u}_k) = \frac{1}{2}(\nabla \mathbf{u}_k + \nabla \mathbf{u}_k^T) \text{ and } I \in \mathbf{R}^{2\times 2}$$

are the symmetric part of the displacement's gradient and the identity tensor, respectively. As usual, given a vector $\mathbf{w} \in \mathbf{R}^2$ we denote its jump across the interface Γ with $[\![\mathbf{w}]\!] = \mathbf{w}_2 - \mathbf{w}_1$.

In the most common formulations, then, the interface conditions are expressed in terms of normal (subscript n) and tangential (subscript t) components with respect to the curve Γ itself, once a right-handed reference frame has been defined on it. In particular, the usual interface conditions are the following:

$$[u_t] = q_0, \quad [u_n] = 0, \quad [(\underline{\sigma}(\mathbf{u}) \cdot \boldsymbol{\nu}_1)_n] = 0, \quad [(\underline{\sigma}(\mathbf{u}) \cdot \boldsymbol{\nu}_1)_t] = 0 \quad \text{on } \Gamma,$$

which impose a continuous stress and a discontinuous tangential displacement.

Resorting to the general case, the **weak form** of problem (4) reads: for $\mathbf{f} \in \mathbf{L}^2(\Omega)$, $\mathbf{q}_0 \in \mathbf{H}^{r-1/2}(\Gamma)$, $\mathbf{q}_1 \in \mathbf{H}^{r-3/2}(\Gamma)$ and $\mathbf{g} \in \mathbf{H}^{r-1/2}(\Gamma_1 \cup \Gamma_2)$ with $1 \le r \le 2$, find $\mathbf{u} \in \mathbf{H}^1(\Omega_1 \cup \Omega_2)$ such that:

$$\begin{aligned} b(\mathbf{u}, \mathbf{v}) = (\mathbf{f}, \mathbf{v})_\Omega - \langle \mathbf{q}_1, \mathbf{v} \rangle_\Gamma, \quad \forall \ \mathbf{v} \in \mathbf{H}_0^1(\Omega) \\ [\![\mathbf{u}]\!] = \mathbf{q}_0 \text{ on } \Gamma, \quad \mathbf{u} = \mathbf{g} \text{ on } \partial\Omega \end{aligned} \tag{5}$$

where the continuous bilinear form $b(\cdot, \cdot) : \mathbf{H}^1(\Omega_1 \cup \Omega_2) \times \mathbf{H}^1(\Omega_1 \cup \Omega_2) \to \mathbf{R}$ associated to the linear elasticity operator is defined as $b(\mathbf{u}, \mathbf{v}) = b^1(\mathbf{u}, \mathbf{v}) + b^2(\mathbf{u}, \mathbf{v})$ with for $k = 1, 2$:

$$b^k(\mathbf{u}, \mathbf{v}) = \int_{\Omega_k} \frac{\mu_k}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T) : (\nabla \mathbf{v} + \nabla \mathbf{v}^T) + \int_{\Omega_k} \lambda_k (\nabla \cdot \mathbf{u}) \cdot (\nabla \cdot \mathbf{v}) \qquad \mathbf{u}, \mathbf{v} \in \mathbf{H}^1(\Omega_k).$$

Notice that the single terms $b^k(\cdot, \cdot)$ are symmetric, but the global bilinear form may be asymmetric if the Lamé coefficients for the two subdomains are different.

Finally, once we define the discrete interface and boundary data via the L^2 projectors introduced in Section 2 as:

$$\mathbf{g}_h := \mathbf{Q}_h\,\mathbf{g} \qquad \mathbf{q}_{0,h} := \mathbf{Q}_{h,\Gamma}\,\mathbf{q}_0$$

we are able to write the same finite element method as before.

The **approximate problem** reads as follows: in Ω find $\mathbf{u}_h \in \mathbf{V}_h$ such that

$$\begin{aligned} b(\mathbf{u}_h, \boldsymbol{\phi}) = (\mathbf{f}, \boldsymbol{\phi})_\Omega - \langle \mathbf{q}_1, \boldsymbol{\phi} \rangle_\Gamma \qquad \forall\ \boldsymbol{\phi} \in \mathbf{V}_h^0 \\ \text{with } [\![\mathbf{u}_h]\!] = \mathbf{q}_{0,h} \text{ on } \Gamma \quad \text{and} \quad \mathbf{u}_h = \mathbf{g}_h \text{ on } \partial\Omega, \end{aligned} \tag{6}$$

where $\mathbf{V}_h^0 = \mathbf{V}_h \cap \mathbf{H}_0^1(\Omega)$ and $\mathbf{f} = \mathbf{0}$ outside Ω. On the right hand side there are the usual inner products in $\mathbf{L}^2(\Omega)$ and $\mathbf{L}^2(\Gamma)$, respectively.

The same remarks on the treatment of the boundary and interface conditions, and the possibility of a solution discontinuous across Γ made for the elliptic equation still hold.

3 Algebraic formulation

In this section we will derive two different algebraic formulations that define the corresponding linear systems associated to the finite element method previously described. At first we will present the matrix coming directly from the discrete form, then we will rearrange its rows and the set of variables in order to reduce the dimension of the final system and to preserve the symmetry, when present.
For the sake of simplicity, all the details are outlined in the case of a simple Poisson problem defined on both domains, but they can be easily extended to the vectorial case when the Lamé coefficients are homogeneous on the whole Ω, i.e. $\lambda_1 = \lambda_2$ and $\mu_1 = \mu_2$. Problems involving different diffusion of Lamé coefficients on the subdomains require additional care, as clarified below.

Let us now specify the notation common to both algebraic formulations for the scalar problem. We denote by $\{\psi_j\}_{j=1}^M$ the basis for the discrete space V_h and by $\{\phi_i\}_{i=1}^N$ the basis for the discrete space V_h^0. Notice that, in order to allow the discontinuity of the solution on the interface, the degrees of freedom located on Γ are associated to functions ψ_j whose support is completely included either on Ω_1 or on Ω_2; in particular, since we only consider conforming grids, these couples of "doubled" dofs are placed at the same physical points on Γ.

The unknown function u_h, then, can be written as a linear combination of the $\{\psi_j\}_{j=1}^M$ with suitable coefficients that constitute the solution $\mathbf{u}^h \in \mathbf{R}^M$ of the associated linear system. In particular we can group the vector of unknowns as follows:

$$\mathbf{u}^h = \left[\mathbf{u}1,\ \mathbf{u}2,\ \mathbf{u}1_\Gamma,\ \mathbf{u}2_\Gamma\right]^T$$

where $\mathbf{u}k$ for $k = 1, 2$ denotes the set of degrees of freedom internal to Ω_k (i.e. whose shape functions have null trace on Γ), while $\mathbf{u}k_\Gamma$ denotes the set of dofs on Γ (i.e. whose shape functions have non zero trace on the interface and support contained in Ω_k).

It is then possible to associate a block of the classic global stiffness matrix to each combination of the different groups of shape functions:

- $A_{k,k}$ contains the terms linked to the test and trial functions associated to the dofs internal to Ω_k;
- $A_{k,\Gamma}$ contains the integrals on Ω_k involving a test functions associated to an internal degree of freedom and a trial function in $V_h(\Omega_k)$ referring to a dof on Γ;
- $A_{\Gamma,k}$ contains the integrals on Ω_k involving "internal" trial functions and "interface" test functions and is the transpose matrix of $A_{k,\Gamma}$; notice that in this case the support of the test function is not contained in Ω_k, but just intersecting, since the shape function is continuous on Ω;
- $A_{\Gamma,\Gamma}$ contains the interaction of functions of V_h^0 associated to the dofs on the interface; it can be equivalently written as $A_{\Gamma 1,\Gamma 1} + A_{\Gamma 2,\Gamma 2}$ where the numerical indices denote the restriction of the integral to the corresponding subdomain.

Finally, to describe the right hand side we must introduce the usual mass matrix M on the interface with terms:

$$M_{i,j} = \int_\Gamma \phi_i \, \phi_j,$$

and for $k = 1, 2$ the vectors **q1** with the coefficient of discrete condition on the conormal derivative on Γ (in the vectorial case for the linear elasticity, this vector will represent the normal stress), **q0** with the coefficients of the discrete interface condition on the solution, and $\mathbf{f}k$, $\mathbf{f}k_\Gamma$ defined starting from the volumetric source as:

$$\mathrm{f}\, k_i = \int_{\Omega_k} f \, \phi_i,$$

where the subscript is used for the integrals with functions associated to dofs on the interface.

3.1 Basic form: unknown solutions on the interface

In this first algebraic formulation the unknowns on the interface are the actual values on the solution on the two sides, i.e. **u**1 and **u**2.
The dimensions of the two discrete spaces V_h and V_h^0 are different ($M > N$) since the test functions in the method 3 are continuous on the interface, producing in the first place a rectangular linear system with more columns than rows; the additional constraints are retrieved by adding in a direct way the equations describing the jump q_0 of the solution. In other words, the interface Γ is treated as a Dirichlet boundary for Ω_1 and as a Neumann boundary for Ω_2, since the jump q_1 of the conormal derivative is considered only in this case.
Therefore the system employed in this "basic" method has dimension $M \times M$ and takes the following form:

$$\begin{bmatrix} A_{1,1} & \mathbf{0} & A_{1,\Gamma} & \mathbf{0} \\ \mathbf{0} & A_{2,2} & \mathbf{0} & A_{2,\Gamma} \\ \mathbf{0} & \mathbf{0} & -I & I \\ A_{\Gamma,1} & A_{\Gamma,2} & \mathbf{0} & A_{\Gamma 2,\Gamma 2} \end{bmatrix} \begin{bmatrix} \mathbf{u}1 \\ \mathbf{u}2 \\ \mathbf{u}1_\Gamma \\ \mathbf{u}2_\Gamma \end{bmatrix} = \begin{bmatrix} \mathbf{f}1 \\ \mathbf{f}2 \\ \mathbf{q0} \\ \mathbf{f}1_\Gamma + \mathbf{f}2_\Gamma - M\mathbf{q1} \end{bmatrix} \tag{7}$$

Even if each single block is symmetric, as in the particular case of the Poisson problem, the global system does not preserve this property, thus reducing the set of solvers where to choose the most suitable and efficient one. On the other side, this formulation can be directly generalized to problems that are not symmetric because of the presence of parameters that are discontinuous across the interface.

3.2 Symmetric form: unknown average on the interface

The second equivalent algebraic formulation is obtained via simple linear transformations of the variables and the equations and allows us to write a smaller linear system with N rows and N columns, which are the same dimensions as for the system associated to a classic Galerkin continuous method defined on the same grid for the whole Ω. Moreover, as already mentioned, it preserves the global symmetry when the coefficients are uniform over the whole computational domain.
To achieve this result we must start from a different point of view on the initial "rectangular" system: instead of adding more equations, we can reduce the set of variables noticing that the actual unknown on the interface is the average of the solutions u_1 and u_2, since the interface datum q_0 already provide information about their jump. Therefore, defining the average and jump of the vector $\mathbf{u}^h$ as follows

$$\{\mathbf{u}^h\} = \frac{1}{2}(\mathbf{u}1_\Gamma + \mathbf{u}2_\Gamma), \qquad [\![\mathbf{u}^h]\!] = \mathbf{u}2_\Gamma - \mathbf{u}1_\Gamma,$$

the final solution on the two sides of the interface can be reconstructed as:

$$\mathbf{u}1_\Gamma = \{\mathbf{u}^h\} - \frac{1}{2}[\![\mathbf{u}^h]\!], \qquad \mathbf{u}2_\Gamma = \{\mathbf{u}^h\} + \frac{1}{2}[\![\mathbf{u}^h]\!].$$

According to the notation previously introduced, the global linear system in this case takes the following form:

$$\begin{bmatrix} A_{1,1} & \mathbf{0} & A_{1,\Gamma} \\ \mathbf{0} & A_{2,2} & A_{2,\Gamma} \\ A_{\Gamma,1} & A_{\Gamma,2} & A_{\Gamma,\Gamma} \end{bmatrix} \begin{bmatrix} \mathbf{u}1 \\ \mathbf{u}2 \\ \{\mathbf{u}^h\} \end{bmatrix} = \begin{bmatrix} \mathbf{f}1 + \frac{1}{2}A_{1,\Gamma}\,\mathbf{q0} \\ \mathbf{f}2 - \frac{1}{2}A_{2,\Gamma}\,\mathbf{q0} \\ \mathbf{f}1_\Gamma + \mathbf{f}2_\Gamma - M\mathbf{q1} \end{bmatrix} \tag{8}$$

which can be easily derived from 7 by writing the set of unknowns on the interface in terms of their average and jump, and by performing a simple change of variables.
The system associated to this intermediate step is:

$$\begin{bmatrix} A_{1,1} & \mathbf{0} & A_{1,\Gamma} & -\frac{1}{2}A_{1,\Gamma} \\ \mathbf{0} & A_{2,2} & A_{2,\Gamma} & \frac{1}{2}A_{2,\Gamma} \\ A_{\Gamma,1} & A_{\Gamma,2} & A_{\Gamma 2,\Gamma 2} & \frac{1}{2}A_{\Gamma 2,\Gamma 2} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & I \end{bmatrix} \begin{bmatrix} \mathbf{u}1 \\ \mathbf{u}2 \\ \{\mathbf{u}^h\} \\ [\![\mathbf{u}^h]\!] \end{bmatrix} = \begin{bmatrix} \mathbf{f}1 \\ \mathbf{f}2 \\ \mathbf{f}1_\Gamma + \mathbf{f}2_\Gamma - M\mathbf{q}1 \\ \mathbf{q}0 \end{bmatrix}$$

where the last row can be moved to the right hand side since it simply describes a datum of the problem.

The same final system 8 can be retrieved from a different initial formulation, which is the one actually implemented by our code starting from the basis functions and degrees of freedom associated to the two discrete spaces $V_h(\Omega_1)$ and $V_h(\Omega_2)$. This implies the assembly of two independent linear systems, one for each subdomain, which can be rearranged in the global one as follows:

$$\begin{bmatrix} A_{1,1} & \mathbf{0} & A_{1,\Gamma} & \mathbf{0} \\ \mathbf{0} & A_{2,2} & \mathbf{0} & A_{2,\Gamma} \\ A_{\Gamma,1} & \mathbf{0} & A_{\Gamma 1,\Gamma 1} & \mathbf{0} \\ \mathbf{0} & A_{\Gamma,2} & \mathbf{0} & A_{\Gamma 2,\Gamma 2} \end{bmatrix} \begin{bmatrix} \mathbf{u}1 \\ \mathbf{u}2 \\ \mathbf{u}1_\Gamma \\ \mathbf{u}1_\Gamma \end{bmatrix} = \begin{bmatrix} \mathbf{f}1 \\ \mathbf{f}2 \\ \mathbf{f}1_\Gamma + \partial\mathbf{u}1 \\ \mathbf{f}2_\Gamma - \partial\mathbf{u}2 \end{bmatrix}$$

where with $\partial\mathbf{u}k$ for $k = 1, 2$ we denote the vector with the coefficients of the L^2 projection on $V_h(\Gamma)$ of the conormal derivatives with respect to $\boldsymbol{\nu}_1$.
As before, we perform the change of variables introducing the jump and average vectors on the interface:

$$\begin{bmatrix} A_{1,1} & \mathbf{0} & A_{1,\Gamma} & -\frac{1}{2}A_{1,\Gamma} \\ \mathbf{0} & A_{2,2} & A_{2,\Gamma} & \frac{1}{2}A_{2,\Gamma} \\ A_{\Gamma,1} & \mathbf{0} & A_{\Gamma 1,\Gamma 1} & -\frac{1}{2}A_{\Gamma 1,\Gamma 1} \\ \mathbf{0} & A_{\Gamma,2} & A_{\Gamma 2,\Gamma 2} & \frac{1}{2}A_{\Gamma 2,\Gamma 2} \end{bmatrix} \begin{bmatrix} \mathbf{u}1 \\ \mathbf{u}2 \\ \{\mathbf{u}^h\} \\ [\![\mathbf{u}^h]\!] \end{bmatrix} = \begin{bmatrix} \mathbf{f}1 \\ \mathbf{f}2 \\ \mathbf{f}1_\Gamma + \partial\mathbf{u}1 \\ \mathbf{f}2_\Gamma - \partial\mathbf{u}2 \end{bmatrix}$$

Finally, we replace the third block row with the sum of the last two block rows and the last one with their difference; after dividing by 2 the last block row we obtain:

$$\begin{bmatrix} A_{1,1} & \mathbf{0} & A_{1,\Gamma} & -\frac{1}{2}A_{1,\Gamma} \\ \mathbf{0} & A_{2,2} & A_{2,\Gamma} & \frac{1}{2}A_{2,\Gamma} \\ A_{\Gamma,1} & A_{\Gamma,2} & A_{\Gamma,\Gamma} & \mathbf{0} \\ -\frac{1}{2}A_{\Gamma,1} & \frac{1}{2}A_{\Gamma,2} & \mathbf{0} & \frac{1}{2}A_{\Gamma 1,\Gamma 1} \end{bmatrix} \begin{bmatrix} \mathbf{u}1 \\ \mathbf{u}2 \\ \{\mathbf{u}^h\} \\ [\![\mathbf{u}^h]\!] \end{bmatrix} = \begin{bmatrix} \mathbf{f}1 \\ \mathbf{f}2 \\ \mathbf{f}1_\Gamma + \mathbf{f}2_\Gamma + \partial\mathbf{u}1 - \partial\mathbf{u}2 \\ \frac{1}{2}\mathbf{f}2_\Gamma - \frac{1}{2}\mathbf{f}1_\Gamma - \frac{1}{2}\partial\mathbf{u}2 - \frac{1}{2}\partial\mathbf{u}1 \end{bmatrix}$$

Notice the introduction of the block $A_{\Gamma,\Gamma} = A_{\Gamma 1,\Gamma 1} + A_{\Gamma 2,\Gamma 2}$ and the arbitrary choice of writing $A_{\Gamma 1,\Gamma 1}$ at the bottom right corner, since it is equal to $A_{\Gamma 2,\Gamma 2}$ thanks to the use of a conforming mesh on Γ. By imposing the two interface data

$$[\![\mathbf{u}^h]\!] = \mathbf{q0} \quad \text{and} \quad \partial\mathbf{u}2 - \partial\mathbf{u}1 = M\mathbf{q1}$$

and moving the last block row to the right hand side we retrieve the linear system in 8.

As a last remark, we must point out that the same set of algebraic operations performed to ensure the symmetry of the global system fits similar interface problems with homogeneous coefficients on the two subdomains. They can be easily generalized to bilinear forms depending on a single parameter, constant but different on the two subdomains, as for of an elliptic problem with diffusion coefficients ε_1 and ε_2; in this case, indeed, the initial matrix is:

$$\begin{bmatrix} \varepsilon_1 A_{1,1} & \mathbf{0} & \varepsilon_1 A_{1,\Gamma} & \mathbf{0} \\ \mathbf{0} & \varepsilon_2 A_{2,2} & \mathbf{0} & \varepsilon_2 A_{2,\Gamma} \\ \varepsilon_1 A_{\Gamma,1} & \mathbf{0} & \varepsilon_1 A_{\Gamma 1,\Gamma 1} & \mathbf{0} \\ \mathbf{0} & \varepsilon_2 A_{\Gamma,2} & \mathbf{0} & \varepsilon_2 A_{\Gamma 2,\Gamma 2} \end{bmatrix}$$

and we can proceed by performing the same change of variables, multiplying the first and third block rows times $\frac{\varepsilon_2}{\varepsilon_1+\varepsilon_2}$ and the second and fourth block rows times $\frac{\varepsilon_1}{\varepsilon_1+\varepsilon_2}$, replacing the last two block rows with their sum and difference as before. After a further division by 2 of the last block row, and the imposition of the interface data, we can reduce the system obtaining a multiple (with coefficient $\frac{\varepsilon_1\varepsilon_2}{\varepsilon_1+\varepsilon_2}$ of the same matrix contained in 8.
The derivation of an equivalent symmetric form for operators depending on two or more (possibly non constant) non uniform coefficients, such as in the linear elasticity equation, requires additional care and goes beyond the scope of this work.

4 Implementation with `GetFEM++`

In order to implement and compare the two different approaches described in the previous sections, we used `GetFEM++ 5.3` (2018), an open source library that can be downloaded from its official website [3]. This library offers the perfect environment to solve systems of partial differential equations with finite element methods and is particularly suitable for contact problems, fluid-structure interactions and special boundary conditions. It also provides a `C++` template library for sparse and dense matrices called `Gmm++`, that includes several algorithms typical of a linear algebra library and generic solvers for linear systems.

`GetFEM++` has proven to be a very powerful tool because it allows to work with problems different dimensions, to choose from a large set of predefined integration methods, finite element methods, geometric transformations of the domain and to easily switch from one to another.
Moreover, one of the strongest point of the latest version is the opportunity to compile generic

assembly terms into optimized basic instructions, reducing the execution time of the matrices construction in most cases. This is what the users call a "high-level" assembly: to create a generic term one can simply write a string reflecting the weak formulation of the differential problem of interest, and then the library itself performs a symbolic differentiation if needed. The most natural way to apply this type of assembly involves the use of the *model* object, which globally describes a PDE model. Its role is to contain a list of variables and a list of *bricks* and to make them interact to produce a system of equations.
To know more about the usage of `GetFEM++` and its modern approach to a "weak form language", we refer you to the link of the documentation[1].

Although this may be very useful to treat nonlinear terms efficiently, for example, in our project it was not possible to employ these new features, because the *model* object prevents the user from accessing the final matrix of the linear system. This feature was a constraint in our case, given that we had to modify the structure of the linear system, in order to enforce the interface conditions, both in the direct and in the symmetric form.
On the other side, however, we have reasons to believe that this limitation did not have a strong impact on the performances of our code. Indeed we needed to assemble only simple linear terms in the problems we considered, action that can be efficiently performed by the "low-level generic assembly" mechanism relying on the pre-computation of the terms on the reference element.

4.1 Code structure

We will now go through the implementation details, describing the design of the code with its classes and methods and explaining the programming choices we made. For a complete documentation of the code, it is possible to generate a reference manual using *Doxygen*, as we will later show. Note that we are not going to report the entire classes with all their attributes and functions. Our goal is to show the most significant and notable ones that we used throughout the project avoiding, for example, the so-called `getter` and `setter` methods or the parts needed for *debugging*.
The global structure of the program comes from an existing code, which we have modified and improved according to our needs; in particular, besides smaller technical contributions to all classes, the major changes can be retrieved in the `Problem` class, with the related inheritance tree described in Subsection 4.2, and in the introduction of the `BulkDatum` class.

4.1.1 The `Bulk` class

First of all, we will refer briefly to the question of the construction of a square domain cut by an interface and to the solution we found to deal with this problem. The most natural choice to build a mesh with a crack would have been to exploit some features provided by the `GetFEM++` library, such as the functions `getfem::partial_mesh` or the attribute `getfem::level_set`.
However, what we first read in the documentation, and then verified trying to use these feature in the implementation, is that our problem is far from the standard environment needed to apply them effectively. The reason is that we want to construct a conforming mesh, aligned

[1] http://getfem.org/userdoc/gasm_high.html

with the interface Γ, whereas the `GetFEM++` tools are more suitable for cases where the interface cuts the elements of the mesh. Besides, the library does not work properly if the fault divides a triangular element in two parts such that one is much bigger than the other. We can understand that our problem was a limit case of this situation and that we were not able to take advantage of these instruments.

So finally, our choice was to pick a vertical crack dividing a square domain $\Omega = [0,1] \times [0,1]$, as shown in Figure 1, then consider each subdomain and build separately two meshes with the same number of elements, so that the resulting global one is aligned with the interface.
The class that handles all the properties related to the triangulation of the domain is called `Bulk`. Note that our implementation choice leads to the presence of two different objects of this class, namely `myDomainLeft` and `myDomainRight`.
Hereunder we report the constructor and some of its attributes.

```
1
2  // (... include ...)
3
4  class Bulk
5  {
6  public:
7     //constructor
8     Bulk ( const GetPot& dataFile, const std::string& section = "bulkData/",
9            const std::string& sectionDomain = "domain/",
10           const std::string& sectionProblem = "laplacian",
11           const std::string& domainNumber = "1");
12
13    void exportMesh(std::string const nomefile) const;
14
15 private:
16     // ( ...other attributes.. )
17
18     //dimension of the domain
19     size_type M_Nx;
20     size_type M_Ny;
21     scalar_type M_Lx;
22     scalar_type M_Ly;
23
24     std::string M_meshType;
25     getfem::mesh M_mesh;
```

Note that we have employed the object `getfem::mesh`, which builds the mesh of the domain according to the type of elements that we can specify in the `inputData` files and that is stored in the string `M_meshType`. Another possibility offered by the constructor is to load a mesh into `M_mesh` from a external file.

4.1.2 The `FEM` class

Once we have partitioned the domain and solved this issue, we can move on to explain the classes regarding the actual discretization of the partial differential equations. To deal with the several aspects of a PDEs system, we have chosen to construct a specific class for each of them. For instance, we will discuss in the following sections the `LinearSystem` class and the `BulkDatum` class which takes into account respectively the features connected to the resolution of the algebraic formulation and all the possible data regarding the problem.

Moreover, we have also been provided with a class named `FEM`, which is actually a wrapper of the already existing `getfem::fem` class. It incorporates only the instruments that we really needed to build a finite element method in our project, such as the choice of the problem's dimension, the definition of the degrees of freedom and the selection of the functional space for the trial and test functions.
Below we include an extract of the source file: in particular we show the constructor of the `FEM` class, based on the information received from the input file.

```
1
2  FEM::FEM ( const getfem::mesh& mesh,
3             const GetPot& dataFile,
4             const std::string& problem,
5             const std::string& variable,
6             const std::string& section,
7             const size_type qdim) :
8  M_section ( section + problem ), M_femType ( ), M_SpaceDim( qdim ), M_FEM(mesh)
9
10 {
11    //finite element method type, linear polynomials as default
12    M_femType = dataFile ( ( M_section+ "FEMType"+variable ).data (),
13                "FEM_PK(2,1)" );
14
15    getfem::pfem pf_v;
16    pf_v = getfem::fem_descriptor(M_femType);
17
18    // dimension of the problem
19    M_FEM.set_qdim(qdim);
20
21    // convex_index returns the list of all the valid convex elements on the mesh
22    M_FEM.set_finite_element(mesh.convex_index(), pf_v);
23
24    //definition of the degrees of freedom
25    for (size_type i=0; i<M_FEM.nb_dof();++i)
26    {
27      M_DOFpoints.push_back(M_FEM.point_of_basic_dof(i));
28    }
29
30 }
```

This class will be the key-point to handle the degrees of freedom both for the solution and for the right hand side, when constructing the final linear system.

4.1.3 The `BC` class

Finally, it is worth noticing that we used a separate class to treat the boundary conditions, which is totally independent from the PDEs system we want to solve. Here we illustrate some of the main attributes and methods of the `BC` class.

```
1
2  //(...include...)
3
4  class BC
5  {
6  public:
7      BC ( const GetPot& dataFile,
8           const std::string& problem,
```

```
9           const std::string& section );
10
11  // (... getter functions ..)
12
13      scalar_type BCNeum(const base_node& x, const size_type what,
14                         const size_type& flag) ;
15
16      scalar_type BCDiri(const base_node& x, const size_type what,
17                         const size_type& flag) ;
18
19  private:
20
21  // indices of the degrees of freedom on Neumann and Dirichlet boundaries
22      std::vector<size_type>  M_NeumRG;
23      std::vector<size_type>  M_DiriRG;
24
25  // (.. parser and other attributes ..)
26
27  };
```

Like most classes in the project, some of the private members of the `BC` class are read from input data files. In this particular framework, we always input four numbers, one for each side of the square domain: these numbers can be either 0, if we want the boundary to have Dirichlet conditions, or 1 to impose Neumann conditions. Once this information is read, the boundary degrees of freedom are grouped depending on whether they are on a Neumann or a Dirichlet edge and stored into two vectors of indices: `M_NeumRG` and `M_DiriDG`.
Besides the constructor and the "getters" for parameters, the main methods in the public interface of `BC` are the functions `BCNeum` and `BCDiri` that evaluate either the Neumann or the Dirichlet conditions on a given node of the domain, returning a scalar number.
Notice that we could not define these methods as `const`; the reason is that the `LifeV::Parser` object contained in the `BC` class changes its state when evaluating an expression. This issue is present throughout the whole project, and prevented us from adding the `const` qualifier to methods that would have needed it.

4.1.4 The `BulkDatum` class

What we will introduce now is the already mentioned `BulkDatum` class, which is a very simple but useful class to handle any kind of data related to the problem, from scalar coefficients to vectorial functions representing the source terms.

```
1  // (..include..)
2
3  class BulkDatum
4  {
5  public:
6   // (..constructor..)
7
8  scalar_type getValue(const base_node & x, const size_type what);
9
10 private:
11
12   std::string M_datum;
13   LifeV::Parser M_parser;
14   // ...other members to read input data
```

```
15
16 };
```

The method `getValue` receives as input the coordinates of the point where we want to evaluate the data and a constant index called `what`. The latter is used inside the method to recognize if the datum is a scalar (`what` $= 0$) or if it is a vector and we want to choose between the first component (`what` $= 0$) and the second one (`what` $= 1$).
Once again, as in the `BC` class, this method can not be defined `const` because of the parser.

4.1.5 The `OperatorsBulk` and `OperatorsBD` functions

In the header files `OperatorsBulk.h` and `OperatorsBD.h` we group all the methods related to the bulk, which are independent of the strategy we want to apply to solve the PDEs system. These are not classes but simple functions whose purpose is only to assemble useful operators that can be employed in several different contexts.
As we can see in the definitions below, these functions compute the stiffness matrix for the Laplacian Problem, the linear elasticity matrix, the volumetric source term, the exact solution, the jump across the interface and the natural boundary condition (`stressRHS`).

```
1
2 #ifndef OPERATORSBULK_H
3 #define OPERATORSBULK_H
4
5 void stiffness ( sparseMatrixPtr_Type M, const FEM& femSol, const FEM& femCoef,
6                  BulkDatum& Diff, const getfem::mesh_im& im);
7
8 void linearElasticity(sparseMatrixPtr_Type M, const FEM& femSol,
9                       const FEM& femCoef, BulkDatum& Mu, BulkDatum& Lambda,
10                      const getfem::mesh_im& im);
11
12 void bulkLoad (scalarVectorPtr_Type V, const FEM& FemSol, const FEM& FemSource,
13                BulkDatum& Source, const getfem::mesh_im& im);
14
15 void exactSolution (scalarVectorPtr_Type V, const FEM& FemSol,
16                     BulkDatum& Solution);
17
18 void jump(scalarVectorPtr_Type V, const FEM& FemSol, BulkDatum& Jump);
19
20 #endif
```

```
1 #ifndef OPERATORSBD_H
2 #define OPERATORSBD_H
3
4 void stressRHS( scalarVectorPtr_Type V, const Bulk& medium, BC& bcRef,
5                 const FEM& femSol, const FEM& femDatum,
6                 const getfem::mesh_im& im);
7
8 #endif
```

Observe that each function receives as input a pointer to a sparse matrix (`sparseMatrixPtr_Type`) or to a vector of scalar numbers (`scalarVectorPtr_Type`), where the result will be stored. It is worth mentioning that this choice was made to comply with other methods, in particular to the ones in the `LinearSystem` class, which will be described in details in the following.
An equivalent, if not even better, alternative would have been to use a more advanced language feature such as the references instead of pointers, but would have required as well major

changes in other parts of the code.
Each of these functions receives an object of the `BulkDatum` class, a constant integration method and obviously the degrees of freedom of the problem through a reference to a `FEM` class object.

We would like to highlight the difference between these methods: while the `exactSolution` and `jump` functions simply evaluate the corresponding `BulkDatum` on the points of the `FEM` object and store them, the others are wrappers of already existing functions provided by `GetFEM++` that automatically generate the matrices for the Laplacian problem and for the Linear Elasticity problem and also compute integrals for the source and the stress terms.
In the following, some parts from the source files are presented in order to show the use of `GetFEM++` facilities.

```
1
2  void stiffness ( sparseMatrixPtr_Type M,const FEM& FemSol, const FEM& FemCoef,
3                   BulkDatum& Diff, const getfem::mesh_im& im)
4  {
5
6  // ( ...definition of the getmesh::mesh_fem objects from the input FEM objects
   and evaluation of the diffusion coefficients ... )
7
8   getfem::asm_stiffness_matrix_for_laplacian(*M, im, femSol, femCoef, diff);
9
10 }
```

```
1
2  void linearElasticity(sparseMatrixPtr_Type M, const FEM& FemSol,
3                        const FEM& FemCoef, BulkDatum& Mu, BulkDatum& Lambda,
4                        const getfem::mesh_im& im)
5  {
6
7  // ( ... definition of the getmesh::mesh_fem objects from the input FEM objects
   and evaluation of the Lame' coefficients ... )
8
9   getfem::asm_stiffness_matrix_for_linear_elasticity(*M, im, femSol, femCoef,
10                                                     lambda, mu);
11
12  }
```

```
1
2  void bulkLoad(scalarVectorPtr_Type V, const FEM& FemSol, const FEM& FemSource,
3                BulkDatum& Source, const getfem::mesh_im& im)
4  {
5
6  // ( ... definition of the getmesh::mesh_fem objects from the input FEM objects
   and evaluation of the source term ... )
7
8  getfem::asm_source_term(*V, im, femSol, femSource, sourceVett);
9
10 }
```

```
1  void stressRHS( scalarVectorPtr_Type V, const Bulk& medium, BC & bcRef,
2                  const FEM& FemSol, const FEM& FemDatum,
3                  const getfem::mesh_im& im)
4  {
5
```

```
// (...definition of the getmesh::mesh_fem objects from the input FEM objects...)

 for ( size_type bndID = 0; bndID < bcRef.getNeumBD().size(); bndID++ )
    {
      scalarVector_Type neumVett(femDatum.nb_dof(), 0.0);
      for (size_type i = 0; i < femDatum.nb_dof(); i += Qdim)
      {
        for (size_type j = 0; j < Qdim; j++)
        {
          neumVett[i + j] = bcRef.BCNeum(femDatum.point_of_basic_dof(i), j,
          bcRef.getNeumBD()[bndID]);
        }
      } //recover the dofs on the Neumann boundary

      getfem::asm_source_term(*V, im, femSol, femDatum, neumVett,
      medium.getMesh().region(bcRef.getNeumBD()[bndID]) );
    }

}
```

4.1.6 The `LinearSystem` class

As anticipated in Section 3, we can always derive a linear system from the discretization of linear partial differential equations.
To treat and eventually modify the matrix and the right hand side according to the finite element method analyzed in this project, we will introduce a general class named **`LinearSystem`**.

```
// (.. include.. )

class LinearSystem
{
public:
     LinearSystem ();

     void copySubMatrix(sparseMatrixPtr_Type source, int first_row,
     int first_column, scalar_type scale=1.0, bool transpose=false) ;

     void addSubMatrix(sparseMatrixPtr_Type source, int first_row,
     int first_column, scalar_type scale=1.0, bool transpose=false) ;

     void extractSubMatrix(sparseMatrixPtr_Type destination, int first_row,
     int number_rows, int first_column, int number_cols ) const ;

     void copySubVector(scalarVectorPtr_Type source, int first_row,
     scalar_type scale=1.0) ;

     void addSubVector(scalarVectorPtr_Type source, int first_row,
     scalar_type scale=1.0);

     void extractSubVector(scalarVectorPtr_Type destination, int first_row,
     std::string where="sol") const ;

     void extractSubVector(scalarVectorPtr_Type&  destination, int first_row,
     std::string where="sol") const ;

     void addSubSystem(LinearSystem* small, size_type shiftRows,
```

```
30      size_type shiftColumns);
31
32      void solve();
33
34      void saveMatrix(const char* nomefile="Matrix.mm") const;
35
36      void eliminateRowsColumns(std::vector<size_type> indexes);
37
38       // (... other methods, inline getters and setters functions...)
39
40  private:
41
42      sparseMatrixPtr_Type M_Matrix;
43      scalarVectorPtr_Type M_RHS;
44      scalarVectorPtr_Type M_Sol;
45      int M_ndof;
46
47  };
```

This class is endowed with different methods that let us copy, add or extract parts of the matrix, of the right hand side or of the solution vector. There are also functions to get and set values and to save the matrix with the `.mm` extension in order to visualize it in the `Matlab` software, if needed. But overall, it provides algorithms that invert the matrix and solve the linear system, which use features included in the `Gmm++` library. In our case, we made use of the solver `SuperLU`, that is one of the most suitable and efficient algorithm to solve a general sparse linear system.

The private members of this class, as well as the input arguments of the public methods, are mostly pointers to sparse matrices or pointers to vectors of scalars. These types have been defined in the header file `Core.h` which includes all the `GetFEM++` and `Gmm++` matrix/vector predefined types that we needed in our project. Below we outline some of them:

```
1  using bgeot::base_small_vector; //special class for small (dim < 16) vectors
2  using bgeot::base_node; // geometrical nodes (derived from base_small_vector)
3  using bgeot::scalar_type; // = double
4  using bgeot::size_type; // = unsigned long
5  using std::string;
6  using sparseVector_Type = gmm::rsvector<scalar_type>;
7  using sparseMatrix_Type = gmm::row_matrix<sparseVector_Type> ;
8  using sparseMatrixPtr_Type = std::shared_ptr<sparseMatrix_Type> ;
9  using scalarVector_Type = std::vector<scalar_type> ;
10 using scalarVectorPtr_Type = std::shared_ptr<scalarVector_Type> ;
11 using sizeVector_Type = std::vector<size_type> ;
12 using sizeVectorPtr_Type = std::shared_ptr<sizeVector_Type> ;
```

Among all the `LinearClass` methods, we are going to present in detail the one that eliminates rows and columns. The reason is that we will need this function in the "symmetric method", where we have to reduce the global dimension of the matrix, erasing the repeated degrees of freedom on the interface.

```
1
2  void LinearSystem::eliminateRowsColumns(std::vector<size_type> indexes)
3  {
4    sort(indexes.begin(),indexes.end()); // sort the indexes in ascending order
5
```

```
6   for (size_type k=0; k<indexes.size(); k++)
7   {
8   // loop on the indexes in descending order
9   size_type idx = indexes[indexes.size()-1-k];
10
11  sparseMatrixPtr_Type A1 =
12        std::make_shared<sparseMatrix_Type>(idx,idx);
13  sparseMatrixPtr_Type A2 =
14        std::make_shared<sparseMatrix_Type>(idx,M_ndof-idx-1);
15  sparseMatrixPtr_Type A3 =
16        std::make_shared<sparseMatrix_Type> (M_ndof-idx-1,idx);
17  sparseMatrixPtr_Type A4 =
18        std::make_shared<sparseMatrix_Type>(M_ndof-idx-1,M_ndof-idx-1);
19
20  extractSubMatrix(A1,0,idx,0,idx);
21  extractSubMatrix(A2,0,idx,idx+1,M_ndof-idx-1);
22  extractSubMatrix(A3,idx+1,M_ndof-idx-1,0,idx);
23  extractSubMatrix(A4,idx+1,M_ndof-idx-1,idx+1,M_ndof-idx-1);
24
25  cleanMAT();
26  M_Matrix->resize(M_ndof-1,M_ndof-1);
27  addSubMatrix(A1, 0, 0);
28  addSubMatrix(A2, 0, idx);
29  addSubMatrix(A3, idx, 0);
30  addSubMatrix(A4, idx, idx);
31
32
33  // elimination of the degrees of freedom from the RHS
34  scalarVectorPtr_Type RHS1=
35        std::make_shared<scalarVector_Type> (idx);
36  scalarVectorPtr_Type RHS2 =
37        std::make_shared<scalarVector_Type> (M_ndof-1-idx);
38
39  extractSubVector(RHS1, 0, "M_RHS");
40  extractSubVector(RHS2, idx+1, "M_RHS");
41
42  cleanRHS();
43  M_RHS->resize(M_ndof-1);
44  addSubVector(RHS1, 0);
45  addSubVector(RHS2, idx);
46
47  M_Sol->resize(M_ndof-1);
48
49  M_ndof=M_ndof-1;
50
51  }
```

First of all, observe that as input argument of the method there is a vector of integers (`size_type`): each of them represents the i-th row and the i-th column we want to discard, in order to maintain the squared shape of the matrix.
The first issue to face is that we can only eliminate one row and one column at a time, since they are not contiguous in general, so we cannot modify the matrix dimension and be sure at the same time that the numbering of the indexes stays unchanged. The idea to solve the problem was to sort this vector in descending order and start to erase the bottom row and the rightmost column. Doing so, at each iteration the indexes left in the input vector will certainly still refer to the same rows and columns as at the time of the function call.

Once we have fixed the sequence of the integers, we can take advantage of the other `LinearClass` methods. We keep track of the four pieces of the matrix, coming from the removal of a row and a column, with the four pointers to sparse matrix `A1, A2, A3, A4` using the `extracSubMatrix` method. Afterwards, we resize the matrix, reducing the dimension by one and we add the stored parts in their new position, through the `addSubMatrix` method.
Finally we apply an analogous algorithm for the corresponding rows in the right hand side of the system. The only difference is that here we deal with pointers to vector of scalars instead of pointers to sparse matrix.

Therefore, we can deduce that there is almost no difference between the two methods or at least, we can state this for a 2-dimensional problem, where the size of the global matrix is reduced only by the number of degrees of freedom on a 1-dimensional fault.

4.2 Classes for problems and algebraic formulations

Now we have all the tools and the operators to introduce the main features for the management of the problems we want to investigate.
We built a hierarchy of classes to take into account all the shared and common methods and avoid useless repetitions. Let us start with the definition of an abstract class called `Problem` from which two other abstract classes (`BasicMethod` and `SymmetricMethod`) inherit. Then we specialize each of them with two classes characterized by the `final` keyword, one for the scalar diffusion problem (`LaplacianBasic` and `LaplacianSymmetric`) and one for the vectorial linear elasticity problem (`LinearElasticityBasic` and `LinearElasticitySymmetric`) as shown in the inheritance tree below (see Figure 2).

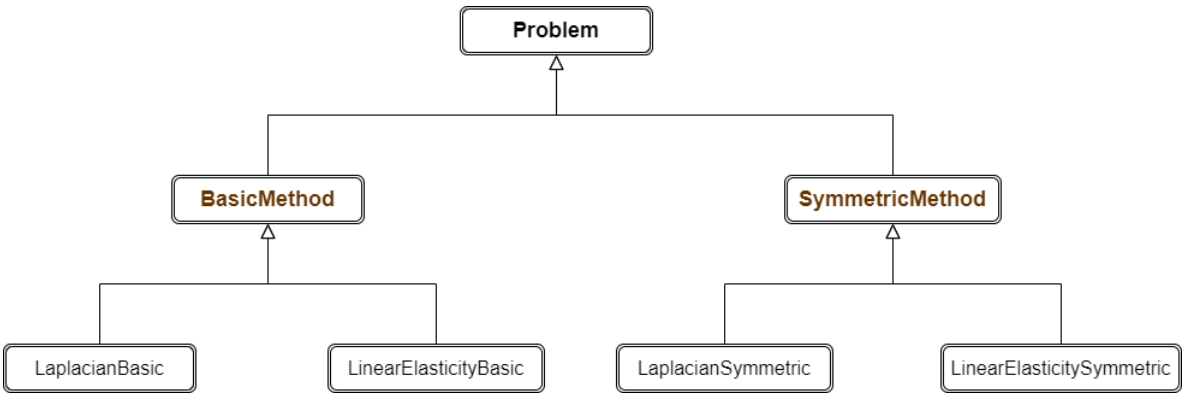


Figure 2: Inheritance tree diagram

The following Subsections are devoted to the detailed analysis of each of them.

4.2.1 The `Problem` abstract class

The `Problem` abstract class is the base class in our hierarchy. It contains all the methods and attributes that both the "symmetric" and the "basic" approach need without specialization and leaves the more peculiar functions as `virtual`. Naturally functions that extract the solution and compute or print errors in the L^2 and H^1 norm can be shared by any kind of finite element method.

```
//(..include..)

class Problem
{
public:
  Problem(  GetPot const & dataFile, std::string const problem, Bulk & bulk1,
  Bulk & bulk2, const size_type dim, LinearSystem & extSys); // constructor

  virtual void assembleMatrix() = 0;
  virtual void assembleRHS() = 0;
  virtual void enforceStrongBC(size_type const domainIdx) = 0;
  virtual void treatIFaceDofs() = 0;
  virtual void solve() = 0;

  void extractSol(scalarVector_Type & destSol,
                  std::string const variable = "all");
  void exportVtk(std::string const folder = "./vtk",
                 std::string const what = "all");
  void computeErrors();

  // other getter functions ...

  virtual ~Problem(){}; // destructor
```

Obviously the methods which deeply change according to the PDEs system and the chosen algebraic formulation are the ones for the construction of the matrix, of the right-hand-side and the treatment of the interface conditions. However, at this point it may not be so evident why we identified `enforceStrongBC` and `solve` as `virtual` methods. Briefly we can say that since the set of unknowns are different (i.e. it contains the average in the symmetric method), also the imposition of the boundary conditions are different; moreover, in the function to solve of the linear system we will have the reconstruction of the values on the interface for the symmetric approach. We will explain in detail these methods with their analogies and dissimilarities in the following sections.

```

protected:
  Bulk & M_Bulk1;
  Bulk & M_Bulk2;

  BC M_BC1, M_BC2;

  FEM M_uFEM1, M_uFEM2;
  FEM M_CoeffFEM1, M_CoeffFEM2;
  getfem::mesh_im M_intMethod1, M_intMethod2;

  LinearSystem& M_Sys;

  BulkDatum M_exact_sol1, M_exact_sol2, M_source1, M_source2;
  scalarVector_Type M_uSol, M_uSol1, M_uSol2;

  scalar_type errL2, errH1;
};
```

As it is common in a hierarchy, we define some attributes as `protected`, so that also the derived classes have access to them.
Note that the `LinearSystem` and the two `Bulk` objects are allocated as references because

they are initialized in the `main` function, whereas the spaces for the finite element method are created as members of the `Problem` class itself.
The only data that are independent from the specific problem are the exact solution and the forcing term, stored as `BulkDatum` objects.

4.2.2 The `BasicMethod` abstract class

In the `BasicMethod` abstract class, as well as in the `SymmetricMethod` one, we specialize the methods related to the construction of the right-hand-side (`assembleRHS`), to the imposition of the boundary condition (`enforceStrongBC`), to the treatment of the interface nodes (`treatIFaceDofs`) and to the resolution of the linear system (`solve`). These two classes are still considered `abstract` because we keep the assembly of the matrix as a `virtual` function: it will be modified in the `final` classes, depending on the PDEs problem we want to solve.

Let us analyze in a more accurate way the overridden methods.

```
1  void BasicMethod::assembleRHS()
2  {
3    // Volumetric source term
4    scalarVectorPtr_Type source1 = std::make_shared<scalarVector_Type> (M_nbDOF1);
5    bulkLoad(source1, M_uFEM1, M_uFEM1, M_source1, M_intMethod1);
6
7    scalarVectorPtr_Type source2 = std::make_shared<scalarVector_Type> (M_nbDOF2);
8    bulkLoad(source2, M_uFEM2, M_uFEM2, M_source2, M_intMethod2);
9
10     // Put the source for Omega1 at the top of the rhs
11     M_Sys.addSubVector(source1, 0);
12     // Put the source for Omega2 at the bottom of the rhs
13     M_Sys.addSubVector(source2, M_nbDOF1);
14
15     // For each interface node...
16     for (size_type k = 0; k < M_nbDOFIFace; k++)
17   {
18         // Sum the values from Omega1 and Omega2
19         scalar_type newVal = (M_Sys.getRHS())->at(dof_IFace1[k]) +
20         (M_Sys.getRHS())->at(dof_IFace2[k] + M_nbDOF1);
21         // Set the new value of the interface dof on Omega2
22         M_Sys.setRHSValue(dof_IFace2[k] + M_nbDOF1, newVal);
23     }
24
25  // Set Neumann boundary conditions
26
27     scalarVectorPtr_Type BCvec1 = std::make_shared<scalarVector_Type> (M_nbDOF1);
28     stressRHS( BCvec1, M_Bulk1, M_BC1, M_uFEM1, M_uFEM1, M_intMethod1);
29
30     scalarVectorPtr_Type BCvec2 = std::make_shared<scalarVector_Type> (M_nbDOF2);
31     stressRHS( BCvec2, M_Bulk2, M_BC2, M_uFEM2, M_uFEM2, M_intMethod2);
32
33     M_Sys.addSubVector(BCvec1, 0);
34     M_Sys.addSubVector(BCvec2, M_nbDOF1);
```

In the assembly of the right-hand-side term, we made use of the `bulkLoad` function that described in Subsection 4.1.5 dedicated to the operators.
This method consists in constructing the source terms separately in the two subdomains and

adding the sub-vectors to the linear system, paying attention to chain them in the right position. Eventually, we get the values on the interface nodes of the left subdomain and we sum them to the corresponding ones coming from Ω_2. The reason of this procedure has been already explained in Section 3, Equation 7 when discussing the algebraic formulation of the "basic" approach.
Observe that the last part of the method is devoted to the treatment of the Neumann boundaries, exploiting the operator `stressRHS`. In fact, it is well-know from the theory that the stress condition contributes with a further integral in the right-hand-side of the weak formulation.

Now let us move on to discuss the strong imposition of the Dirichlet boundary condition, explaining the corrections needed for some rows of the matrix.

```
void BasicMethod::enforceStrongBC(size_type const domainIdx)
{
// (... Set local variables according to the current domain and recover the DOFs
on the Dirichlet boundaries ... )

        // For each dof on the Dirichlet boundaries
        for (size_type i = 0; i < M_rowsStrongBCcur.size(); i += Qdim)
        {
            size_type ii;
            ii = M_rowsStrongBCcur[i] ;

            // For each dimension of the problem
            for(size_type j = 0; j < Qdim; j++)
            {
              // Get coordinates of the node
              bgeot::base_node where =
                M_uFEMcur->getFEM().point_of_basic_dof(ii + j);
              // Evaluate the Dirichlet condition
              scalar_type value =
                M_BCcur->BCDiri(where, j,
                                M_BCcur->getDiriBD()[M_rowsStrongBCFlagscur[i]]);

              M_Sys.setNullRow(ii + j + (domainIdx==1 ? 0 : M_nbDOF1 ));
              // Set the diagonal value to 1
              M_Sys.setMatrixValue(ii + j + (domainIdx==1 ? 0 : M_nbDOF1 ),
                                   ii + j + (domainIdx==1 ? 0 : M_nbDOF1 ), 1);

             // Set value on the rhs
              M_Sys.setRHSValue(ii + j + (domainIdx==1 ? 0 : M_nbDOF1 ), value);
            }
        }
}
```

This algorithm has the purpose to strongly impose the Dirichlet boundary condition, in the sense that we modify the rows associated to the opportune degrees of freedom with all zeros. Subsequently we set the number 1 in the diagonal entries and change the right-hand-side accordingly, by computing the exact solution on the physical points. Note that one must take care of the indexes to consider, depending on the current subdomain.
The method is generalized for problems with any dimension, but observe that in the case of a scalar problem (`Qdim` $= 1$) the cycle on the j is reduced to a single iteration. This "trick" is often present in the project, because it allows to manage together the scalar and the vectorial problem, avoiding the rewriting of the same function for the two different cases. We were able

to implement it in this way given the ordering of the vectorial degrees of freedom imposed by `GetFEM++`: the library, indeed, groups the dofs associated to the two (or three) components of the function linked to a grid node, and then considers the ones of the next physical point.

We will now describe the most interesting method, that is also what indeed differentiates the "basic" from the "symmetric" form: the imposition of the interface conditions.

```
void BasicMethod::treatIFaceDofs(){

     // (.. recover of the interface dofs...)

    // For each interface node...
    for (size_type k = 0 ; k < M_nbDOFIFace ; k += Qdim){

        size_type idx1 = dof_IFace1[k];
        size_type idx2 = dof_IFace2[k] + M_nbDOF1;

        // For each dimension of the problem
        for (size_type j = 0; j < Qdim; j++)
        {
            // Get coordinates of the node
            bgeot::base_node where =
                M_uFEM1.getFEM().point_of_basic_dof(idx1 + j);

            // Change the matrix
            M_Sys.setNullRow(idx1 + j);
            M_Sys.setMatrixValue(idx1 + j, idx1 + j, 1);
            M_Sys.setMatrixValue(idx1 + j, idx2 + j, -1);

            // Evaluate the interface condition q0 and set the rhs value
            scalar_type value =
               M_BC1.BCDiri(where, j, M_BC1.getDiriBD()[idxIFaceInDiriBD]);
            M_Sys.setRHSValue(idx1 + j, value);

        }

    }
}
```

Similarly to the enforcement of the Dirichlet conditions, the `treatIFaceDofs` method of the `BasicMethod` class edits some rows, setting blocks of identity matrices (compare Section 3, Equation 7). We would like just to point out that here the jump of the solution across the crack is computed by taking the difference between the left and the right solution, in opposition to what stated in the theoretical analysis.

Finally, we present an extract from the `solve` method, to facilitate the comparison with the corresponding method of the class `SymmetricMethod` described in the following.

```

void BasicMethod::solve()
{
  M_Sys.solve();

  //Store the solution contained in LinearSystem in the vector of Problem
  M_Sys.extractSubVector(M_uSol, 0, "sol");
```

```
}
```

This function only solves the linear system making use of the `SuperLU` solver of the `Gmm++` library, then extracts and stores the solution in the variable of the `Problem` class.
As anticipated, we will see in the corresponding method of the `SymmetricMethod` class that this part needs to be changed in order to reconstruct the solution on the two sides of the interface.

4.2.3 The `SymmetricMethod` abstract class

As regards the `SymmetricMethod` abstract class, we will present the main differences in the implementation of the methods with respect to the corresponding ones in the `BasicMethod` class. Once again, only the `assemblyMatrix()` stays `virtual`, while all the other methods have been overridden.
We decided to link the unknown averages to the interface degrees of freedom coming from Ω_2 and the jump to the ones coming from Ω_1. Thus in many methods the two domains will be treated differently, as the interface dofs in Ω_1 will be eventually discarded to reduce the system.

To start with, we point out that in this class we will make an extensive use of the operator `jump`, which was not at all involved in the basic method. This is the reason why we decided to initialize two pointers to vector of scalars in the constructor of the class. These two variables store the computed jump q_0 or $\mathbf{q}_0$ in the physical points on both domains, as shown below.

```

SymmetricMethod::SymmetricMethod(GetPot const & dataFile,
                                 std::string con
                                 st problem, Bulk & bulk1, Bulk & bulk2,
                                 const size_type dim, LinearSystem & extSys):
    Problem(dataFile, problem, bulk1, bulk2, dim, extSys),
    M_jump1(dataFile, "bulkData/", problem, "1", "qzero"),
    M_jump2(dataFile, "bulkData/", problem, "2", "qzero")
    {
        // Initialize variables storing the q0 values on both subdomains
        scalarVectorPtr_Type q01 = std::make_shared<scalarVector_Type>(M_nbDOF1);
        scalarVectorPtr_Type q02 = std::make_shared<scalarVector_Type>(M_nbDOF2);
        jump(q01, M_uFEM1, M_jump1);
        jump(q02, M_uFEM2, M_jump2);
        M_q01 = *q01;
        M_q02 = *q02;

    }
```

Now that we defined what the vectors `M_q01` and `M_q02` represent, we can proceed to show and comment the overridden methods.
Regarding the assembly of the right-hand-side term (which happens after the construction of the global matrix in 8), the difference arises in the rows corresponding to the internal nodes, where we must add two additional expressions. Observe in the algebraic formulation 8 that we need to recover some blocks of the assembled matrix: we get those values whose row index corresponds to an internal node and whose column index corresponds to an interface node; then we multiply these terms times the jump evaluated on the interface points. Below we show how we proceeded in the implementation of the code.

```
void SymmetricMethod::assembleRHS()
{
  // (... same part of the Basic Method to construct source1 and source2 ... )

  //For each dof in Omega1...
  for (size_type i = 0; i < M_nbDOF1; ++i)
  {
        scalar_type value1 = 0;
        // If NOT on the interface modify the rhs
        if (index_inside(i, dof_IFace1) == 0)
        {
            for (size_type j = 0; j < M_nbDOFIFace; ++j)
            {
                // Compute the additional term
                value1 +=
                (*(M_Sys.getMatrix()))(i, dof_IFace2[j] + M_nbDOF1)*
                M_q01.at(dof_IFace1[j]);
            }

        source1->at(i) -= value1/2;
    }
    }

  // For each dof in Omega2...
  for (size_type i=0; i < M_nbDOF2; ++i)
  {
        scalar_type value2 = 0;
        // If NOT on the interface modify the rhs
        if (index_inside(i, dof_IFace2) == 0)
        {
            for (size_type j=0; j < M_nbDOFIFace; ++j)
            {
                // Compute the additional term
                value2 +=
                (*(M_Sys.getMatrix()))(i + M_nbDOF1, dof_IFace2[j] + M_nbDOF1)*
                M_q02.at(dof_IFace2[j]);
            }

         source2->at(i)+= value2/2;
      }
  }

 // (.. same part of the Basic Method to add the Neumann boundary conditions ..)

 }
```

Notice that even with this second approach the implemented jump condition has the opposite sign with respect to the theoretical analysis, and thus the signs of the corrections on the right hand side terms are different from Equation 8.

Let us briefly analyze now the method for the imposition of the Dirichlet boundary conditions. The only difference appears in the treatment of the dofs associated to the two nodes that lay both on the fault and on the boundary of the domain, i.e. $(0.5, 0)$ and $(0.5, 1)$. Indeed, this algorithm requires the unknowns on the interface nodes to be the average of the solutions, instead of their actual value. This is why we need to define a different enforcement of the

Dirichlet condition for those points.
The average value can be recovered using the value of the solution in the right part of the domain and adding the jump divided by two (according to the definition of q_0 and $\mathbf{q}_0$ actually implemented), namely:

$$\{\mathbf{u}^h\} = \mathbf{u}2_\Gamma + \frac{1}{2}[\![\mathbf{u}^h]\!].$$

In the following the implementation details.

```

// inside the cycles ( see the method in the Basic class )

// Special treatment for the first and last node on the interface

    if(domainIdx==1) // No special treatment
    {
        M_Sys.setRHSValue(ii+j, value);
    }
    else
    {
        // If the current dof is NOT on the interface: no special treatment
        if (index_inside(ii+j,dof_IFace2) == 0)
        {
             M_Sys.setRHSValue(ii + j + M_nbDOF1, value);
        }
        else
        {
            // Add 1/2 of the jump
            M_Sys.setRHSValue(ii  + j + M_nbDOF1, value + M_q02.at(ii+j)/2);
        }
    }
```

It is important to mention that the functions in both methods are implemented knowing the order in which they are called from the `main`. Basically we build the global matrix and the complete forcing term, then we apply all the necessary changes according to the chosen method, we impose the boundary conditions and finally we treat the interface nodes. In this particular case, the function `treatIFaceDofs` removes the degrees of freedom corresponding to the interface points of Ω_1, so that the ones on the right domain will be linked to the "new" unknowns: the average of the solution.

Let us illustrate the very simple function which calls the `eliminateRowsColumns` method of the `LinearSystem` class described in the following Section 4.1.6.

```
void SymmetricMethod::treatIFaceDofs()
{
    // reduce the dimension of the system
    M_Sys.eliminateRowsColumns(dof_IFace1);
    M_nbTotDOF= M_nbDOF1 + M_nbDOF2 - M_nbDOFIFace;
}
```

Finally, we will highlight the section that needed to be added in the function `solve`. Once the linear system has been solved, we must reconstruct the actual values of the solution on both sides of the interface, starting from the jump (already given as datum) and the computed average, namely:

$$\mathbf{u}1_\Gamma = \{\mathbf{u}^h\} + \frac{1}{2}[\![\mathbf{u}^h]\!], \qquad \mathbf{u}2_\Gamma = \{\mathbf{u}^h\} - \frac{1}{2}[\![\mathbf{u}^h]\!].$$

```
1
2  // (... resolution of the linear system ... )
3
4  // Reconstruct the solution on Omega1
5  for (size_type j=0; j < M_nbDOFIFace; j++)
6    {
7       size_type idx = dof_IFace1[j];
8       // new value = average + jump/2
9       scalar_type value =
10         M_uSol.at(dof_IFace2[j] + M_nbDOF1 - M_nbDOFIFace + j) + M_q01.at(idx)/2;
11      // Insert the value in the correct place according to the global numbering
12      auto it= M_uSol.insert(M_uSol.begin() + idx, value);
13   }
14
15 // Reconstruct the solution on Omega2
16 for (size_type j=0; j< M_nbDOFIFace; j++)
17 {
18     // new value = average - jump/2
19     M_uSol.at(dof_IFace2[j] + M_nbDOF1) -= M_q02.at(dof_IFace2[j])/2;
20 }
21
22 // Update the dimension of the solution
23 M_nbTotDOF = M_uSol.size();
```

4.2.4 The `final` classes

To conclude this section dedicated to the implementation of the code, we recall the structure of the `final` classes (`LaplacianBasic`, `LaplacianSymmetric`, `LinearElasticityBasic` and `LinearElasticitySymmetric`).
As anticipated, here we specialized the assembly of the matrices and defined the dimension of the problem as a `static` public member, to be able to access this information even before instantiating any object of these classes, as it happens in the constructor. Moreover, we provided each of them with new `BulkDatum` objects that are peculiar of the problem to solve, namely the diffusion coefficients for the scalar PDEs system (one for each subdomain) and the Lamé parameters μ and λ (one pair for each subdomain) for the linear elasticity problem.

```
1
2  class LaplacianBasic final : public BasicMethod
3  {
4  public:
5    static const size_type Qdim = 1;
6
7    LaplacianBasic ( GetPot const & dataFile, Bulk & bulk1, Bulk & bulk2,
8    LinearSystem & extSys);
9
10   void assembleMatrix() override;
11
12 private:
13   BulkDatum diff1, diff2;
14
15 };
```

```
1
2  class LinearElasticityBasic final : public BasicMethod
```

```
{
public:
  static const size_type Qdim = 2;

  LinearElasticityBasic ( GetPot const & dataFile, Bulk  & bulk1, Bulk & bulk2,
  LinearSystem & extSys);

  void assembleMatrix() override;

private:
  BulkDatum mu1, lambda1, mu2, lambda2;

};
```

For the sake of simplicity, we will show in detail only the assembly matrix for the Laplacian problem with the two different formulations. The only difference with the vectorial problem is the use of the operator `linearElasticity` instead of the `stiffness` one (see section 4.1.5).

```

void LaplacianBasic::assembleMatrix()
{
  // Assemble the matrices separately on each subdomain
  sparseMatrixPtr_Type A1 =
                      std::make_shared<sparseMatrix_Type> (M_nbDOF1, M_nbDOF1);
  stiffness( A1, M_uFEM1, M_CoeffFEM1, diff1, M_intMethod1);

  sparseMatrixPtr_Type A2 =
                      std::make_shared<sparseMatrix_Type> (M_nbDOF2, M_nbDOF2);
  stiffness( A2, M_uFEM2, M_CoeffFEM2, diff2, M_intMethod2);


  // Place the matrices in the global system
  M_Sys.addSubMatrix(A1, 0, 0);
  M_Sys.addSubMatrix(A2, M_nbDOF1, M_nbDOF1);

  sparseMatrixPtr_Type curRow =
                std::make_shared<sparseMatrix_Type>(1, M_nbTotDOF);

  // For each interface node...
  for (size_type k = 0; k < M_nbDOFIFace; k++)
  {
    // Get the associated row in the block from Omega1
    M_Sys.extractSubMatrix( curRow, dof_IFace1[k], 1, 0, M_nbTotDOF );
    // Add the row to the terms from Omega2 associated to the dof linked
    // to the same grid node
    M_Sys.addSubMatrix(curRow, dof_IFace2[k] + M_nbDOF1, 0);

  }
}
```

This method is the exact computational counterpart of the matrix in Equation 7. We start by assembling the matrices separately in each subdomain in order to have all the matrix blocks available: $A_{k,k}$, $A_{k,\Gamma}$ and $A_{\Gamma,\Gamma}$ for $k = 1, 2$. Then we place these portions in the global system and sum the rows associated to the interface nodes in Ω_1 to the corresponding rows of the degrees of freedom in Ω_2, linked to the same physical points.
Note that to get the final matrix of the system contained in Equation 7, we need to call later

the methods to enforce the Dirichlet condition and the treatment of nodes laying on the fault. In this case, that is the basic form, the imposition of the interface conditions will introduce the block of identity matrices, that are not yet present after the execution of `assembleMatrix`.

Finally we compare this assembly with the one in the symmetric method, which instead corresponds to implementation of Equation 8.

```
1
2  void LaplacianSymmetric::assembleMatrix()
3  {
4    // (... Assembly of the matrices separately on each subdomain
5    as in the basic method ...)
6
7    // Construct the term A_Gamma,Gamma: for each interface node...
8      for (size_type k = 0; k < M_nbDOFIFace; k++)
9    {
10     // Get the values in A_Gamma1,Gamma1
11       size_type idx1 = dof_IFace1[k];
12     scalar_type value = (*A1)(idx1, idx1);
13
14     // Sum them to A_Gamma2,Gamma2
15     size_type idx2 = dof_IFace2[k];
16     (*A2)(idx2, idx2) += value;
17   }
18
19   // Place the matrices in the global system
20     M_Sys.addSubMatrix(A1, 0, 0);
21     M_Sys.addSubMatrix(A2, M_nbDOF1, M_nbDOF1);
22
23   sparseMatrixPtr_Type curRow1 =
24                std::make_shared<sparseMatrix_Type>(1, M_nbTotDOF);
25   sparseMatrixPtr_Type curCol2 =
26                std::make_shared<sparseMatrix_Type>(M_nbTotDOF, 1);
27
28   // Link the interface dofs from Omega1 to the interface dofs from Omega2
29
30   // For each interface node...
31   for (size_type k=0; k < M_nbDOFIFace; k++)
32     {
33     // Get the row from A_Gamma,1
34     M_Sys.extractSubMatrix( curRow1, dof_IFace1[k], 1, 0, M_nbTotDOF );
35     // Get the column from A_1,Gamma
36     M_Sys.extractSubMatrix( curCol2, 0, M_nbTotDOF, dof_IFace1[k], 1 );
37     // Add the row to the equation associated to the same node from Omega2
38     M_Sys.addSubMatrix(curRow1, dof_IFace2[k] + M_nbDOF1, 0);
39     // Add the column to include the dependence on the unknown average
40     M_Sys.addSubMatrix(curCol2, 0, dof_IFace2[k] + M_nbDOF1);
41   }
42
43 }
```

Here the assembly of the matrix is slightly different: in addition to the construction of the matrices in the two subdomains and the summation of the rows of interface nodes in Ω_1 to the corresponding ones in Ω_2, this method applies exactly the same operation for the columns connected to the interface nodes of the right domain in order to include the dependence on the unknown average. Moreover, the diagonal terms of the rows linked to the interface indexes of

the left subdomain are summed to the ones of the right domain.
Note that, after the execution of this method, the dimension of the global matrix is not yet reduced. We need to call the function `treatIFaceDofs` to discard the degrees of freedom linked to interface nodes on the left subdomain.

5 Compilation and execution instructions

The code was developed and run on `Ubuntu 18.04.2` and `Ubuntu 18.04.4` with the `g++` compiler, version 7.4.0. It is based on the library `GetFEM++`, version 5.3, which can be downloaded from the official website [3] and installed following the instructions provided by the developers. It may require the installation of other common libraries, such as BLAS, LAPACK and Qhull, if not already available; instructions are provided on the same website [3]. In particular, we used LAPACK version 3, BLAS version 3, and Qhull version 7.2.0.

The folder `pacs-elasticityequation` collects the whole code and the input data to run the tests. It contains four subfolders, associated to each combination problem/method we implemented, each of which is organised as follows: `inputData` contains the input files to perform the tests, `output_vtk` will store the mesh and the solutions in the `.vtk` format, `outputData` will store the matrices with the `.mm` extension and the files with the errors in the L^2 and H^1 norms. Each of them also contains a `main_test.cc` and a `Makefile` to compile, according to the `C++11` standard, the corresponding problem/method with the following options:

- `all` : builds the program with few output information at runtime and generating only the `.vtk` files;
- `clean` : as usual, deleting also all the output files in `output_vtk` and `outputData`;
- `debug` : builds the program with extensive output information at runtime and generating also the `.mm` and errors files;
- `optimised` : as for `all` with the `-O3` flag active;
- `test` : as for `optimised` generating also the `.mm` and errors files.

Additional instructions on how to compile the code, run the tests, write the input files and generate the documentation are provided in the `README` file.

6 Numerical tests

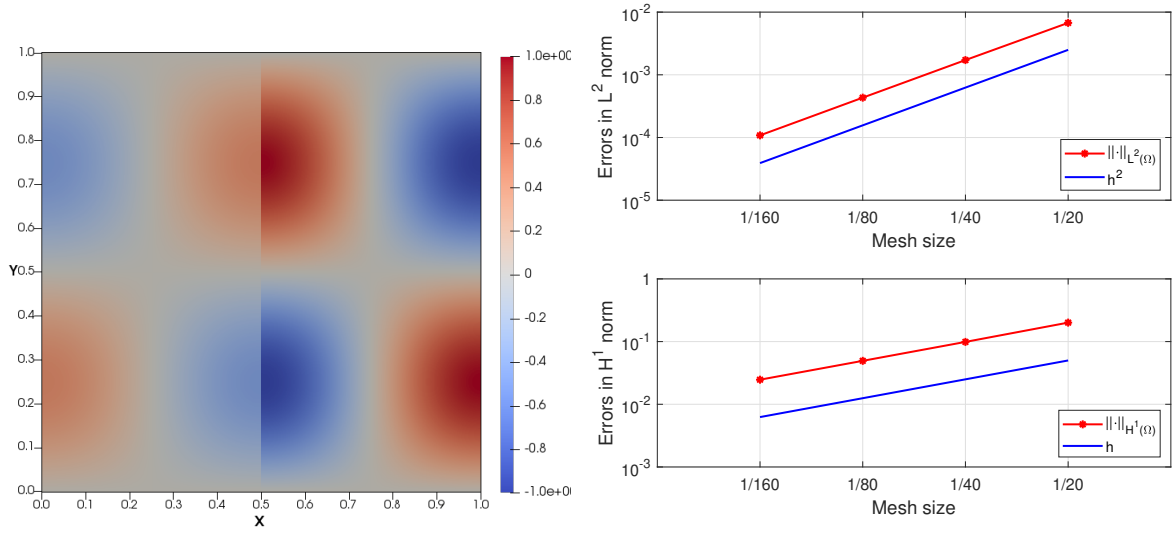
In this section we will illustrate the tests we performed both for the elliptic and the linear elasticity equation with the two methods. For each test we will show its data and the plot of the solution, extracted in a `.vtk` extension and visualized with the software `Paraview`. Moreover we add the graphs for the errors convergence, realized with `Matlab`, which will show the optimal order accuracy of the method under h-refinement.
In last subsection, we will make a comparison between the condition number of the matrices

and we will draw conclusions about the advantages and disadvantages of both methods.

We remind that all the numerical tests have been computed using strong Dirichlet conditions on each side of $\partial\Omega$, in order to avoid labile problems and that all the plots refer to a triangulation with 160×160 elements in the square domain.

6.1 Convergence analysis: elliptic equation

Test 1: the first numerical test we investigated is the resolution of the elliptic equation with two different diffusion coefficients ($\mu_1 = 2$, $\mu_2 = 1$), applying the basic method. Indeed, we recall that such a problem with discontinuous parameters can not be tested with the symmetric method, because the algebraic operations performed to obtain the final matrix assume the homogeneity of the coefficients. We saw a generalization with constant but different parameters in the theoretical part, in the Subsection 3.2.
The source term is the same for both the subdomains in order to model a realistic situation from the physical point of view; then the manufactured solution is computed consequently.

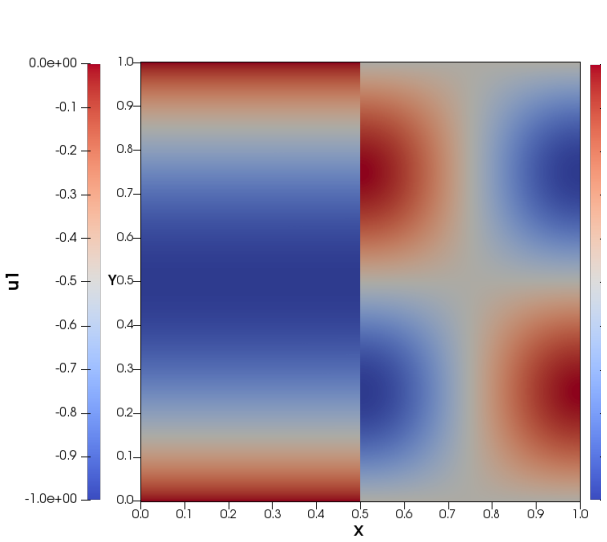


(a) Numerical solution visualized with `Paraview`.

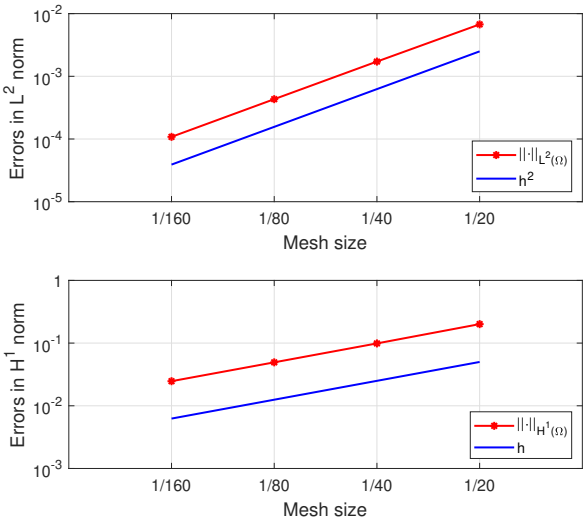
(b) Computed errors in the L^2-norm and H^1-norm versus the mesh size h (log-log scale).

Figure 3: TEST 1 Exact solution: $u = \frac{1}{2}\cos(2\pi x)\sin(2\pi y)$ on Ω_1 and $u = \cos(2\pi x)\sin(2\pi y)$ on Ω_2.

Test 2: in this test we chose a parabolic solution, with a periodic jump q_0 and q_1, and we employed the symmetric method: note that the optimal rate of convergence is achieved also in this case.



(a) Numerical solution visualized with `Paraview`



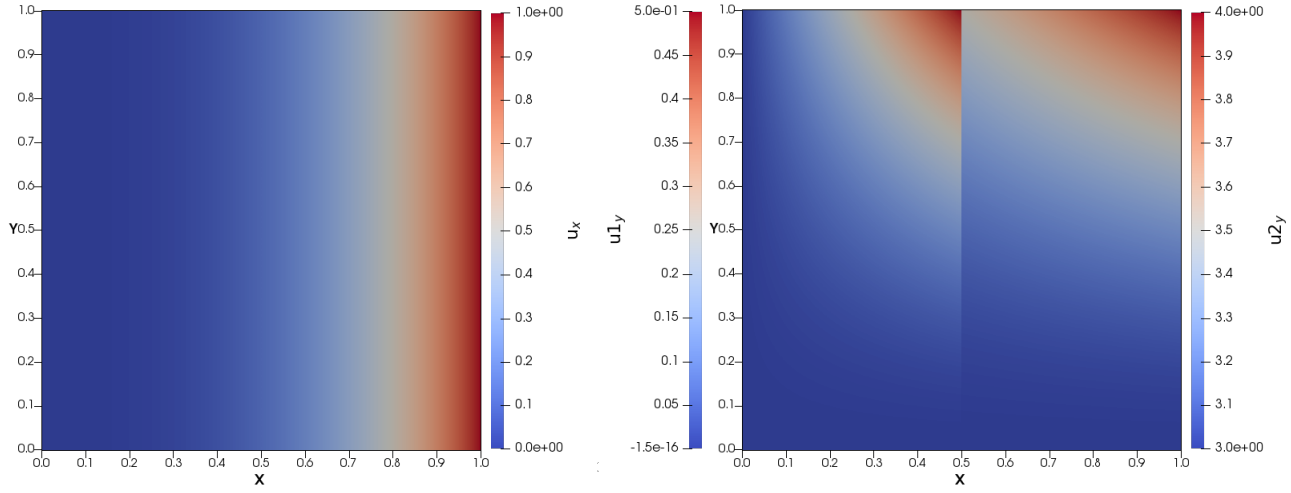
(b) Computed errors in the L^2-norm and H^1-norm versus the mesh size h (log-log scale).

Figure 4: TEST 2 Exact solution: $u = 4y(y-1)$ on Ω_1 and $u = \cos(2\pi x)\sin(2\pi y)$ on Ω_2.

6.2 Convergence analysis: linear elasticity equation

Now we will analyze two extra numerical tests for the linear elasticity equations, which are more significant from the modelling point of view. In both cases we suppose to have a homogeneous material, i.e. the Lamè parameters are uniform and constant on the whole Ω.

Test 3: this manufactured elastic displacement $\mathbf{u} \in \mathbf{R}^2$ has a discontinuity only in the x-direction, whereas the y-component is taken continuous, as well as the Cauchy stress tensor. The employed method is the basic one; also in the vectorial problem we retrieve the optimal rate of convergence.



(a) Numerical solution visualized with `Paraview` (x-component)

(b) Numerical solution visualized with `Paraview` (y-component)

Figure 5: TEST 3 Exact solution $\mathbf{u} = [x^3, y^2x]^T$ on Ω_1 and $\mathbf{u} = [x^3, y^2x + 3]^T$ on Ω_2.

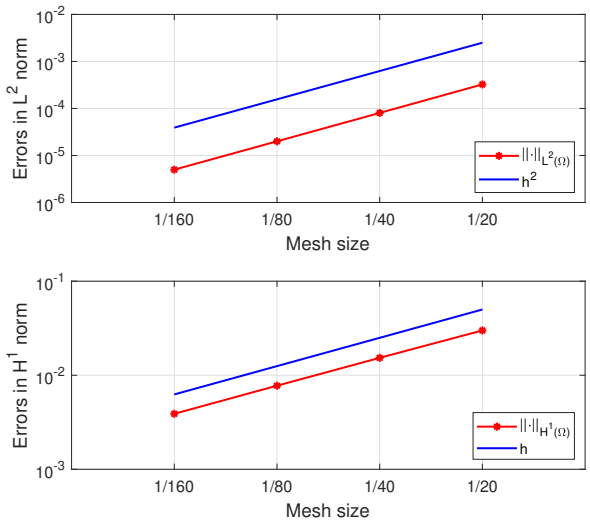
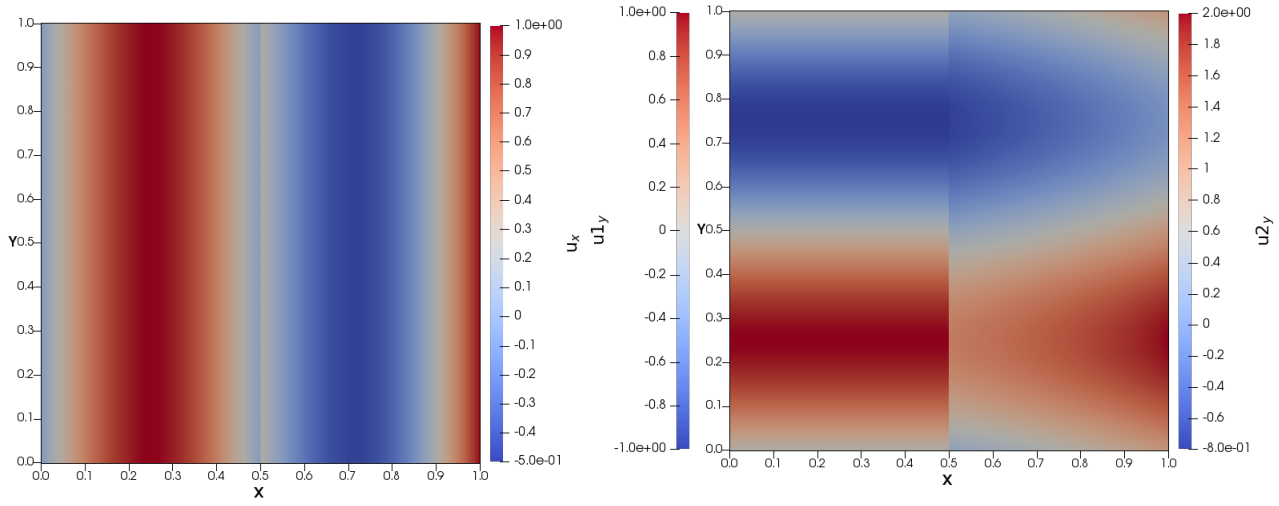


Figure 6: Computed errors in the L^2-norm and H^1-norm versus the mesh size h (log-log scale).

Test 4: The last presented test concerns again a linear elasticity model, but this time implemented with the symmetric method. Here both the solution and the normal stress feature a discontinuity along the interface Γ.



(a) Numerical solution visualized with `Paraview` (x-component)

(b) Numerical solution visualized with `Paraview` (y-component)

Figure 7: TEST 4 Exact solution: $\mathbf{u} = [\sin(2\pi x), \sin(2\pi y)]^T$ on Ω_1 and $\mathbf{u} = [x^2 + \sin(2\pi x), x^2 + \sin(2\pi y)]^T$ on Ω_2.

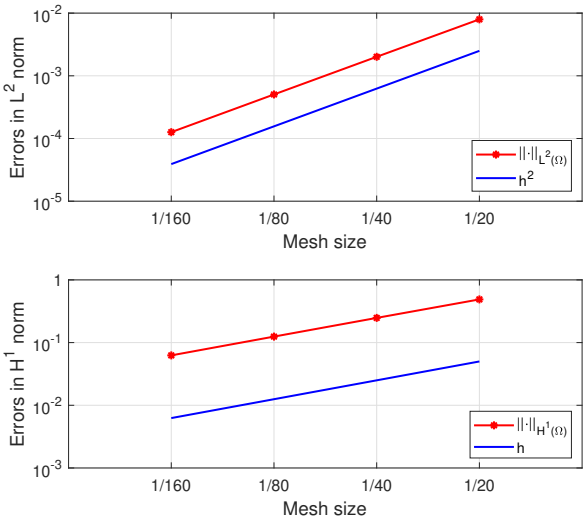


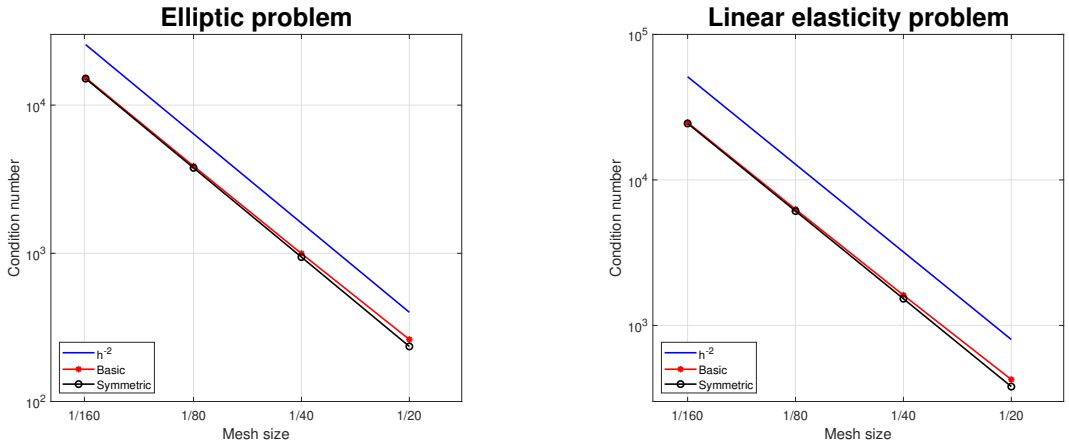
Figure 8: Computed errors in the L^2-norm and H^1-norm versus the mesh size h (log-log scale).

6.3 Condition number comparison

We complete the section regarding the numerical tests by computing the condition numbers of the implemented matrices and by showing the plots of their nonzero elements, via the `Matlab` function `spy`. In Figure 9 the global symmetry of the second method is evident, even if it is not

complete because of the imposition of the Dirichlet boundary conditions via the modification of the suitable rows. It is worth mentioning, however, that the complete symmetry could be retrieved by changing even the corresponding columns and moving the known datum to the right hand side.

As one can expect, the condition number of the matrices increases as h^{-2} (see Figure 10). Even if it is not so evident, for coarser meshes with the symmetric method, this number is slightly smaller. Therefore, we can deduce that there is almost no difference between the two methods or at least, we can state this for a 2-dimensional problem, where the size of the global matrix is reduced only by the number of degrees of freedom on a 1-dimensional fault. The values estimated with the `Matlab` function `condest` are remarkably large, especially for the finest grids; however we recall that the `SuperLU` solver tries to rebalance the matrix with permutations and scaling operations in order to ensure a solution as stable as possible.



(a) Trend of the condition numbers with the basic and the symmetric method for the laplacian problem

(b) Trend of the condition numbers with the basic and the symmetric method for the linear elasticity problem

Figure 10: Comparison of the condition numbers.

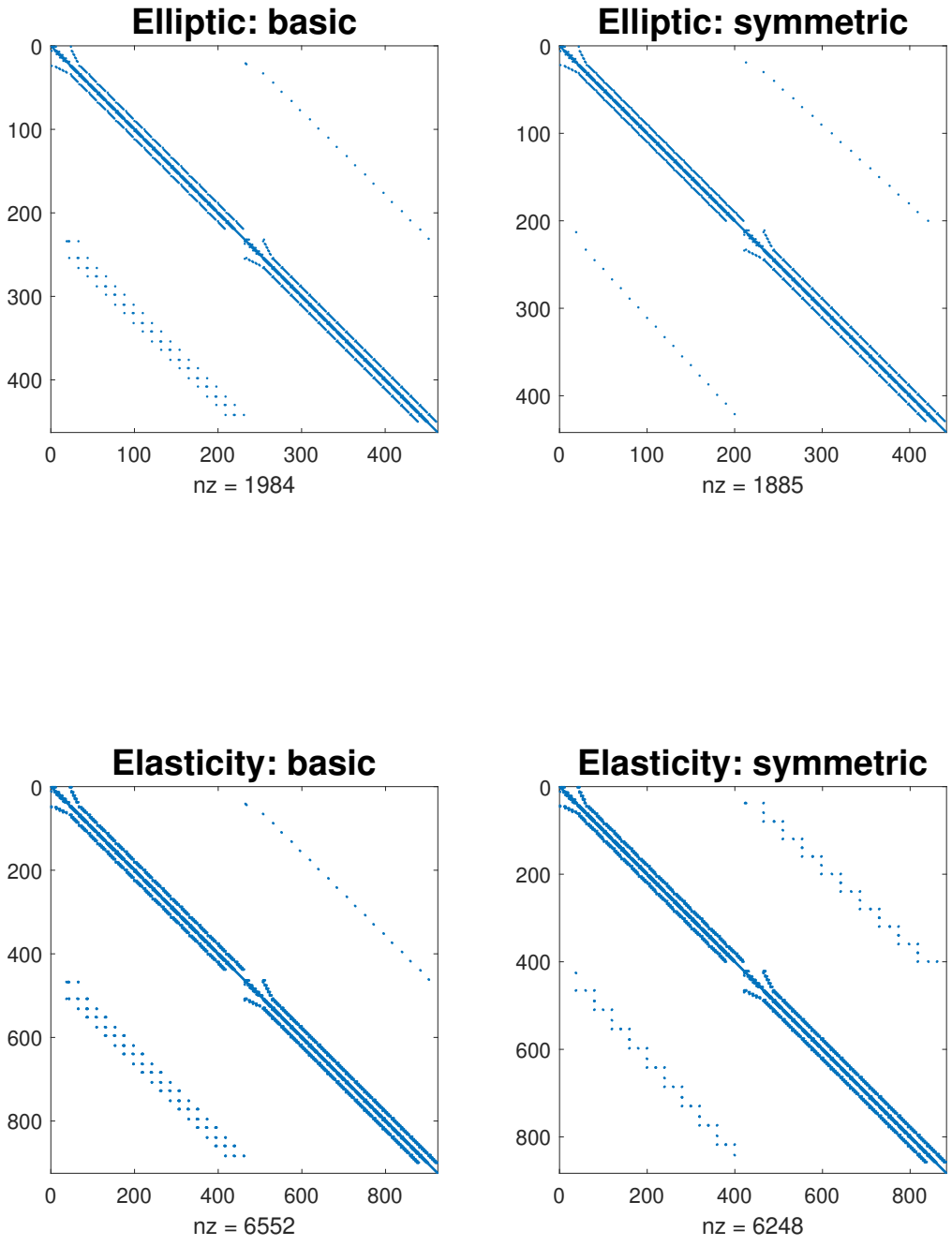


Figure 9: Sparsity patterns of the matrices for the laplacian and the linear elasticity problem with the number of non zero elements.

7 Conclusions and future work

In this project we verified the optimal order convergence of a Petrov-Galerkin finite element method to solve linear scalar and vectorial interface problems using a mesh conforming to the crack. Two equivalent algebraic formulations have been implemented exploiting the "low level assembly" of the `GetFEM++` library: even if the symmetric method defines a smaller system, we were not able to highlight remarkable differences in the performances of the two approaches. It might be interesting to test them on a 3D problem, where the reduced system may be much smaller than the original one, leading to a faster solution step; notice however that the assembly of the global symmetric matrix is longer than the one of the basic matrix, at least with the current implementation.

The same code, with some possible small modifications, can be employed to solve problems with Neumann boundary conditions on some parts of $\partial\Omega$ or with higher order finite elements defined in `GetFEM++`. Moreover, other different linear problems can be easily integrated in the hierarchy structure we designed, once the associated operators are defined.
Finally, the computational domain can be described by a different mesh built with some common external software, provided that it is conforming to the single interface. On the contrary, the current implementation can hardly be generalized to multiple fractures and non conforming grids, even if in this case more advanced and optimized features of the library could be exploited.

References

[1] J.H. Bramble, J.T. King, *A Finite Element Method for Interface Problems in Domains with Smooth Boundaries and Interfaces*, Advances in Computational Mathematics, 1996.

[2] P.F. Antonietti, C. Facciolà, A. Russo, M. Verani, *Discontinuous Galerkin Approximation of Flows in Fractured Porous Media on Polytopic Grids*, SIAM Journal on Scientific Computing, Vol. 41, No. 1, pp. A109-A138, 2019.

[3] J. Pommier, Y. Renard, http://getfem.org/index.html.