A 3D Mimetic Finite Difference code for Darcy problem in fractured porous media

Jacopo De Ponti

Politecnico di Milano
Master degree in Mathematical Engineering

APSC project

Febrary 2018

POLITECNICO
MILANO 1863

Contents

Introduzione **3**

1 Mathematical model **4**

1.1 Governing equations . . . 4

1.2 Weak formulation . . . 6

2 Numerical model **8**

2.1 Mimetic Finite Difference approximation . . . 8

2.1.1 The mesh and the degrees of freedom . . . 8

2.1.2 The mimetic divergence operator . . . 9

2.1.3 The mimetic inner products . . . 10

2.1.4 Discretization of the bullk problem . . . 13

2.2 Finite volume approximation . . . 14

2.3 Algebraic formulation . . . 16

2.4 Saddle point reduction . . . 18

2.5 Preconditioning . . . 18

3 The C++ code **21**

3.1 The numerical operators . . . 22

3.1.1 The class `global_Operator` . . . 23

3.1.2 The class `global_InnerProduct` . . . 25

3.1.3 The class `global_Div` . . . 27

3.1.4 The class `CouplingConditions` . . . 27

3.1.5 The class `FluxOperator` . . . 29

3.2 The builders of the system . . . 30

3.2.1 The class `global_BulkBuilder` . . . 30

3.2.2 The class `FractureBuilder` . . . 32

3.2.3 The class `SaddlePoint_StiffMatHandler` . . . 33

3.3 The preconditioners . . . 36

3.4 The solvers . . . 38

3.4.1 The class `Solver` . . . 39

3.4.2 The class `IterativeSolver` . . . 40

3.4.3 The class `imlGMRES` 41
3.5 The class `Problem` 42
3.6 Input parameters 45
3.7 Tutorial 47

4 Numerical results **51**
4.1 The single fracture test case 51
4.2 The fracture network test case 56

Conclusions **61**

Bibliography **62**

Introduction

The project consists of the implementation of a C++ code to solve a 3D Darcy problem in a fractured porous medium. The numerical method is an hybrid scheme that employs the Mimetic Finite Difference (MFD) method in the bulk and the Finite Volume (FV) method in the fractures. The Darcy problem in the bulk is treated in mixed form in order to compute the velocity, indeed an important advantage of the MFD with respect the FV is the possibility to have a better approximation of the velocity. Whereas in fractures a FV two-point flux approximation (TPFA) is used in order to have a simpler and flexible scheme, as a matter of fact the usage of FV in fractures allows to treat the fractures intersections in a relatively easy way.

The programming work starts from an existing fully FV software. In the code there was only a first simple implementation of the MFD in the bulk, but the bulk problem was treated in primal form. The main goals of the project have been: to implement the MFD scheme in a mixed form, employing a C++ object-oriented way of programming, and to implement an iterative solver with a proper preconditioner. This latter is really important in 3D problems the discretization of which normally gives rise to large linear systems.

The report is organized in the following way: the chapter 1 introduces the mathematical model in terms of governing equations and weak formulation, the chapter 2 describes the mimetic discretization of the bulk problem, the finite volume approximation of the fracture problem, the algebraic problem arising from the discretization and the preconditioner techniques, the chapter 3 describes in details the structure of the code and it shows a practical usage through a tutorial, finally the chapter 4 shows some numerical tests, in order to study the convergence order of the method with different types of grids and by varying the permeability and to show a realistic case with a complex network of fractures.

Chapter 1

Mathematical model

1.1 Governing equations

In this chapter we briefly introduce the mathematical model, starting from the model for the porous matrix and then describing the fracture model and the coupling conditions.

Let $\Omega \in \mathbb{R}^3$ be an open, bounded, convex and polyhedral domain representing the reservoir, which is considered a porous medium saturated by a liquid, e.g. water. The mathematical model consists of the Darcy law coupled with the mass conservation:

$$\begin{cases} \mathbf{u} = -\mathsf{K}\nabla p & in\ \Omega \\ \nabla \cdot \mathbf{u} = f & in\ \Omega, \end{cases} \tag{1.1}$$

where $\mathbf{u}$ is the velocity of the flow, p the pressure, K the symmetric positive and definite permeability tensor and f the source/sink term. We have assumed that the fluid and the medium are incompressible and we have also neglected the gravitational effects. We keep the model in mixed form because we want to approximate jointly both the pressure and the velocity.

The treatment of the fractures is more delicate because they have a much lower thickness with respect their length and the dimension of the reservoir Ω; moreover the fractures are tipically filled with a porous material that can differs a lot in terms of permeability with respect the one of the bulk. The usage of the model (1.1) also in the fractures would require a very refined grid in the zones of fracture with a resulting increment of the computational cost. So the thickness of the fractures is considered negligible and a reduced model is used. The reduced model that we consider is the one introduced in [9], to which we refer for the derivation and the details. In Figure 1.1 a sketch of the mathematical model is reported.

The fracture is modeled via a manifold $\Gamma \in \mathbb{R}^2$ and we assume only one fracture that can cut completely the domain or be partially or fully immersed; in any case the fracture Γ splits the domain in two subregion Ω_+ and Ω_- (for the partially of fully immersed fracture Γ must be extended to obtain this partition). The code is implemented to handle complex fracture networks but from a mathematical point of view we have considered only one fracture to ease the theoretical presentation. With $\mathbf{n}_\Gamma$ we indicate the unit normal to Γ, whose orientation must be considered fixed and it is arbitrary and with $\boldsymbol{\tau}_\Gamma$ the matrix whose columns form an orthonormal base for the tangent space at each $\mathbf{x} \in \Gamma$.

Let us define the bulk domain as $\Omega_\Gamma = \Omega \backslash \Gamma$. We consider the partition of the boundary of the domain $\partial\Omega = \Gamma_D \cap \Gamma_N$ and $\mathbf{n}$ as its unit outward normal. We prescribe a function g on the Dirichlet boundary Γ_D and a function h on the Neumann boundary Γ_N. The

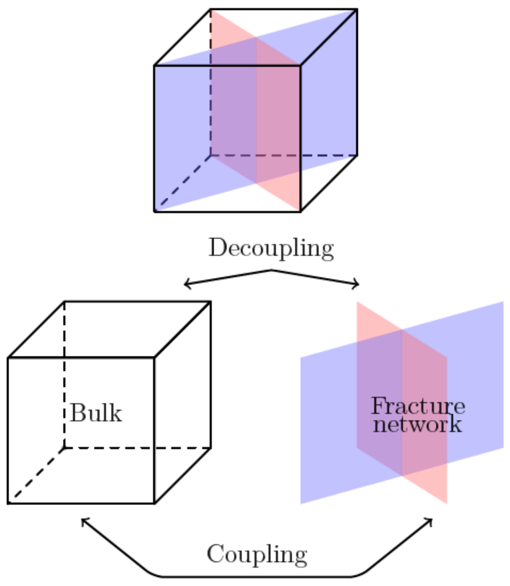

Figure 1.1: Sketch of the mathematical model.

model for the bulk with its boundary conditions is

$$\begin{cases} \mathsf{K}\nabla p + \mathbf{u} = 0 & in\ \Omega_\Gamma \\ \nabla \cdot \mathbf{u} = f & in\ \Omega_\Gamma \\ p = g & on\ \Gamma_D \\ -\mathsf{K}\nabla p \cdot \mathbf{n} = h & on\ \Gamma_N. \end{cases} \tag{1.2}$$

Regarding the fracture we indicate the aperture (i.e. the thickness) with $l_\Gamma = l_\Gamma(\mathbf{x})$ and we assume for the fracture flow a permeability tensor with the block diagonal structure

$$\mathsf{K}_\Gamma = \begin{bmatrix} k_\Gamma^n & 0 \\ 0 & k_\Gamma^\tau \end{bmatrix}, \tag{1.3}$$

where $k_\Gamma^\tau \in \mathbb{R}^{2\times 2}$ is a symmetric and positive definite tensor. Now let us define the jump and avarage of a function v (that must be regular enough) accross Γ:

$$[\![v]\!] = v_+ - v_-, \qquad \{v\} = \frac{1}{2}(v_+ + v_-), \tag{1.4}$$

where $v_i = v|_{\Gamma_i}$ with $i = \pm$ (Γ_i is the side of Γ from the part of Ω_i). Setting with p_Γ and $\mathbf{u}_\Gamma$ the pressure and the velocity in the fracture, the governing equations of the fracture flow are the mass balance and the Darcy law, written in tangential components. Both of them must be properly integrated across the thickness of the fracture and after some calculations they can be written as:

$$\begin{cases} k_\Gamma \nabla_\Gamma p_\Gamma + \mathbf{u}_\Gamma = 0 & in\ \Gamma \\ \nabla_\Gamma \cdot \mathbf{u}_\Gamma = l_\Gamma f_\Gamma + [\![\mathbf{u}\cdot\mathbf{n}_\Gamma]\!] & in\ \Gamma, \end{cases}$$

where $k_\Gamma = l_\Gamma k_\Gamma^\tau$, f_Γ is the source/sink term in the fracture and the differential operators have to be intended acting in the tangetial direction to the fracture Γ. Let us set $\partial\Gamma_N = \Gamma \cap \Gamma_N$, $\partial\Gamma_D = \Gamma \cap \Gamma_D$ and $\partial\Gamma_{N_0} = \partial\Gamma \setminus (\partial\Gamma_N \cup \partial\Gamma_D)$. Erasing by substitution the velocity variable to have a primal formulation, we obtain the model for the fracture:

$$\begin{cases} -\nabla_\Gamma \cdot (k_\Gamma \nabla_\Gamma p_\Gamma) = l_\Gamma f_\Gamma + [\![\mathbf{u}\cdot\mathbf{n}_\Gamma]\!] & in\ \Gamma \\ p_\Gamma = g & on\ \partial\Gamma_D \\ -k_\Gamma \nabla_\Gamma p_\Gamma \cdot \boldsymbol{\tau}_\Gamma = h & on\ \partial\Gamma_N \\ -k_\Gamma \nabla_\Gamma p_\Gamma \cdot \boldsymbol{\tau}_\Gamma = 0 & on\ \partial\Gamma_{N_0}. \end{cases} \tag{1.5}$$

Note that on the internal boundary of the fracture an homogenous Neumann condition is imposed, because we assume that the flow exchange between bulk and fractures happens most directly between the two domains.

Finally we provide the interface conditions to couple the bulk model (1.2) and the fracture model (1.5). Following [9], let $\xi \in [0,1]$, the coupling conditions are given by

$$\begin{cases} \frac{2\xi-1}{4}\eta_\Gamma [\![\mathbf{u}\cdot\mathbf{n}_\Gamma]\!] = \{p\} - p_\Gamma & on\ \Gamma \\ \eta_\Gamma \{\mathbf{u}\cdot\mathbf{n}_\Gamma\} = [\![p]\!] & on\ \Gamma, \end{cases} \tag{1.6}$$

where $\eta_\Gamma = l_\Gamma (k_\Gamma^n)^{-1}$. The second equation derives from the integration of the fracture Darcy law, written in the normal direction, while the first summarizes, by varying ξ, different possibities of closing the system; e.g. if $\xi = 1/2$ we are simply assuming that the pressure in the fracture is the albegraic avarage of the bulk pressures across the fracture.

Let us point out that in the case of a network of intersecating fractures the model is exactly the same of the one presented above, but we have to require in addition the continuity of the pressure and of the velocity accross the intersections.

1.2 Weak formulation

In this section we present a weak formulation of the bulk problem (1.2), because the mimetic discretization is carried out on the weak form of the governing equations. The coupling conditions (1.6) are introduced directly in the weak formulation. Whereas the fracture problem is kept in strong form because it will be discretized with FV and in this section we consider the pressure in the fracture p_Γ as given.

Let us introduce the following functional spaces:

$$\begin{aligned} &Q = L^2(\Omega_\Gamma) \\ &W = \{\mathbf{v} \in H(\mathrm{div}, \Omega_\Gamma) : \mathbf{v}\cdot\mathbf{n}_\Gamma \in L^2(\Gamma)\}, \end{aligned}$$

with the associated norms

$$\begin{aligned} &||q||_Q = ||q||_{L^2(\Omega_\Gamma)} \\ &||\mathbf{v}||_W^2 = ||\mathbf{v}||^2_{L^2(\Omega_\Gamma)} + ||\nabla\cdot\mathbf{v}||^2_{L^2(\Omega_\Gamma)} + ||\mathbf{v}\cdot\mathbf{n}_\Gamma||^2_{L^2(\Gamma)}. \end{aligned}$$

Note that we are requesting more regularity on the velocity $\mathbf{v}$ than $H(\mathrm{div}, \Omega_\Gamma)$; this is necessary to accomodate the coupling conditions (1.6) in the weak formulation. It can be shown that the spaces Q and W are Hilbert spaces with scalar products inducing the above stated norms. To handling the Neumann boundary conditions we define also the spaces:

$$\begin{aligned} &W_{g,\Gamma_N} = \{\mathbf{v} \in W : \mathbf{v}\cdot\mathbf{n} = g\ on\ \Gamma_N\} \\ &W_{0,\Gamma_N} = \{\mathbf{v} \in W : \mathbf{v}\cdot\mathbf{n} = 0\ on\ \Gamma_N\} \end{aligned}$$

Let us define the following bilinear forms:

$$a_\xi(\mathbf{u},\mathbf{v}) = \int_{\Omega_\Gamma} \mathsf{K}^{-1}\mathbf{u}\cdot\mathbf{v}\, dV + \int_\Gamma \eta_\Gamma \{\mathbf{u}\cdot\mathbf{n}_\Gamma\}\{\mathbf{v}\cdot\mathbf{n}_\Gamma\}\, dS + \xi_0 \int_\Gamma \eta_\Gamma [\![\mathbf{u}\cdot\mathbf{n}_\Gamma]\!][\![\mathbf{v}\cdot\mathbf{n}_\Gamma]\!]\, dS$$

$$B(\mathbf{u},q) = \int_{\Omega_\Gamma} (\nabla\cdot\mathbf{u})q\, dV,$$

where $\xi_0 = (2\xi - 1)/4$. We do not show the derivation of the weak formulation, for which we refer to [10]. In any case the weak formulation reads as follows: find $(\mathbf{u}, p) \in W_{g,\Gamma_N} \times Q$ such that

$$\begin{cases} a_\xi(\mathbf{u},\mathbf{v}) - B(\mathbf{v},p) + \displaystyle\int_\Gamma [\![\mathbf{v}\cdot\mathbf{n}_\Gamma]\!] p_\Gamma\, dS = -\int_{\Gamma_D} g(\mathbf{v}\cdot\mathbf{n})\, dS & \forall\, \mathbf{v}\in W_{0,\Gamma_N} \\ B(\mathbf{u},q) = \displaystyle\int_{\Omega_\Gamma} fq\, dV & \forall\, q \in Q. \end{cases} \tag{1.7}$$

To prove the well-posedness of the problem also the weak formulation of the fracture equation is done, here we don't lead this way because we present the mathematical model only as a starting point for the discretization. For the well-posedness of the problem in the case of a single fracture cutting the whole domain see [2]. We stress that also the weak formulation of the problem in the case of a fracture network is of the same form than the one presented above.

To summarize: the model that in the next chapter we discretize is given by (1.7), which takes into account the bulk problem and the coupling conditions, and by (1.5), which is the fracture problem.

Chapter 2

Numerical model

2.1 Mimetic Finite Difference approximation

Now we introduce the mimetic discretization of the bulk problem, i.e. of (1.7), assuming p_Γ given. We start describing the mesh and the mimetic spaces, then we introduce the projection operators and the mimetic divergence operator and finally we describe the construction of the mimetic inner products. For more details on the mimetic theory see [3].

2.1.1 The mesh and the degrees of freedom

Let Ω_h be a partition of Ω into non-overlapping polyhedras P, which are aligned with the fracture Γ. We consider the diameter of the mesh as $h = max_{\mathsf{P}\in\Omega_h} h_{\mathsf{P}}$, where h_{P} is the diameter of the element P. Being the polyhedra aligned with Γ, the mesh Ω_h can be partitioned into two sets of cells Ω_h^+ and Ω_h^- such that $\Omega_h = \Omega_h^+ \cup \Omega_h^-$. Moreover the mesh Ω_h induces a natural partition of Γ that we indicate with Γ_h and that is the fracture bidimensional grid of polygons.

In the mimetic framework the classical regularity assumptions are: there exist two positive real numbers N^s and ρ_s such that Ω_h admits a conforming subpartition T_h of tetrahedras such that:

AM1 every polyhedron $\mathsf{P} \in \Omega_h$ admits a decomposition $\mathsf{T}_h|_{\mathsf{P}}$ of at most N^s tetrahedras;

AM2 every tetrahedra $\mathsf{T} \in \mathsf{T}_h$ is regular: the ratio between the radius r_{T} of the inscribed sphere and the diameter h_{T} is bounded from below:

$$\frac{r_{\mathsf{T}}}{h_{\mathsf{T}}} \geq \rho_s > 0;$$

AM3 each polyhedron P is star-shaped with respect to a point $\overline{\mathbf{x}}_{\mathsf{P}} \in \mathsf{P}$ and each facet f is star-shaped with respect a point $\overline{\mathbf{x}}_{\mathsf{f}} \in \mathsf{f}$. Moreover the tetrahedral subpartition T_h is simple.

These assumptions requires weak restictions on the shape of admissible elements, the grid Ω_h may contain generally shaped elements, like non-convex elements.

We denote a facet of the polyhedron P by f and by $|\mathsf{P}|$ and $|\mathsf{f}|$ the volume of P and the area of f. For every facet f we consider a unit normal vector $\mathbf{n}_{\mathsf{f}}$, whose orientation is considered to be fixed once and for all. We indicate with $\mathbf{x}_{\mathsf{P}}$ and $\mathbf{x}_{\mathsf{f}}$ the barycenters of the polyhedron P and of the facet f.

Now we introduce the mimetic spaces. We denote by Q_h the discrete space representing the degrees of freedom of the scalar variable. In particular we associate the degrees of freedom of the scalar variable to mesh cells, so that for $q_h \in Q_h$ we have $q_h = \{q_{\mathsf{P}}\}_{\mathsf{P}\in\Omega_h}$, being $q_{\mathsf{P}} \in \mathbb{R}$ the value of the discrete pressure associated to the cell P. Fixed an enumeration of the cells, a discrete field q_h can be represented by an algebraic vector $q_h \in \mathbb{R}^{N_{\mathsf{P}}}$. Let us now introduce the space W_h of the discrete velocities. We associate a velocity degree of freedom $u_{\mathsf{f}} \in \mathbb{R}$ to every facet f of the mesh. It's important to stress that the velocity degrees of freedom of the fracture facets are decoupled, in order to take into account the non conservation of the flux accross the fracture. So given $\mathbf{u}_h \in W_h$ we have $\mathbf{u}_h = \{u_{\mathsf{f}}\}_{\mathsf{f}\in\mathscr{F}_h}$, where $\mathscr{F}_h$ is the set of the facets of the mesh that takes into account the decoupling of fracture facets. Denoting with M_{f} the number of facets of the mesh and with N_Γ the number of facets of the fracture, having fixed an enumeration of the facets of the grid and an enumaration of the facets of the fracture, a discrete vector field $\mathbf{u}_h \in W_h$ can be represented through an algebraic vector $\mathbf{u}_h \in \mathbb{R}^{M_{\mathsf{f}}+N_\Gamma}$, where the additional N_Γ degrees of freedom of the decoupling are added at the tail of the vector. Let us indicate the total number of the facets of the mesh with $N_{\mathsf{f}} = M_{\mathsf{f}} + N_\Gamma$ and the number of elements of the mesh as N_{P}.

The discrete subspace $W_h^{g,\Gamma_N} \subset W_h$ is defined by incorporating the Neumann boundary conditions in W_h:

$$W_h^{g,\Gamma_N} \subset W_h = \left\{ \mathbf{v}_h \in W_h : u_f = \frac{1}{|\mathsf{f}|}\int_{\mathsf{f}} g \; dS \quad \forall\, \mathsf{f} \in \mathscr{F}_h^N \right\},$$

where $\mathscr{F}_h^N \subset \mathscr{F}_h$ is the subset of the facets of the mesh tangent to the Neumann boundary.

We also introduce the jump and the avarage operators across the fracture facet $\mathsf{f} \in \Gamma_h$:

$$[\![\mathbf{u}_h]\!]_{\mathsf{f}} = u_{\mathsf{f}_+} - u_{\mathsf{f}_-}, \qquad \{\mathbf{u}_h\}_{\mathsf{f}} = \frac{u_{\mathsf{f}_+} + u_{\mathsf{f}_-}}{2}$$

where u_{f_+} and u_{f_-} are the degrees of freedom associated to the fracture facet f; f_+ refers to f considered as a facet of the polyhedron $\mathsf{P}_+ \in \Omega_+$ and f_- refers to f considered as a facet of the polyhedron $\mathsf{P}_- \in \Omega_-$.

2.1.2 The mimetic divergence operator

Let us introduce the two projection operators, denoted by the subscript I, from $L^2(\Omega)$ to Q_h and from $H(\mathrm{div}, \Omega)$ to W_h:

$$q_h = q^I = \{q_{\mathsf{P}}\}_{\mathsf{P}\in\Omega_h}, \quad q_{\mathsf{P}} = \frac{1}{|\mathsf{P}|}\int_{\mathsf{P}} q \; dV \tag{2.1}$$

$$\mathbf{u}_h = \mathbf{u}^I = \{u_{\mathsf{f}}\}_{\mathsf{f}\in\mathscr{F}_h}, \quad u_{\mathsf{f}} = \frac{1}{|\mathsf{f}|}\int_{\mathsf{f}} \mathbf{u}\cdot\mathbf{n}_{\mathsf{f}} \; dS. \tag{2.2}$$

Now we define a mimetic discrete divergence operator $\mathrm{div}_h : W_h \to Q_h$. Let us apply the divergence theorem on a polyhedron $\mathsf{P} \in \Omega_h$:

$$\int_{\mathsf{P}} \mathrm{div}\mathbf{u} \; dV = \int_{\partial\mathsf{P}} \mathbf{u}\cdot\mathbf{n}_{\mathsf{P}} \; dS = \sum_{\mathsf{f}\in\partial\mathsf{P}} \int_{\mathsf{f}} \mathbf{u}\cdot\mathbf{n}_{\mathsf{P},\mathsf{f}},$$

where $\mathbf{n}_{\mathsf{P}}$ is the outward unit normal to the element P and $\mathbf{n}_{\mathsf{P},\mathsf{f}}$ is symply the unit normal to the facet $\mathsf{f} \in \partial\mathsf{P}$ outward orientated with respect to P. From this formula, if $\mathbf{u}_h = \mathbf{u}^I$, we define the discrete divergence as

$$(\mathrm{div}_h \mathbf{u}_h)_{\mathsf{P}} = \frac{1}{|\mathsf{P}|}\sum_{\mathsf{f}\in\partial\mathsf{P}} \alpha_{\mathsf{P},\mathsf{f}} |\mathsf{f}| u_{\mathsf{f}}, \tag{2.3}$$

where $\alpha_{\mathsf{P},\mathsf{f}} = \mathbf{n}_{\mathsf{P},\mathsf{f}} \cdot \mathbf{n}_{\mathsf{f}} = \pm 1$. We recall that by definition of div_h it holds:

$$(\mathrm{div}\mathbf{u})^I = \mathrm{div}_h(\mathbf{u}^I).$$

2.1.3 The mimetic inner products

The next goal is to define suitable inner products in Q_h and W_h. On Q_h we simply set

$$[p_h, q_h]_{Q_h} = \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| p_{\mathsf{P}} q_{\mathsf{P}}. \tag{2.4}$$

The construction of the inner product in W_h is more complex. It is defined by assembling elementwise contributions from each cell, i.e.

$$[\mathbf{u}_h, \mathbf{v}_h]_{W_h} = \sum_{\mathsf{P}\in\Omega_h} [\mathbf{u}_{\mathsf{P}}, \mathbf{v}_{\mathsf{P}}]_{\mathsf{P}}, \tag{2.5}$$

where $\mathbf{u}_{\mathsf{P}}, \mathbf{v}_{\mathsf{P}} \in W_{h,\mathsf{P}}$. $W_{h,\mathsf{P}}$ contains the discrete vector fields that collect the degrees of freedom associated to the facets of the element P. From an algebraic point of view the inner product is built in the following way

$$[\mathbf{u}_h, \mathbf{v}_h]_{W_h} = \mathbf{u}_h^{\mathsf{T}} \mathsf{M} \mathbf{v}_h, \qquad [\mathbf{u}_{\mathsf{P}}, \mathbf{v}_{\mathsf{P}}]_{\mathsf{P}} = \mathbf{u}_{\mathsf{P}}^{\mathsf{T}} \mathsf{M}_{\mathsf{P}} \mathbf{v}_{\mathsf{P}}, \tag{2.6}$$

where the $\mathsf{M} \in \mathbb{R}^{N_{\mathsf{f}}\times N_{\mathsf{f}}}$ is the symmetric and positive definite matrix representing the mimetic inner product that approximates the one in L^2 weightened with K^{-1}. The matrix $\mathsf{M}_{\mathsf{P}} \in \mathbb{R}^{N_{\mathsf{P}}^{\mathsf{f}}\times N_{\mathsf{P}}^{\mathsf{f}}}$ represents the local inner product matrix and, as M, it is symmetric and positive definite. With $N_{\mathsf{P}}^{\mathsf{f}}$ we have indicated the number of facets of the cell P.

Let us present some theoretical notions about the construction of a mimetic scalar product; for more details see [3]. We denote as $\mathscr{S}_{h,\mathsf{P}}$ the restriction to the element P of one of the mimetic spaces previously defined. Let us introduce the local projection operator $\Pi_{\mathsf{P}}^{\mathscr{S}} : X|_{\mathsf{P}} \to \mathscr{S}_{h,\mathsf{P}}$ and the space of trial functions $\mathscr{T}_{\mathsf{P}}$, whose definition depends on the accuracy of the mimetic scheme; for a low order mimetic scheme it consists of the constant scalar or vector functions. Let $S_{h,\mathsf{P}}$ be a subspace of $X|_{\mathsf{P}}$ with the following properties.

(1) The projection operator $\Pi_{\mathsf{P}}^{\mathscr{S}}$ is surjective from $S_{h,\mathsf{P}}$ to $\mathscr{S}_{h,\mathsf{P}}$.

(2) The space $S_{h,\mathsf{P}}$ contains the trial space $\mathscr{T}_{\mathsf{P}}$.

(3) Let $q \in \mathscr{T}_{\mathsf{P}}$ and $v \in S_{h,\mathsf{P}}$, then the integral $\int_{\mathsf{P}} qv\, dV$ can be computed exactly using the degrees of freedom, i.e. the components of the vector $\Pi_{\mathsf{P}}^{\mathscr{S}}(v)$.

The assumption (1) states that $S_{h,\mathsf{P}}$ is reach enough, the assumption (2) is related to the accuracy of the mimetic scheme, whereas the condition (3) is problem dependent and allows the concrete construction of the discrete scalar product.

We give two conditions that a mimetic scalar product must satisfy.

Definition 2.1. (**Consistency condition**) The inner product is said to satisfy the consistency condition if

$$[\Pi_{\mathsf{P}}^{\mathscr{S}}(q), \Pi_{\mathsf{P}}^{\mathscr{S}}(v)]_{\mathscr{S}_{h,\mathsf{P}}} = \int_{\mathsf{P}} qv\, dV \qquad \forall\, q \in \mathscr{T}_{\mathsf{P}},\ \forall\, v \in S_{h,\mathsf{P}}. \tag{2.7}$$

The consistency condition ensures that the mimetic scalar product reproduces the L^2 scalar product for a particular choice of the integrands.

Definition 2.2. (**Stability condition**) The inner product is said to satisfy the stability condition if

$$C_*|\mathsf{P}|\ ||v_\mathsf{P}||^2 \le [v_\mathsf{P}, v_\mathsf{P}]_{\mathscr{S}_{h,\mathsf{P}}} \le C^*|\mathsf{P}|\ ||v_\mathsf{P}||^2 \qquad \forall\ v_\mathsf{P} \in \mathscr{S}_{h,\mathsf{P}}, \tag{2.8}$$

with C_* and C^* that are two positive constant independt of P and v_P.

In the latter definition $||.||$ is a discrete norm that is problem dependent. The stability condition ensures the positive definitness of the mimetic inner product and it allows to avoid numerical instabilities. Note that the term $|\mathsf{P}|$ in (2.8) ensures that the induced norm scales as a volume integral.

Let us now turn back to our case in which $\mathscr{S}_{h,\mathsf{P}} = W_{h,\mathsf{P}}$. The goal is to build up a discrete innner product that approximates the one in L^2 weightened with K^{-1}. Let K_P be an a approximation of K over the cell P, e.g. $\mathsf{K}_\mathsf{P} = |\mathsf{P}|^{-1}\int_\mathsf{P} \mathsf{K}\ dV$. We particularize the stability and consistency conditions, previously defined in a general mimetic framework. For the stability condition we state

Definition 2.3. (**S**) There exist two positive constants σ_* and σ^*, independent of the mesh size h, such that

$$\sigma_*|\mathsf{P}|\sum_{\mathsf{f}\in\partial\mathsf{P}} |v_\mathsf{f}|^2 \le [v_\mathsf{P}, v_\mathsf{P}]_\mathsf{P} \le \sigma^*|\mathsf{P}|\sum_{\mathsf{f}\in\partial\mathsf{P}} |v_\mathsf{f}|^2 \qquad \forall\ v_{h,\mathsf{P}} \in W_{h,\mathsf{P}}, \tag{2.9}$$

Before stating the consistency condition we define a concrete space $S_{h,\mathsf{P}}$:

$$S_{h,\mathsf{P}} = \{\mathbf{v} \in (L^s(\mathsf{P}))^3,\ s > 2,\ \mathrm{div}\mathbf{v} = \mathit{const},\ \mathbf{v}\cdot\mathbf{n}_\mathsf{f} = \mathit{const}\ \forall\ \mathsf{f} \in \partial\mathsf{P}\}$$

and the trial space $\mathscr{T}_\mathsf{P}$ as

$$\mathscr{T}_\mathsf{P} = \{\mathbf{v} : \mathsf{P} \to \mathbb{R}^3 : \mathbf{v} = \mathsf{K}_\mathsf{P}\nabla q,\ q \in \mathbb{P}_1(\mathsf{P})\}.$$

It is trivial to verify that $S_{h,\mathsf{P}}$ ensures conditions (1) and (2).

Now we state the consistency condition.

Definition 2.4. (**C**) For any vector function $\mathbf{v} \in S_{h,\mathsf{P}}$, any linear polynomial q and every element $\mathsf{P} \in \Omega_h$ it holds:

$$[(\mathsf{K}_\mathsf{P}\nabla q)^I_\mathsf{P}, \mathbf{v}^I_\mathsf{P}]_\mathsf{P} = \int_\mathsf{P} \mathsf{K}_\mathsf{P}^{-1}(\mathsf{K}_\mathsf{P}\nabla q)\cdot\mathbf{v}\ dV. \tag{2.10}$$

Observe that $I : H(\mathrm{div}, \mathsf{P}) \to W_{h,\mathsf{P}}$ is the local version of the projection operator defined in (2.2) and note the tensorial weight $\mathsf{K}_\mathsf{P}^{-1}$.

Now we show that the right-hand side of (**C**) is computable, i.e. that the condition (3) on the space $S_{h,\mathsf{P}}$ holds. Integrating by parts and using the properties of $S_{h,\mathsf{P}}$ we get

$$\begin{aligned}\int_\mathsf{P} \mathsf{K}_\mathsf{P}^{-1}(\mathsf{K}_\mathsf{P}\nabla q)\cdot\mathbf{v}\ dV = \int_\mathsf{P} \nabla q\cdot\mathbf{v}\ dV &= -\int_\mathsf{P} q\mathrm{div}\mathbf{v}\ dV + \sum_{\mathsf{f}\in\partial\mathsf{P}}\int_\mathsf{f} q(\mathbf{v}\cdot\mathbf{n}_{\mathsf{P},\mathsf{f}})\ dS \\ &= -\mathrm{div}_\mathsf{P}\mathbf{v}^I_\mathsf{P}\int_\mathsf{P} q\ dV + \sum_{\mathsf{f}\in\partial\mathsf{P}} \alpha_{\mathsf{P},\mathsf{f}} v^I_\mathsf{f}\int_\mathsf{f} q\ dS,\end{aligned} \tag{2.11}$$

since $\mathrm{div}_\mathsf{P}\mathbf{v}^I_\mathsf{P} = (\mathrm{div}\mathbf{v})|_\mathsf{P} = \mathit{const}$ and $v^I_\mathsf{f} = \mathbf{v}\cdot\mathbf{n}_\mathsf{f} = \mathit{const}$, where $\mathrm{div}_\mathsf{P} : W_{h,\mathsf{P}} \to Q_{h,\mathsf{P}}$ is the local version of the divergence operator defined in (2.3). Thus the avarage normal components of $\mathbf{v}$ on the facets $\mathsf{f} \in \partial \mathsf{P}$ are all what is needed to compute the integral and so the condition (3) about $S_{h,\mathsf{P}}$ holds.

The condition ($\mathbf{C}$) holds for every linear polinomial q. Now let us restrict to the linear polynomials in P of zero mean, i.e. we require ($\mathbf{C}$) not for $\mathbb{P}_1(\mathsf{P})$ but for $\widetilde{\mathbb{P}}_1(\mathsf{P}) = \{q \in \mathbb{P}_1(\mathsf{P}) : \int_\mathsf{P} q\ dV = 0\}$. In that way the volume integral in the equation (2.11) is zero and the consistency condition ($\mathbf{C}$) becomes

$$[(\mathsf{K}_\mathsf{P}\nabla q)^I_\mathsf{P}, \mathbf{v}^I_\mathsf{P}]_\mathsf{P} = \sum_{\mathsf{f}\in\partial\mathsf{P}} \alpha_{\mathsf{P},\mathsf{f}} v^I_\mathsf{f} \int_\mathsf{f} q\ dS \quad \forall\ \mathbf{v} \in S_{h,\mathsf{P}},\ \forall\ q \in \widetilde{\mathbb{P}}_1(\mathsf{P}). \tag{2.12}$$

Discarding constant functions is not restrictive because taking $q = 1$ in (2.11) the definition of the discrete divergence can be retrieved.

Now our goal is to reformulate ($\mathbf{C}$) in an equivalent way using a simple basis of $\widetilde{\mathbb{P}}_1(\mathsf{P})$ given by

$$q_1(x,y,z) = x - x_\mathsf{P},\ q_2(x,y,z) = y - y_\mathsf{P},\ q_3(x,y,z) = z - z_\mathsf{P},$$

where $\mathbf{x}_\mathsf{P} = (x_\mathsf{P}, y_\mathsf{P}, z_\mathsf{P})$ is the barycenter of the cell P. Let us define, for $j = 1, 2, 3$, a vector $\mathsf{N}_j \in \mathbb{R}^{N^\mathsf{f}_\mathsf{P}}$. The definition is the following:

$$\mathsf{N}_j = (\mathsf{K}_\mathsf{P}\nabla q_j)^I_\mathsf{P}, \quad (\mathsf{N}_j)_i = \frac{1}{|\mathsf{f}_i|}\int_{\mathsf{f}_i} \mathbf{n}_{\mathsf{f}_i} \cdot \mathsf{K}_\mathsf{P}\nabla q_j\ dS = \mathbf{n}^\mathsf{T}_{\mathsf{f}_i}\mathsf{K}_\mathsf{P} \cdot \nabla q_j$$

Let us define the matrix $\mathsf{N}_\mathsf{P} \in \mathbb{R}^{N^\mathsf{f}_\mathsf{P}\times 3}$ as $\mathsf{N}_\mathsf{P} = [\mathsf{N}_1, \mathsf{N}_2, \mathsf{N}_3]$. Since $\nabla q_j = \mathbf{e}_j$, where $\mathbf{e}_j$ is the j-th vector of the canonical base of $\mathbb{R}^3$, the N_P formula is

$$\mathsf{N}_\mathsf{P} = \begin{bmatrix} \mathbf{n}^\mathsf{T}_{\mathsf{f}_1} \\ \vdots \\ \mathbf{n}^\mathsf{T}_{\mathsf{f}_i} \\ \vdots \\ \mathbf{n}^\mathsf{T}_{\mathsf{f}_{N^\mathsf{f}_\mathsf{P}}} \end{bmatrix} \mathsf{K}_\mathsf{P}. \tag{2.13}$$

The ($\mathbf{C}$) can be reformulated as

$$[(\mathsf{K}_\mathsf{P}\nabla q_j)^I_\mathsf{P}, \mathbf{v}^I_\mathsf{P}]_\mathsf{P} = (\mathbf{v}^I_\mathsf{P})^\mathsf{T}\mathsf{M}_\mathsf{P}\mathsf{N}_j = (\mathbf{v}^I_\mathsf{P})^\mathsf{T}\mathsf{R}_j, \tag{2.14}$$

where the vector $\mathsf{R}_j \in \mathbb{R}^{N^\mathsf{f}_\mathsf{P}}$ is defined as

$$(\mathsf{R}_j)_i = \alpha_{\mathsf{P},\mathsf{f}_i}\int_{\mathsf{f}_i} q_j\ dV. \tag{2.15}$$

Let us define the matrix $\mathsf{R}_\mathsf{P} \in \mathbb{R}^{N^\mathsf{f}_\mathsf{P}\times 3}$ as $\mathsf{R}_\mathsf{P} = [\mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3]$. Because $\mathbf{v}^I_\mathsf{P}$ is arbitrary, from (2.14) we get

$$\mathsf{M}_\mathsf{P}\mathsf{N}_j = \mathsf{R}_j, \quad j = 1, 2, 3.$$

The latter condition can be written in a compact form as

$$\mathsf{M}_\mathsf{P}\mathsf{N}_\mathsf{P} = \mathsf{R}_\mathsf{P}, \tag{2.16}$$

which is the algebraic consistency condition.

Finally we provide the explicit formula for matrix R_P.The face integral (see (2.15)) of a linear function equals its value at the barycenter $\mathbf{x}_\mathsf{f}$ times the facet area $|\mathsf{f}|$. Thus,

$$\mathsf{R}_\mathsf{P} = \begin{bmatrix} \alpha_{\mathsf{P},\mathsf{f}_1}|\mathsf{f}_1|(\mathbf{x}_{\mathsf{f}_1} - \mathbf{x}_\mathsf{P}) \\ \vdots \\ \alpha_{\mathsf{P},\mathsf{f}_i}|\mathsf{f}_i|(\mathbf{x}_{\mathsf{f}_i} - \mathbf{x}_\mathsf{P}) \\ \vdots \\ \alpha_{\mathsf{P},\mathsf{f}_{N^\mathsf{f}_\mathsf{P}}}|\mathsf{f}_{N^\mathsf{f}_\mathsf{P}}|(\mathbf{x}_{\mathsf{f}_{N^\mathsf{f}_\mathsf{P}}} - \mathbf{x}_\mathsf{P}) \end{bmatrix} \tag{2.17}$$

A possible solution of (2.16), that can be simply verified by substitution, is given by

$$\mathsf{M_P} = \mathsf{R_P}(\mathsf{R_P^T N_P})^{-1}\mathsf{R_P^T} \tag{2.18}$$

Thanks to the following lemma $(\mathsf{R_P^T N_P})^{-1}$ must be not explicitly computed.

Lemma 2.5. *For any polyhedral cell* P *we have*

$$\mathsf{N_P^T R_P} = \mathsf{K_P}|\mathsf{P}|. \tag{2.19}$$

For the proof of this result we refer to [3]. Using the lemma (2.5) the solution (2.18) reduces to

$$\mathsf{M_P} = \mathsf{R_P}\Big(\frac{1}{|\mathsf{P}|}\mathsf{K_P^{-1}}\Big)\mathsf{R_P^T}. \tag{2.20}$$

However this formula provides a matrix that may not satisfy the stability condition (**S**). The problem is rectified by adding in (2.20) a symmetric and semi-positive definite matrix $\mathsf{M_P^1}$ such that $Ker(\mathsf{M_P^1}) = Range(\mathsf{N_P})$. The important point to make $\mathsf{M_P}$ stable is to ensure that $\mathsf{M_P^1}$ has the same scaling of the consistency term. An effective choice is given by the scaled orthogonal projector. So that finally the $\mathsf{M_P}$, that satisfy both (**C**) and (**S**), is computed as follows

$$\mathsf{M_P} = \mathsf{M_P^0} + \mathsf{M_P^1}, \tag{2.21}$$

where the consistency term $\mathsf{M_P^0}$ is given by

$$\mathsf{M_P^0} = \mathsf{R_P}\Big(\frac{1}{|\mathsf{P}|}\mathsf{K_P^{-1}}\Big)\mathsf{R_P^T} \tag{2.22}$$

and the stability term $\mathsf{M_P^1}$ is given by

$$\mathsf{M_P^1} = \gamma_\mathsf{P}(\mathsf{I} - \mathsf{N_P}(\mathsf{N_P^T N_P})^{-1}\mathsf{N_P^T}), \quad \gamma_\mathsf{P} = \frac{2}{N_\mathsf{P}^\mathsf{f}|\mathsf{P}|}tr(\mathsf{R_P K_P^{-1} R_P^T}). \tag{2.23}$$

2.1.4 Discretization of the bullk problem

We are now ready to state the mimetic discretization of the bulk problem, i.e. of (1.7), under the assumption that p_Γ is given. Let us consider an approximation of p_Γ given by

$$p_{\Gamma,\mathsf{f}} = \frac{1}{|\mathsf{P}|}\int_\mathsf{f} p_\Gamma \, dS \qquad \forall f \in \Gamma_h.$$

Actually the approximation of the pressure in the fracture will be computed through finite volume as explained in the next section. The numerical scheme on bulk reads as: find $(\mathbf{u}_h, p_h) \in W_h^{g,\Gamma_N} \times Q_h$ such that

$$\begin{cases} [\mathbf{u}_h, \mathbf{v}_h]_{W_h} + [\mathbf{u}_h, \mathbf{v}_h]_\Gamma - [p_h, \mathrm{div}_h \mathbf{v}_h]_{Q_h} + \sum\limits_{\mathsf{f}\in\Gamma_h} |\mathsf{f}| p_{\Gamma,\mathsf{f}} [\![\mathbf{v}_h]\!]_\mathsf{f} = L_h(\mathbf{v}_h) & \forall\ \mathbf{v}_h \in W_h^{0,\Gamma_N} \\ [\mathrm{div}_h \mathbf{u}_h, q_h]_{Q_h} = G_h(q_h) & \forall\ q_h \in Q_h, \end{cases} \tag{2.24}$$

where

$$[\mathbf{u}_h, \mathbf{v}_h]_\Gamma = \sum_{\mathsf{f}\in\Gamma_h} \eta_\Gamma |\mathsf{f}| \{\mathbf{u}_h\}_\mathsf{f} \{\mathbf{v}_h\}_\mathsf{f} + \sum_{\mathsf{f}\in\Gamma_h} \xi_0 \eta_\Gamma |\mathsf{f}| [\![\mathbf{u}_h]\!]_\mathsf{f} [\![\mathbf{v}_h]\!]_\mathsf{f} \tag{2.25}$$

$$L_h(\mathbf{v}_h) = -\sum_{\mathsf{f}\in\mathscr{F}_h^{\Gamma_D}} \alpha_{\mathsf{P},\mathsf{f}} |\mathsf{f}| g_\mathsf{f} v_\mathsf{f}, \qquad g_\mathsf{f} = \frac{1}{|\mathsf{f}|}\int_\mathsf{f} g \, dS \tag{2.26}$$

$$G_h(\mathbf{v}_h) = [f^I, q_h]_{Q_h}. \tag{2.27}$$

With the notation $\mathscr{F}_h^{\Gamma_D}$ we have indicated the set of facets tangent to the Dirichlet boundary. We have assumed a constant $\eta_\Gamma = l_\Gamma (k_\Gamma^n)^{-1}$, i.e. that the fracture has a constant aperture l_Γ and a constant normal permeability k_Γ^n.

2.2 Finite volume approximation

In this section we present the finite volume discretization of the fracture problem (1.5), based on the two-point flux approximation. For more details on FV and on the two-point flux scheme see [7]. Let us assume that the single fracture is actually a plane. We consider in the fracture a local coordinates system given by the orthonormal columns of the matrix $\boldsymbol{\tau}_\Gamma$. With this coordinate system the fracture equation is

$$-\nabla \cdot (k_\Gamma \nabla p_\Gamma) = l_\Gamma f_\Gamma + [\![\mathbf{u} \cdot \mathbf{n}_\Gamma]\!] \qquad \textit{in } \Gamma,$$

where, clearly, the differential operators collect derivatives with respect the local coordinates.

In our finite volume scheme the fracture facets are the so called control volumes or cells. Having fixed an enumeration of the fracture facets, let us integrate over a fracture facet $\mathsf{f}_i \in \Gamma_h$ the latter equation:

$$-\int_{\mathsf{f}_i} \nabla \cdot (k_\Gamma \nabla p_\Gamma)\; dS = \int_{\mathsf{f}_i} l_\Gamma f_\Gamma \; dS + \int_{\mathsf{f}_i} [\![\mathbf{u} \cdot \mathbf{n}_\Gamma]\!]\; dS \qquad i = 1, ..., N_\Gamma.$$

The terms involving the jump of the velocity accros the fracture and the source are discretized as

$$\int_{\mathsf{f}_i} [\![\mathbf{u} \cdot \mathbf{n}_\Gamma]\!]\; dS \simeq |\mathsf{f}_i| [\![\mathbf{u}_h]\!]_{\mathsf{f}_i}, \qquad \int_{\mathsf{f}_i} l_\Gamma f_\Gamma \; dS \simeq |\mathsf{f}_i| l_\Gamma f_{\Gamma,i}$$

where $\mathbf{u}_h \in W_h^{g,\Gamma_N}$ is the discrete mimetic velocity and $f_{\Gamma,i}$ is a mean value over the fracture facet f_i of the forcing term. So that the equation becomes

$$-\int_{\mathsf{f}_i} \nabla \cdot (k_\Gamma \nabla p_\Gamma)\; dS - |\mathsf{f}_i| [\![\mathbf{u}_h]\!]_{\mathsf{f}_i} = |\mathsf{f}_i| l_\Gamma |f_{\Gamma,i}| \qquad i = 1, ..., N_\Gamma$$

Applying the divergence theorem and splitting the integral over the the boundary of f_i in integrals over the edges e_{ij} of f_i, we get

$$-\sum_{j=1}^{N_{\mathsf{e}}^i} \int_{\mathsf{e}_{ij}} k_\Gamma \nabla p_\Gamma \cdot \mathbf{n}_{ij}\; dS - |\mathsf{f}_i| [\![\mathbf{u}_h]\!]_{\mathsf{f}_i} = |\mathsf{f}_i| l_\Gamma |f_{\Gamma,i}| \qquad i = 1, ..., N_\Gamma, \tag{2.28}$$

where $\mathbf{n}_{ij}$ is the normal to the edge e_{ij}, outward orientated with respect f_i and N_{e}^i is the number of edges of the facet f_i.

We consider a cell-centered scheme and so we locate the degrees of freedom of the fracture pressure in the barycenters of the fracture facets. Every edge integral in the equation (2.28) represents an outward flux from the fracture cell f_i through the edge e_{ij}. The fluxes have the following form

$$q_{\mathsf{e}} = -\int_{\mathsf{e}} k_\Gamma \nabla p_\Gamma \cdot \mathbf{n}\; dS$$

and the equation (2.28) is the balance of fluxes in the fracture cell f_i, where the summation collects the contributions between f_i and the adjacent fracture cells and the velocity jump the contributions between f_i and the adjacent bulk cells. The way through which we approximate the flux q_{e} defines the FV scheme. We use a two-point flux approximation, i.e. we want to approximate q_{e} as a pressure difference in the barycenters of two adjcent fracture cells.

Let us consider two cells that share an edge and let us introduce some geometrical quantities like the barycenters C_1 and C_2 of the cells, the midpoint C_0 of the edge, the

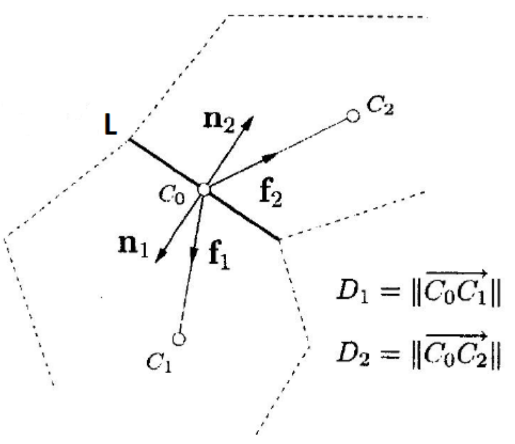
L
n_2
C_2
C_0
f_2
n_1
f_1
C_1
$D_1 = \|\overrightarrow{C_0C_1}\|$
$D_2 = \|\overrightarrow{C_0C_2}\|$

Figure 2.1: Geometrical quantities involved in the TPFA of FV.

edge length $L_1 = L_2 = L$, $\mathbf{n}_1$ and $\mathbf{n}_2$ indicating the unit normal vectors to the edge, D_1 and D_2 denoting the length of the segments C_0C_1 and C_0C_2 and the unit vectors $\mathbf{f}_1$ from C_0 to $C1$ and $\mathbf{f}_2$ from C_0 to C_2. To have a graphical view of the situation see Figure 2.1. We assume the tensor k_Γ constant in each fracture cell and denote with $k_{\Gamma,1}$ and $k_{\Gamma,2}$ its values on the two cells. Because of the simmetry of the permeability tensor it holds $k_\Gamma \nabla p_\Gamma \cdot \mathbf{n} = \nabla p_\Gamma \cdot k_\Gamma \mathbf{n}$, so that $k_\Gamma \nabla p_\Gamma \cdot \mathbf{n}$ is simply the gradient of p_Γ along the $k_\Gamma \mathbf{n}$ direction. Let us assume that the grid Γ_h has the K-orthogonality property, i.e. $\mathbf{f}_1$ and $\mathbf{f}_2$ are parallel to $k_\Gamma \mathbf{n}$ and this holds for every control cells f_i (see [7] for more details about K-orthogonal grids).

With this assumption it makes sense to approximate the flux q_{12} between the two cells with a pressure difference in the following ways

$$q_{12} \simeq L_1 \left[\frac{p_{\Gamma,1} - p_{\Gamma,0}}{D_1} \mathbf{f}_1 \cdot k_{\Gamma,1} \mathbf{n}_1 \right]$$
$$q_{12} \simeq L_2 \left[\frac{p_{\Gamma,0} - p_{\Gamma,2}}{D_2} \mathbf{f}_2 \cdot k_{\Gamma,2} \mathbf{n}_2 \right],$$

where $p_{\Gamma,0}$ denote the fracture pressure in C_0 and $p_{\Gamma,1}$ and $p_{\Gamma,2}$ are the degrees of freedom associated to the fracture cells. Erasing the flux and making some calculations we obtain an equation in terms of the only available degrees of freedom $p_{\Gamma,1}$ and $p_{\Gamma,2}$:

$$p_{\Gamma,1} - p_{\Gamma,2} = \frac{q_{12}}{L} \left(\frac{D_1}{\mathbf{f}_1 \cdot k_{\Gamma,1} \mathbf{n}_1} + \frac{D_2}{\mathbf{f}_2 \cdot k_{\Gamma,2} \mathbf{n}_2} \right).$$

We rewrite the latter equation as

$$q_{12} = T_{12}(p_{\Gamma,1} - p_{\Gamma,2}),$$

where $T_{1,2}$ is the transmissibility between the cells 1 and 2. More in general we can write

$$q_{ij} = T_{ij}(p_{\Gamma,i} - p_{\Gamma,j}),$$

where

$$T_{ij} = L \left(\frac{D_i}{\mathbf{f}_i \cdot k_{\Gamma,i} \mathbf{n}_i} + \frac{D_j}{\mathbf{f}_j \cdot k_{\Gamma,j} \mathbf{n}_j} \right)^{-1} \tag{2.29}$$

is the transmissibility accross the edge e_{ij}. Tipically we define the coefficient

$$\alpha_i = \frac{L_i k_{\Gamma,i}}{D_i} \mathbf{f}_i \cdot \mathbf{n}_i \tag{2.30}$$

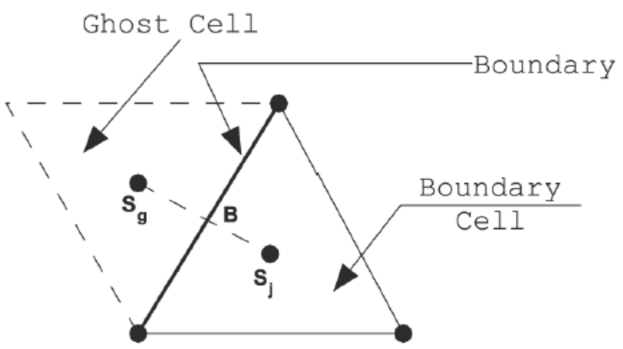

Figure 2.2: Sketch of the boundary for the ghost cell approach.

and we express the transmissibility as

$$T_{ij} = \frac{\alpha_i \alpha_j}{\alpha_i + \alpha_j}. \tag{2.31}$$

Now let us focus on the treatment of boundary conditions on fracture. For the Dirichlet boundary conditions we employ a "ghost cell" approach; see Figure 2.2 to have an idea of this technique. In particular we create a ghost cell beside the boundary cell in such a way that the degree of freedom associated with the ghost cell, s_g, is placed at the same distance of s_j from the boundary and such that the line that connects s_j to the centroid of the boundary facet is orthogonal to the boundary. The numerical flux it will be $q_{jg} = T_{ij}(p_{\Gamma,j} - p_{\Gamma,g})$, where $p_{\Gamma,g}$ is determined through the boundary condition. In the case of Neumann boundary conditions we simply prescribe the flux that can be computed as $h_j L_j$, where h_j is determined through the boundary condition and L_j is the length of the boundary edge.

Now we focus on the case of fracture intersections. Up to now we have supposed only one fracture to ease the presentation of the theoretical part, but the code handle network of fractures and so we briefly describe the treatment of fracture intersections. In this case the transmissibility definition given by (2.31) has to be generalized. The employed technique is the so called "star-delta" approach. The idea is to proceed by analogy between the flow of a fluid through porous media and the flow of electric charges through a network of resistors and use the “star-delta” transformation. Hence the transmissibility between a pair of fractures i and j in an intersection of n fractures is computed as follow:

$$T_{ij} = \frac{\alpha_i \alpha_j}{\sum_{k=1}^{n} \alpha_k} \tag{2.32}$$

For more details on the "star-delta" approach see [11].

Remark 2.1. *We have considered a local coordinate system for the fracture, in order to show that the evaluation of the tangential differential operators in the fracture model (1.5) is not needed. Actually from an implementative point of view also the knowledge of the local coordinate system is not needed, because the TPFA scheme of FV involves only geometrical quantities that do not depend on the coordinates system. This latter thing is not trivial because the knowledge of a local coordinates system may be complex in a network of fractures.*

2.3 Algebraic formulation

In this section we describe the linear system arising from the hybrid MFD-FV numerical scheme. From the mimetic formulation of the bulk problem, i.e. from the equations

(2.24), the following algebraic problem arises:

$$\begin{cases} \mathsf{M}_c\mathbf{u} + \mathsf{B}^\mathsf{T}\mathbf{p} + \mathsf{C}^\mathsf{T}\mathbf{p}_\Gamma = \mathbf{g} \\ \mathsf{B}\mathbf{u} = \mathbf{h}, \end{cases} \tag{2.33}$$

where the $\mathsf{M}_c \in \mathbb{R}^{N_\mathsf{f}\times N_\mathsf{f}}$ represents the weightened mimetic inner product and the coupling conditions terms $[\mathbf{u}_h, \mathbf{v}_h]_{W_h} + [\mathbf{u}_h, \mathbf{v}_h]_\Gamma$, $\mathsf{B} \in \mathbb{R}^{N_\mathsf{P}\times N_\mathsf{f}}$ represents the weak divergence operator $-[p_h, \mathrm{div}_h\mathbf{v}]_{Q_h}$, $\mathsf{C} \in \mathbb{R}^{N_\Gamma\times N_\mathsf{f}}$ represents the other term of the coupling conditions $\sum_{\mathsf{f}\in\Gamma_h} |\mathsf{f}|p_{\Gamma,\mathsf{f}}[\![\mathbf{v}_h]\!]_\mathsf{f}$, $\mathbf{g} \in \mathbb{R}^{N_\mathsf{f}}$ takes into account the Dirichlet and Neumann boundary conditions on bulk and $\mathbf{h} \in \mathbb{R}^{N_\mathsf{P}}$ represents the forcing term in the bulk. The vectors $\mathbf{u} \in \mathbb{R}^{N_\mathsf{f}}$ and $\mathbf{p} \in \mathbb{R}^{N_\mathsf{P}}$ are the algebraic counterparts of the discrete velocity $\mathbf{u}_h$ and of the discrete bulk pressure p_h.

From the discretization of the fracture problem with the TPFA scheme of FV, i.e. from the equation (2.28) changed in sign, the following algebraic equation arises:

$$\mathsf{C}\mathbf{u} - \mathsf{T}\mathbf{p}_\Gamma = \mathbf{h}_\Gamma, \tag{2.34}$$

where $\mathsf{T} \in \mathbb{R}^{N_\Gamma\times N_\Gamma}$ is the transmissibility matrix of FV and $\mathbf{h}_\Gamma \in \mathbb{R}^{N_\Gamma}$ takes into account the forcing term and the Dirichlet and Neumann boundary conditions on fractures. The vector $\mathbf{p}_\Gamma \in \mathbb{R}^{N_\Gamma}$ collects the fracture cell degrees of freedom for the pressure.

The complete algebraic system is:

$$\begin{bmatrix} \mathsf{M}_c & \mathsf{B}^\mathsf{T} & \mathsf{C}^\mathsf{T} \\ \mathsf{B} & 0 & 0 \\ \mathsf{C} & 0 & -\mathsf{T} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \\ \mathbf{p}_\Gamma \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ \mathbf{h} \\ \mathbf{h}_\Gamma \end{bmatrix}. \tag{2.35}$$

Imposing the Neumann boundary conditions by modifying the rows of the system corresponding to Neumann faces implies to make the matrix of the system non symmetric, but a symmetric matrix turns out in good properties in terms of iterative solvers and preconditioning. Moreover from an implementative point of view performing a zero reset of Neumann rows can be done efficiently only if the sparse matrix of the system is stored by row, but in that way some efficient solution techniques like the UmfPack direct solver are unavailable. So it has been decided to impose the boundary conditions through a Nitsche technique that preserves the symmetry of the system and that can be implemented well with a column storage of the sparse matrix of the system. Roughly speaking it consits of a modification of the penalty approach (that would convert the Neumann boundary condition in a Robin one with a penalty term) to recover the cosistency of the numerical scheme. For a detailed description of the Nitsche method we refer to [5]. The Nitsche technique affects the terms M_c, B, B^T, $\mathbf{g}$ and $\mathbf{h}$, because it adds some boundary terms in the mimetic formulation (2.24); we do not enter the details and for its application in the case of fractured porous media we refer to [6].

The M_c matrix is the inner product matrix M, assembled through the local matrices M_P given by the set of formulas (2.21)-(2.23), modified with the contributions of the coupling conditions. We do not enter the details of each matrix and right-hand side vector of the linear system; for a detailed description of the bulk algebraic problem (2.33) we refer to [10]. Here we focus only on the matrix B, showing that it can be built, as for the inner product matrix, from local weak divergence matrices B_P. Later on we will do some interesting comments from an implementative point of view on how to build the cell-related matrices M_P and B_P and to assemble them in M and B in an efficient way. The B matrix arises from

$$-[\mathrm{div}_h\mathbf{u}_h, p_h]_{Q_h} = -\sum_{\mathsf{P}\in\Omega_h} p_\mathsf{P} \sum_{\mathsf{f}\in\mathscr{F}_\mathsf{P}} \alpha_{\mathsf{P},\mathsf{f}}|\mathsf{f}|u_\mathsf{P} = \mathbf{p}^\mathsf{T}\mathsf{B}\mathbf{u}$$

and the local divergence contribution $\mathsf{B}_\mathsf{P} \in \mathbb{R}^{1\times N_\mathsf{P}^{\mathscr{F}}}$ is actually a row vector of the form:

$$\mathsf{B}_\mathsf{P} = -\begin{bmatrix}\alpha_{\mathsf{P},\mathsf{f}_1} & \cdots & \alpha_{\mathsf{P},\mathsf{f}_i} \cdots & \alpha_{\mathsf{P},\mathsf{f}_{N_\mathsf{P}^{\mathscr{F}}}}\end{bmatrix}. \tag{2.36}$$

2.4 Saddle point reduction

The system (2.35) can be rewritten in a saddle point form. This is very important because saddle point problems are rather common in scientific computing and a lot of theory has been developed on this topic (see e.g. [4]).

Let us define the following block matrices $\widetilde{\mathsf{B}} \in \mathbb{R}^{(N_\mathsf{P}+N_\Gamma)\times N_\mathsf{f}}$ and $\widetilde{\mathsf{T}} \in \mathbb{R}^{(N_\mathsf{P}+N_\Gamma)\times(N_\mathsf{P}+N_\Gamma)}$ as

$$\widetilde{\mathsf{B}} = \begin{bmatrix}\mathsf{B}\\ \mathsf{C}\end{bmatrix}, \quad \widetilde{\mathsf{T}} = \begin{bmatrix}0 & 0\\ 0 & \mathsf{T}\end{bmatrix}. \tag{2.37}$$

Defining the vectors $\widetilde{\mathbf{p}} = [\mathbf{p}, \mathbf{p}_\Gamma]^\mathsf{T}$ and $\widetilde{\mathbf{h}} = [\mathbf{h}, \mathbf{h}_\Gamma]^\mathsf{T}$, the problem can be rewritten as

$$\begin{bmatrix}\mathsf{M}_c & \widetilde{\mathsf{B}}^\mathsf{T}\\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}}\end{bmatrix}\begin{bmatrix}\mathbf{u}\\ \widetilde{\mathbf{p}}\end{bmatrix} = \begin{bmatrix}\mathbf{g}\\ \widetilde{\mathbf{h}}\end{bmatrix}, \tag{2.38}$$

that is the classical generalized (because $\widetilde{\mathsf{T}} \neq 0$) saddle point algebraic system. What we have done is simply to collect the pressure degrees of freedom of bulk and fractures cells.

For a saddle point matrix it holds the following decomposition:

$$\mathsf{A} = \begin{bmatrix}\mathsf{M}_c & \widetilde{\mathsf{B}}^\mathsf{T}\\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}}\end{bmatrix} = \begin{bmatrix}\mathsf{I} & 0\\ \widetilde{\mathsf{B}}\mathsf{M}_c^{-1} & \mathsf{I}\end{bmatrix}\begin{bmatrix}\mathsf{M}_c & 0\\ 0 & \mathsf{S}\end{bmatrix}\begin{bmatrix}\mathsf{I} & \mathsf{M}_c^{-1}\widetilde{\mathsf{B}}^\mathsf{T}\\ 0 & \mathsf{I}\end{bmatrix}, \tag{2.39}$$

where $\mathsf{S} = -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\mathsf{M}_c^{-1}\widetilde{\mathsf{B}}^\mathsf{T})$ is the Schur complement. From (2.39) A is non singular if and only if M_c and S are non singular. We know that M_c is symmetric and positive definite, so the invertibility of A depends on the Schur complement. Let us note that in our case $\widetilde{\mathsf{T}}$ is symmetric semi-positive definite because T is symmetric semi-positive definite, so it holds

$$\mathbf{x}^\mathsf{T}\mathsf{S}\mathbf{x} = -\mathbf{x}^\mathsf{T}\widetilde{\mathsf{T}}\mathbf{x} - (\widetilde{\mathsf{B}}^\mathsf{T}\mathbf{x})^\mathsf{T}\mathsf{M}_c^{-1}(\widetilde{\mathsf{B}}\mathbf{x}) < 0 \quad \forall \mathbf{x} \in \mathbb{R}^{(N_\mathsf{P}+N_\Gamma)\times(N_\mathsf{P}+N_\Gamma)} \setminus \{\mathbf{0}\}$$

if and only if

$$Ker(\widetilde{\mathsf{T}}) \cap Ker(\widetilde{\mathsf{B}}^\mathsf{T}) = \{\mathbf{0}\}. \tag{2.40}$$

In [2] for the single fracture case it has been already shown that $Ker(\widetilde{\mathsf{B}}^\mathsf{T}) = \mathbf{0}$ and so that S is symmetric and definite negative, therefore A is non singular and the scheme admits a unique solution.

The system, as always for saddle point, is symmetric but indefinite. The non-definitness of the matrix makes not possible the usage of some efficient solvers like the Cholesky factorization or the Conjugate Gradient; in any case there are several other methods, e.g. a standard LU factorization or the UmfPack as direct solvers or the GMRES and BiCGSTAB as iterative methods.

2.5 Preconditioning

The saddle point nature of the problem leads the way to many possible choices in terms of preconditioning (see [4]). At the same time it's not always easy to accomodate the $\widetilde{\mathsf{T}} \neq 0$ term and there are not standard choices as in a classical Stokes problem. In the code two preconditioners based on the diagonal part of M_c have been succesfully implemented: a block triangular preconditioner and a preconditioner based on an inexact LU factorization.

Given $\widehat{\mathsf{M}}_c$ and $\widehat{\mathsf{S}}$, two approximations of M_c and S, a typical block triangular preconditioner is of the form

$$\mathsf{P} = \begin{bmatrix} \widehat{\mathsf{M}}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ 0 & \widehat{\mathsf{S}} \end{bmatrix}.$$

This choice is based on the fact that if we had the exact M_c and S in the preconditioner formula then we would have a preconditioned system matrix such that $\sigma(\mathsf{P}^{-1}\mathsf{A}) = 1$ (see [4]); clearly storing and inverting such a preconditioner would be too expensive and so in practice we replace M_c and S with easily invertible approximations. In our case good results have been obtained in the case $\widehat{\mathsf{M}}_c = \mathsf{D}_{\mathsf{M}_c}$ and $\widehat{\mathsf{S}} = -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\mathsf{D}_{\mathsf{M}_c}^{-1}\widetilde{\mathsf{B}}^{\mathsf{T}})$ where $\mathsf{D}_{\mathsf{M}_c}$ is the diagonal part of M_c, so the employed preconditioner is:

$$\mathsf{P} = \begin{bmatrix} \mathsf{D}_{\mathsf{M}_c} & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ 0 & -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\mathsf{D}_{\mathsf{M}_c}^{-1}\widetilde{\mathsf{B}}^{\mathsf{T}}) \end{bmatrix}. \tag{2.41}$$

Solving a linear system in an iterative way requires basically two operations:

(1) matrix-vector product $\mathbf{y} = \mathsf{A} * \mathbf{x}$,

(2) solving the linear system $\mathsf{P}\mathbf{z}^k = \mathbf{r}^k$,

where $\mathbf{r}^k$ is the residual vector of the system at the k-th iteration and $\mathbf{z}^k$ is the residual of the preconditioned system at the k-th iteration. Because the system matrix A is a block matrix the product with a vector can be performed without storing the whole matrix. In this way in the case of an iterative solver we can avoid to store the whole matrix plus the single blocks to compute the inexact Schur complement $\widehat{\mathsf{S}}$, but we can simply store the single blocks M_c, $\widetilde{\mathsf{B}}$ and $\widetilde{\mathsf{T}}$ and through them we can assemble $\widehat{\mathsf{S}}$ and we can do all we need to solve the linear system. So the operation (1) is performed as

$$\begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathsf{M}_c\mathbf{x} + \widetilde{\mathsf{B}}^{\mathsf{T}}\mathbf{y} \\ \widetilde{\mathsf{B}}\mathbf{x} - \widetilde{\mathsf{T}}\mathbf{y} \end{bmatrix}. \tag{2.42}$$

The operation (2) can be performed through the following factorization of P^{-1}:

$$\mathsf{P}^{-1} = \begin{bmatrix} \mathsf{D}_{\mathsf{M}_c}^{-1} & 0 \\ 0 & \mathsf{I} \end{bmatrix} \begin{bmatrix} \mathsf{I} & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ 0 & \mathsf{I} \end{bmatrix} \begin{bmatrix} \mathsf{I} & 0 \\ 0 & -\widehat{\mathsf{S}}^{-1} \end{bmatrix},$$

so that, fixed an iteration, given the residual $\mathbf{r} = [\mathbf{r}_1, \mathbf{r}_2]^{\mathsf{T}}$, solving the preconditioner system to compute the preconditioned residual $\mathbf{z} = [\mathbf{z}_1, \mathbf{z}_2]^{\mathsf{T}}$ requires the following steps:

$$\boxed{\begin{aligned} -\widehat{\mathsf{S}}\mathbf{y} &= \mathbf{r}_2 \\ \gamma &= \mathbf{r}_1 + \widetilde{\mathsf{B}}^{\mathsf{T}}\mathbf{y} \\ \mathbf{z}_1 &= \mathsf{D}_{\mathsf{M}_c}^{-1}\gamma \\ \mathbf{z}_2 &= -\mathbf{y}. \end{aligned}} \tag{2.43}$$

The more demanding step is solving the Schur complement linear system. Let us note that $\widehat{\mathsf{S}}$ is sparse, symmetric and definite positive and so we can exploit some efficient methods like a sparse Cholesky factorization or the conjugate gradient method. The matrix $\widehat{\mathsf{S}}$ is well-conditioned and the code solves the linear system through some iterations of conjugate gradient with a diagonal preconditioner.

Now we introduce the other preconditioner implemented in the code. It is based on an inexact block LU factorization of the matrix of the system. Starting from the factorization

$$\begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^\mathsf{T} \\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} \mathsf{M}_c & 0 \\ \widetilde{\mathsf{B}} & \mathsf{S} \end{bmatrix} \begin{bmatrix} \mathsf{I} & \mathsf{M}_c^{-1}\widetilde{\mathsf{B}}^\mathsf{T} \\ 0 & \mathsf{I} \end{bmatrix},$$

the idea is again to substitute M_c and S with proper approximations, that in our case are $\widehat{\mathsf{M}} = \mathsf{D}_{\mathsf{M}_c}$ and $\widehat{\mathsf{S}} = -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\mathsf{D}_{\mathsf{M}_c}^{-1}\widetilde{\mathsf{B}}^\mathsf{T})$. So that the preconditioner is

$$\mathsf{P} = \begin{bmatrix} \mathsf{D}_{\mathsf{M}_c} & 0 \\ \widetilde{\mathsf{B}} & -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\mathsf{D}_{\mathsf{M}_c}^{-1}\widetilde{\mathsf{B}}^\mathsf{T}) \end{bmatrix} \begin{bmatrix} \mathsf{I} & \mathsf{D}_{\mathsf{M}_c}^{-1}\widetilde{\mathsf{B}}^\mathsf{T} \\ 0 & \mathsf{I} \end{bmatrix}. \tag{2.44}$$

For the matrix-vector product (1) the formula (2.42) is employed, while for the operation (2) it's easy to recover the following steps:

$$\boxed{\begin{aligned} \mathbf{y}_1 &= \mathsf{D}_{\mathsf{M}_c}^{-1}\mathbf{r}_1 \\ -\widehat{\mathsf{S}}\mathbf{y}_2 &= \widetilde{\mathsf{B}}\mathbf{y}_1 - \mathbf{r}_2 \\ \mathbf{z}_1 &= \mathbf{y}_1 - \mathsf{D}_{\mathsf{M}_c}^{-1}\widetilde{\mathsf{B}}^\mathsf{T}\mathbf{y}_2 \\ \mathbf{z}_2 &= \mathbf{y}_2. \end{aligned}} \tag{2.45}$$

Let us point point out that the required operations are basically the same as in the block triangular preconditioner with an additional $\mathsf{D}_{\mathsf{M}_c}^{-1}$ multiplication. The Schur complement system is solved in the same way as before.

Remark 2.2. *Another possible technique to solve the system (2.38) is the so called "Schur complement reduction". In this case the two algebraic equations of the linear system are handled by substitution to obtain a reduced linear system in only the pressure variable. The matrix of this pressure system is the Schur complement* S *and then, if it's necessary to compute the velocity, another system with matrix* M_c *can be solved. In this case to avoid the storage of* S *an iterative solver must be used to solve the* S *system, because in an iterative routine only the action of* S *over a vector is needed and this latter can be implemented inverting the matrix* M_c*, i.e. solving a linear system with* M_c *matrix. At every iteration one must solve an* M_c *system (for computing the action of* S *over a vector) plus the system involving the preconditioner of the Schur complement. Instead, solving iteratively the whole system (the one with the* A *matrix) does not require, at each iteration, the inversion of* M_c *but only of its approximation* $\widehat{\mathsf{M}}_c$*, that is much easier (in our case is trivial because* $\widehat{\mathsf{M}}_c = \mathsf{D}_{\mathsf{M}_c}$*). This is the reason because it has been preferred focusing on this latter approach.*

Chapter 3

The C++ code

The project starts from an existing code implementing the TPFA scheme of FV in a 3D fractured porous medium discretized via a polyhedral mesh (the numerical model was the 3D extension of the work developed in [11]). The original program solves, via the FV method, both the steady and pseudo-steady problem with direct solvers (in any case all the work done for the project refers to the steady state). In the code there was also a first simple implementation of MFD in primal form in the bulk and the coupling counditions were implemented in a particular case ($\xi = 1$). With this project we have converted the implementation of MFD in a mixed form, in order to compute also the velocity (the possibility to have a better approximation of velocity is one of the main advantage of using MFD instead of FV) and we have extended the coupling conditions to the most general form. Actually most of the work in this sense has been carried out in order to have the mimetic code in a C++ object-oriented framework of programming. Then the other main goal of the project has been to develop a proper iterative solver because in a 3D framework computations with direct solvers become unfeasible mainly due to memory limitation. To implement some iterative methods we have integrated the `IML++` library in the code; taking into account that the code makes use of the `Eigen` library for the matrix storage and the matrix-vector algebra, therefore the `IML++` library has been adapted to the `Eigen` framework. Clearly an iterative solver at this type of complexity must be properly preconditioned and so the two preconditioners introduced in section 2.5 of chapter 2 have been implemented in the code.

As already pointed out, in order to make the code more efficient, if the solver used is iterative the matrix of the system is stored in a block way, i.e. only the three blocks of the matrix are stored, while if the solver is of direct type the whole matrix of the system must be necessarily stored. The fact that the storage format of the matrix must be choosen at run time, depending on the type of solver selected by the user, makes the class structure of the C++ code more complex from different point of views, as we will point out later.

With respect to the code works developed in [11] and [10] all the part in term of iterative solver is clearly new, but also the assembly of the matrices has been developed in a different way in order to increase the efficiency of the code.

A first important aspect is clearly the mesh. The mesh is read from a `.fvg` file (see section 3.6 for the description of this format) and stored in an object of class `Mesh3D`, which follows up the class developed in the `CGAL` library (see [12] for the details). It allows the storage of a generic unstructured polyhedral mesh, but its data structures are too flexible to ensure efficiency in the assemlage process. The reason is simply that the `Mesh3D` class implements a mesh flexible to refinement and modification, indeed a lot of data in the `Mesh3D` are stored through the `map` and `set` utilities of the standard library. To build up a mesh that is rigid in term of modifications but allows an efficient assembly by the numerical method, the class `Rigid_Mesh` has been developed. In practice all the mesh information are stored using the `vector` container of the standard library, ensuring in this way a fast access to the data required for the assembly process, like a cell, a facet of a cell, the centroid of a cell or the normal of a facet. The class `Rigid_Mesh` has been widely described both in [11] and [10], the main differences are that in our case `Rigid_Mesh` is no more a template class on the problem dimension, because we focus only on the 3D case, and, more important, that in a 3D framework facets and edges are two different things. In the code proper classes for cells and facets have been implemented, but in addition a proper class `edge` has been developed, toghether with several other classes to distinguish the different types of edge. We stress the fact that in any case the programming structure of the class `Rigid_Mesh` is very similar to the one described in [11] and [10] and, because there are many other new classes interesting to describe, we skip its presentation.

3.1 The numerical operators

In [11] and [10] for every matrix M_c, B, C and T (and so for every numerical operator represented by that matrix) a particular class has been developed. This is an interesting way of using the C++ object-oriented programming to reproduce in the code the mathematical structure of the model. In [11] and [10] every class has a method `fillMatrix` that assembles the matrix of the numerical operator in the monolithic matrix of the system. In practice to assemble the system it's necessary to define a proper object for every discrete operator (the mimetic inner product, the weak divergence, the coupling conditions...) and to build its matrix directly in the system. Let us now recover the mimetic inner product matrix M (we are not considering the modifications due to coupling conditions) and B, which are assembled through the local matrices M_P and B_P (see (2.21)-(2.23) and (2.36)). The best way of proceeding for assembling the bulk problem is:

- Do a loop over the cells of the mesh.
    (1) Do a loop over the facets of the cell to build up M_P and B_P.
    (2) Do a loop over the facets of the cell to assemble M_P and B_P in M and B.

The assembly step (2) is equivalent to what is done normally in FEM schemes (so one recovers the global IDs of the degrees of freedom of the element and assemble them in the global matrix). If the utilities of the code are only single classes that handle the assembly of a specific matrix of the problem, a routine that builds up M and B jointly with only one cell loop and two cell-facets loops is not possible. In practice for the bulk problem the

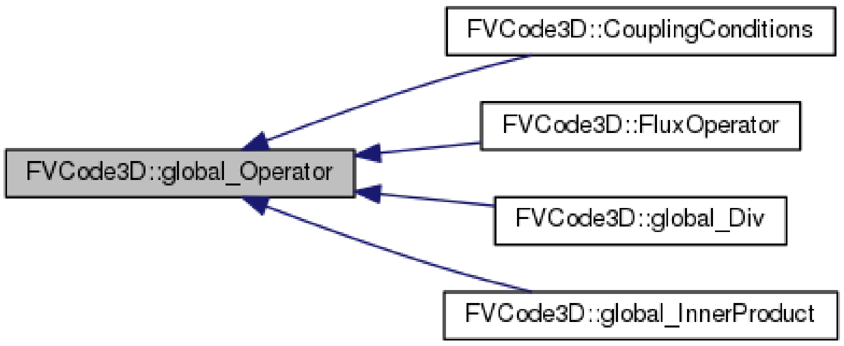
FVCode3D::CouplingConditions
FVCode3D::FluxOperator
FVCode3D::global_Operator
FVCode3D::global_Div
FVCode3D::global_InnerProduct

Figure 3.1: Inheritance graph for global_Operator.

routine pointed out before is repeated two times, one for M and the other for B and so we are cycling more times both on cells and facets. A similar consideration can be done for the assembly in fracture, where the best way would be to assemble with only one loop over the fracture facets the matrix C, the modification of M due to coupling conditions and the matrix T. In the code it has been attempted to keep the single classes of the discrete operators in order to have a well-structured and more extensible code and, at the same time, to provide utilities in order to assemble M and B jointly with only one loop over the bulk cells and C, the modification of M and T jointly with only one loop over the fracture facets, in order to have a more efficient procedure. This has been implemented through specific builder classes, and this is a notable improvement with respect to [11] and [10].

3.1.1 The class `global_Operator`

Four classes have been implemented, each of them represents a numerical operator and collects the utilities necessary to assemble its matrix. All operator classes inherit from an abstract base class `global_Operator`; the inheritance diagram is showed in Figure 3.1. The term "global" reflects the fact that these classes assemble their matrices on all the mesh (for the mimetic operators there's also a local version of the operators). The class `global_Operator` is breafly shown below.

```
class global_Operator
{
public:
    global_Operator(const Rigid_Mesh & rMesh, dType Prow, dType Pcol);
    global_Operator(const global_Operator &) = delete;
    global_Operator() = delete;
    virtual ~global_Operator() = default;

    virtual void ShowMe() const;
    void CompressMatrix();

    virtual void reserve_space() = 0;
    virtual void assemble() = 0;

protected:
    const Rigid_Mesh          & M_mesh;
    dType                       row_policy;
    dType                       col_policy;
    UInt                        Nrow;
    UInt                        Ncol;
    SpMat                       M_matrix;
};
```

The attributes are `protected` in order to be accessible by the derived classes; in the derived classes the attribute will be kept `private`. The attributes `row_policy` and `col_policy` are two discretization policies, one for the matrix rows and the other for the matrix columns, that indicate which type of degrees of freedom are used by the operator (note that we use the term operator but "in principle" we are talking about discrete bilinear form, at least for the bulk problem that is in weak form). This policy is introduced in the code with a proper `enum`:

```
enum dType{dFacet, dCell, dFracture};
```

It follows the meaning of each discretization policy:

- `dFacet` means a facet-based discretization.

- `dCell` means a bulk cell-based discretization.
- `dFracture` means a fracture cell-based discretization.

The class stores the number of rows in `N_row` and the number of columns in `N_col` (`UInt` is simply a `typedef` for `unsigned int`). The matrix dimensions are set at run time using the `enum dType` in the following way

```
Nrow( (row_policy==dFacet)*(M_mesh.Facets()+M_mesh.FrFacets())+(row_policy
    ==dCell)*(M_mesh.Cells())+(row_policy==dFracture)*(M_mesh.FrFacets()) )
```

For the `Ncol` is clearly the same. The `M_mesh` is a constant reference to the `Rigid_Mesh` and plays an essential role in the assembly; it's also fundamental to size properly the matrix as seen above. It's important to point out that `dFacet` takes into account the decoupling of fracture facets, that, in other words, is the decoupling of the velocity degrees of freedom for the fracture facets. We add to the N_f facets degrees of freedom of velocity N_Γ additional degrees, one for every fracture facet. Given a facet f of a polyhedron P that is matching a fracture cell, we state the following convenction:

- If $\alpha_{\mathsf{P},\mathsf{f}} = \mathbf{n}_\mathsf{f} \cdot \mathbf{n}_{\mathsf{P},\mathsf{f}} > 0$, then the facet is a plus facet f_+ and it keeps its ID.
- If $\alpha_{\mathsf{P},\mathsf{f}} = \mathbf{n}_\mathsf{f} \cdot \mathbf{n}_{\mathsf{P},\mathsf{f}} < 0$, then the facet is a minus facet f_- and its ID is $\mathsf{N}_\mathsf{f} + \mathsf{ID}_\Gamma$, where ID_Γ is the identifier of the facets f_- as a fracture cell.

In practice the decoupled degrees of freedom are added at the tail of the velocity vector.

The method `ShowMe` prints out the basic information of the operator, i.e. the discretization policies and the number of degrees of freedom. It is `virtual` because every class will overload it adding other information to the printing base method and it is `const` because it does not modify the attributes of the class, making it `const` allows us to call it also from `const` objects and from `const` references to objects.

In terms of constructors only one constructor is implemented and both the default and the copy constructor are deleted, because a default constructor cannot initialize a reference and the copy makes sense only for the matrix and not for the other attributes. In this sense we will have appropriate `get` methods through which the matrix can be copied if necessary (the `get` methods are not shown here). The destructor is defaulted and `virtual`, in order to make possible deleting an instance of a derived class through a pointer to the base class. For all the derived classes we have exactly the same schemes in terms of constructors and destructor (this latter will be defaulted but not virtual), so in many cases we will not show them.

Let us now focus on the matrix. The code is fully adapted for the `Eigen` library and the `Eigen` utilities are used to store the sparse matrix. The matrix is stored as an `SpMat` that is a proper `typedef` for an `Eigen` sparse matrix:

```
typedef Eigen::SparseMatrix<Real> SpMat;
```

where `Real` is symply a `typedef` for a `double`. The `SpMat` format offers high performance and low memory usage; it implements a more versatile variant of the widely-used Compressed Column Storage scheme. It consists of four compact arrays, the first two of which collect the basic information of non zero values and their row indices; for more details see [13]. Another important aspect with respect to the Compressed Column Storage scheme is the free space available to quickly insert new elements. When the matrix is completely assembled this space can bee freed with a `Eigen makeCompressed` method, used in the class in the `CompressMatrix` method. The `SpMat` is defined without the template parameter that indicates the type of storage because the default, the column storage, is better for our purposes, as already pointed out, as it allows to use some efficient direct solvers, like the UmfPack, and also for the simple fact that all `Eigen` algorithms have been tested more extensively on column-major matrices. There are basically two ways of inserting elements in an `SpMat`, one is to use a vector of `Eigen::Triplet` that stores for every non zero three information: its value, its row position and its column position; then with the method `setFromTriplets` the matrix is fully assembled automatically; regarding this we stress that if a triplet is repeated in the vector the method `setFromTriplets` sums up the corresponding values. The other technique is to insert directly every non zero element in the correct position with an `insert` or `coeffRef` method (the second if accumulation of values is necessary). Because of low memory consumption this latter method is more efficient if the sparsity pattern is easily computable beforehand, and this is our case because the sparsity pattern of our matrix is always related to the topology of the grid and it is easily computable with an appropriate loop (it is necessary to know the number of non zeros for every column). This latter technique has been used in the code to fill the matrix. The pure virtual method `reserve_space` has to be overridden to compute the sparsity pattern and store the proper space (this is done via the `Eigen` method `reserve`). The pure virtual `assemble` method has to be overridden to perform the assembly of the matrix `M_matrix`, after the `reserve_space` has been called.

To build the matrix directly in the system a method `fillMatrix(SpMat & MAT)` (where `Mat` is the matrix of the system) would be necessary; we have not introduced it in the class because, as we have already pointed out, the matrices of the bulk and that of the fractures are assembled jointly with less iterations. We have only included the `reserve_space` and `assemble` methods to allow, given a numerical operator object, to build the operator matrix `M_matrix` individually. So that, given e.g. an inner product object, it's possible to build the inner product matrix a part from the system. Note that in the assembly of the system these methods are not used, because only the matrix of the system is stored and there's no need to store the individual operator matrices, so that `M_matrix` is kept void. What is actually done is to use appropriate builder classes, that employ the utilities of the single operators to build the matrices of the problem directly in the system in a more efficient way.

3.1.2 The class `global_InnerProduct`

Let us starts from the derived class `global_InnerProduct` that implements the mimetic inner product, i.e. the matrix M. The coupling conditions modifications, which allow to obtain the matrix M_c, are added to M by the class `Coupling Conditions` that we analyze later on. The class does not have new attributes and its main `public` methods are:

```
void assembleFace(const UInt iloc, const Mat & Mp, const Rigid_Mesh::
    Cell & cell);
void assembleFace(const UInt iloc, const Mat & Mp, const Rigid_Mesh::
    Cell & cell,SpMat & S, const UInt rowoff, const UInt coloff) const;

void reserve_space();
void assemble();
```

It overrides the two pure virtual methods `reserve_space` and `assemble` and implements some assembly utilities in the overloaded method `assembleFace`. Let us point out the assembly algorithm:

- Do a loop over the mesh cells.
    1. Build the local matrix M_P.
    2. Assemble M_P into the global matrix M.

The methods assembleFace takes care of step (2). The first version of the assembleFace method takes the following input data: the local facet ID iloc, the local matrix Mp and a constant reference to the cell P cell (the class Rigid_Mesh::Cell implements a 3D cell and it is nested into the Rigid_Mesh class). The method assembles the local contribution corresponding to the facet iloc in the global matrix M (that is the M_matrix attribute in global_Operator). To do this the row and column $\mathsf{M}(1,\texttt{iloc:end})$ and $\mathsf{M}(\texttt{iloc:end},1)$ are assembled in the global matrix M. The second version of assembleFace is actually the one that will be used, because it assembles the local contribution of the facet not in M_matrix, but in the matrix S that takes in input; the assembly is carried out with the row offset rowoff and column offset coloff. In practice this assembleFace version alllows us to build up the M block inside the matrix S, whatever position the block M occupies. The matrix S may be the system matrix in the case of a direct solver or the M_c block matrix that, together with the other blocks $\widetilde{\mathsf{B}}$ and $\widetilde{\mathsf{T}}$, represents the system matrix storage in the case of an iterative solver. This version of the method is const because it does not modify the attributes of the class, infact the assembly is carried out in S and not in the M_matrix attribute.

Step (1) is performed through a concrete class local_InnerProduct. It inherits from a base class local_MimeticOperator that contains a reference to the cell and to the Rigid_Mesh; we do not list it here. Let us show here the code of the class.

```
class local_InnerProduct: public local_MimeticOperator
{
public:
    local_InnerProduct(const Rigid_Mesh & rMesh, const Rigid_Mesh::Cell &
         cell);
    local_InnerProduct(const local_InnerProduct &) = delete;
    local_InnerProduct() = delete;
    ~local_InnerProduct() = default;

    void free_unusefulSpace();
    void assemble();

private:
    Mat3                          Np;
    Mat3                          Rp;
    Mat                           Mp0;
    Mat                           Mp1;
    Mat                           Mp;
    static constexpr Real         gamma = 2.;
};
```

It collects all the possible data about the local mimetic inner product. So we have five private attributes that are the matrices involved in the construction of the operator: the $\mathsf{N_P}$ and $\mathsf{R_P}$ matrices that are built through some geometrical quantities of the cell (see (2.13) and (2.17)), the consistency matrix $\mathsf{M_P^0}$ (see (2.22)), the stability matrix $\mathsf{M_P^1}$ (see (2.23)) and the final local inner product matrix $\mathsf{M_P}$ araising from the summation of the previous two matrices. In the code the Mat refers to an Eigen dense matrix of dynamic size, while the Mat3 refers to an Eigen dense matrix of dynamic row size and fixed (equal to three) column size. We have added in the class all the matrices involved in the construction of $\mathsf{M_P}$ because one may be interested in studying them (the theory behind the construction of $\mathsf{M_P}$, e.g. its spectral properties, are a challanging topic: see [3]). In principle the storage as attributes of these matrices is not required, so after the creation of $\mathsf{M_P}$ via the assemble method, we simply erase them through the method free_unusefulSpace. Finally we comment about the static attribute, it is constexpr and it represents a multiplication factor in the γ_P expression in (2.23) that is the 2 factor in the formula. The 2 value has been estimated as the best one in [3]. It is `static` because it can be considered a constant of the scheme. Whereas for the constructors and destructor we can do exactly the same considerations already done in the `global_Operator` case.

Let us turn back focusing on `global_InnerProduct`. We list below the method `assemble` that implements the assembly algorithm:

```
void global_InnerProduct::assemble()
{
    for (auto& cell : M_mesh.getCellsVector())
    {
        // Define the local inner product
        local_InnerProduct localIP(M_mesh, cell);
        // Assemble the local inner product
        localIP.assemble();
        // Erase Np,Rp,Mp0,Mp1
        localIP.free_unusefulSpace();
        // Assemble the local matrix in the global one
        for(UInt iloc=0; iloc<cell.getFacetsIds().size(); ++iloc)
            assembleFace(iloc, localIP.getMp(), cell);
    }
}
```

3.1.3 The class `global_Div`

We do only few comments on the class `global_Div` because it is about the same of the previous class `global_InnerProduct`. The main methods are:

```
void assembleFace(const UInt iloc, const std::vector<Real> & Bp, const
    Rigid_Mesh::Cell & cell);
void assembleFace(const UInt iloc, const std::vector<Real> & Bp,
    const Rigid_Mesh::Cell & cell, SpMat & S, const UInt rowoff, const
    UInt coloff, const bool transpose = true) const;
SpMat getTranspose() const;

void reserve_space();
void assemble();
```

The second version of the `assembleFace` method has an additional parameter `transpose` that indicates if we want to assemble in S also B^T; in practice it must be used `transpose` = `true` in the case in which the monolithic matrix of the system is assembled (direct solver case); otherwise it's not necessary to assemble B^T, because B will be assembled directly in the matrix $\widetilde{\mathsf{B}}$ and $\widetilde{\mathsf{B}}^T$ will be obtained through an appropriate method. The other difference with respect to `global_Operator` is the method `getTranpose`, that gives out the B^T through the `Eigen` method `M_matrix.transpose()`. Let us note that as for the `global_InnerProduct` there's a local counterpart of this operator, it is the `local_Div` class that inherits from `local_MimeticOperator`, it builds up the B_P and it is much simpler then `local_InnerProduct`, we dot not show it; clearly it is used in the `assemble` method of `global_Div`.

3.1.4 The class `CouplingConditions`

The class `CouplingConditions` takes care of the application of the coupling conditions and so of the modification of M and of the creation of the matrix C. The class with its main methods is shown here:

```
class CouplingConditions: public global_Operator
{
```

```
public:
    CouplingConditions(const Rigid_Mesh & rMesh, dType Prow, dType Pcol,
        SpMat & IP_matrix);
    //Copy and default constructor deleted, destructor defaulted

    void Set_xsi(const Real & xsiToSet) throw();
    void assembleFrFace_onM(const Rigid_Mesh::Fracture_Facet & facet_it);
    void assembleFrFace_onM(const Rigid_Mesh::Fracture_Facet & facet_it,
         SpMat & S, const UInt rowoff, const UInt coloff) const;
    void assembleFrFace(const Rigid_Mesh::Fracture_Facet & facet_it);
    void assembleFrFace(const Rigid_Mesh::Fracture_Facet & facet_it, SpMat
        & S, const UInt rowoff, const UInt coloff, const bool transpose =
        true) const;

    void reserve_space();
    void assemble()

private:
    SpMat                           & M;
    Real                              xsi;
    static constexpr Real             Default_xsi = 0.5;
};
```

The class has an attribute that is a reference to the inner product matrix M in order to modify it to obtain the matrix M_c, then it has a `Real xsi` that is the ξ parameter of the coupling conditions and there's a static attribute `Default_xsi` that is the default value of ξ. The reference to the `SpMat` is set in the constructor, while the `xsi` parameter can be set through the method `Set_xsi`, that throws an exception if the parameter is not such that $0 \leq \xi \leq 1$. In any case it is defaulted using the `static` attribute in the constructor of the class.

The assembly algorithm for the coupling conditions is:

- Do a loop over the fracture facets.
  1. Modify M with the contributions of the fracture facet.
  2. Assemble in C the contributions due to the fracture facet.

Given a fracture facet `facet_it` (`Rigid_Mesh::Fracture_Facet` is a proper class nested into `Rigid_Mesh`) the `assembleFrFace_onM` takes care of the step (1), while the `assembleFrFace` takes care of the step (2). Both the methods have a version that assembles the fracture facet contributes directly in an S matrix using the offsets `rowoff` and `coloff`. In the case of `assembleFrFace` there's also a `bool` to indicate if also the construction of C^T in S is necessary. We list here the simple `assemble()` method:

```
void CouplingConditions::assemble()
{
    for (auto& facet_it : M_mesh.getFractureFacetsIdsVector())
    {
        // Modify M
        assembleFrFace_onM(facet_it);
        // Assemble C
        assembleFrFace(facet_it);
    }
}
```

3.1.5 The class `FluxOperator`

Now we focus on the FV assembly over the fracture facets. The class dedicated to do this is `FluxOperator` whose main methods are here shown.

```
Real findAlpha (const UInt & facetId, const Rigid_Mesh::Edge_ID * edge)
    const;
Real findDirichletAlpha (const UInt & facetId, const Rigid_Mesh::
    Edge_ID * edge) const;
Real findFracturesAlpha (const Fracture_Juncture fj, const UInt n_Id)
    const;
void assembleFrFace(const Rigid_Mesh::Fracture_Facet & facet_it);
void assembleFrFace(const Rigid_Mesh::Fracture_Facet & facet_it, SpMat
    & S, const UInt rowoff, const UInt coloff) const;

void reserve_space();
void assemble();
```

Let us point out first some geometrical considerations. Consider an edge of a fracture facet, if the edge is internal with respect to the boundary of the fracture domain we call it juncture edge, if it belongs to that boundary we call it tip edge: we will call it border tip edge if it belongs also to the boundary of the bulk domain, otherwise we call it internal tip edge. On a border tip edge the boundary condition corresponding to the bulk boundary is imposed, while on a internal tip edge an homogenous Neumann condition is imposed. A juncture edge here is treated with the `public typedef`:

```
typedef std::pair<UInt,UInt> Fracture_Juncture;
```

that is a couple of IDs that are the identifiers of the two verteces of the edge.

The assembly algorithm to build up T is:

- Do a loop over the fracture facets.
  (1) Do a loop over the juncture edges of the facet.
  (2) Do a loop over the fracture facets separeted by the edge.
  (3) Compute the transmissibility and assemble it in T.

The alpha-methods implement the computation of the α coefficient involved in the transmissibility formula (see definitions (2.31) and (2.32)). The `findFracturesAlpha` method computes the α coefficient corresponding to the juncture edge `fj` and the fracture facet that has `n_Id` as identifier. The `findAlpha` and `findDirichletAlpha` implement the same in the case of a border tip edge; these methods are used in the imposition of boundary conditions, that is handled by a specific class on which we do not focus now. The `edge` that these methods receive is the border tip edge of the fracture facet `facet_it`, in particular here we have a pointer to `Rigid_Mesh::Edge_ID`, that is a base class of a complex inheritance tree (that here we skip) implementing the different types of edge. The `findAlpha` computes the α corresponding to the fracture cell adjacent to the boundary, while the `findDirichletAlpha` computes the one corresponding to the ghost cell. The `assembleFrFace` methods handle the (1)-(3) steps that are the core of the FV assembly, so that it's very easy implementing the assembly:

```
void FluxOperator::assemble()
{
    for (auto& facet_it : M_mesh.getFractureFacetsIdsVector())
        // Assemble the transmissibilities in T
        assembleFrFace(facet_it);
}
```

3.2 The builders of the system

In the code the builders are those classes that are dedicated to the actual assembly of the matrix of the system. We have two basic builders: the `global_BulkBuilder` class for the assembly of the bulk problem and the `global_FractureBuilder` class for the assembly of the fracture problem. Both of them inherit from a base abstract class `global_Builder`, listed here.

```cpp
 class global_Builder
{
public:
    global_Builder(const Rigid_Mesh & rMesh);
    global_Builder(const global_Builder &) = delete;
    global_Builder() = delete;
    virtual ~global_Builder() = default;

    virtual void reserve_space(SpMat & S)=0;
    virtual void build(SpMat & S)=0;

protected:
    const Rigid_Mesh  & M_mesh;
};
```

It has only one attribute that is a reference to the `Rigid_Mesh`, which will be used by the derived classes in the assembly process; for this reason it is `protected`. A builder does not store any data, it builds the data structure that are passed as argument to its methods. For this reason a copy constructor does not make sense and so it's deleted togheter with the default one, that could not initialize the reference; only one constructor that initilizes the reference is kept. The destructor is defaulted and `virtual`.

We have two pure virtual method: `reserve_space` and `build`. The first one computes the sparsity patterns of the interested matrices and store the proper space in `S`, the second actually build the interested matrices in `S`, that here must be intended as the monolithic matrix of the system. So these two methods refer only to the case of a direct solver. The iterative case is handled by methods implemented in the derived classes because the number of arguments required is variable.

3.2.1 The class `global_BulkBuilder`

The class `global_BulkBuilder` is dedicated to the assembly of the bulk problem. It inherits from `global_Builder` and it is listed below.

```cpp
class global_BulkBuilder: public global_Builder
{
public:
    global_BulkBuilder(const Rigid_Mesh & rMesh, global_InnerProduct & ip,
        global_Div & div);
    // Copy and default constructor deleted, destructor defaulted

    void reserve_space(SpMat & S);
    void build(SpMat & S);
    void reserve_space(SpMat & M, SpMat & Bt);
    void build(SpMat & M, SpMat & Bt);

private:
    global_InnerProduct   & IP;
    global_Div            & Div;
};
```

It has two attributes: `IP` is a reference to the `global_InnerProduct` class, `Div` is a reference to the `global_Div` class; through them the builder employs the utilities implemented in the inner product and divergence operators to build the bulk problem. Both the references are clearly initialized in the unique constructor of the class.

Let us point out the assembly scheme that employs only one cell loop to build both the inner product and the divergence operators.

- Do a loop over the cells of the mesh
  1. Compute M_P and B_P.
  2. Assemble M_P and B_P in M and B.

The class overloads the methods **`reserve_space`** and **`build`** in the case of only one argument `S`, that represents the monolithic matrix of the system. The **`reserve_space`** method computes the sparsity pattern of the M, B and B^T matrices and stores the proper space for these matrices in `S`, while the method **`build`** assembles the three matrices in the matrix `S`. The other two methods refer to the iterative solver case, in which we are interested only to store the block matrices of the saddle point problem, in the bulk case the inner product matrix M (the coupling conditions will be added by the fracture builder) and the block $\widetilde{\mathsf{B}}$ (see section 2.4 in chapter 2); the argument `M` and `Bt` in input represent these two block matrices. So in that case the **`reserve_space`** method computes the sparsity pattern of M and B and stores the proper space in the blocks `M` and `Bt` in input, while the **`build`** method assembles the inner product matrix in the argument `M` and the divergence matrix B in the block `Bt` (that is the block $\widetilde{\mathsf{B}}$).

Let us show the method **`build`** for the iterative solver case (more interesting), for the direct case the method is about the same, only the row offset and the `bool transpose` in the `assembleFace` of the divergence operator change, to accomodate B and B^T in the monolithic matrix S.

```
void global_BulkBuilder::build(SpMat & M, SpMat & Bt)
{
    for (auto & cell : M_mesh.getCellsVector())
    {
        // Define the local inner product
        local_InnerProduct    localIP(M_mesh,cell);
        // Define the local divergence
        local_Div             localDIV(M_mesh,cell);
        // Define the local builder
        local_builder         localBUILDER(localIP,localDIV,cell);
        // Build the local matrices
        localBUILDER.build();
        // Erase Np,Rp,Mp0,Mp1
        localIP.free_unusefulSpace();
        // Assemble the local matrices in the global ones
        for(UInt iloc=0; iloc<cell.getFacetsIds().size(); ++iloc)
        {
            IP.assembleFace(iloc,localIP.getMp(),cell,M,0,0);
            Div.assembleFace(iloc,localDIV.getBp(),cell,Bt,0,0,false);
        }
    }
}
```

First of all we note that, as already pointed out, the assembly is carried out with only one cell loop. Moreover also the M_P and B_P are built together, i.e. only one loop over the facets of the cell is done to build them. This is performed through an object `localBUILDER` of class `local_builder`; we skip the code of this class here, but it is very simple and the idea is exactly the same of the global builder: using the utilities defined in the local operator classes `local_InnerProduct` and `local_Div` to build $\mathsf{M_P}$ and $\mathsf{B_P}$ in a more efficient way. Also the assembly of the local matrices in the global ones is performed with only one loop over the facets of the cell. Note that the offsets in the `assembleFace` methods are all zero because the inner product matrix and the divergence matrix occupy the position (0,0) in the blocks M and $\widetilde{\mathsf{B}}$. The `false` bool argument passed to the `assembleFace` of the divergence operator indicates that B^T must not be built.

3.2.2 The class `FractureBuilder`

The class `FractureBuilder` inherits from the `global_Builder` class. It implements the assembly of the fracture problem and it has the same structure of the previous class; so we conduct a faster survey of this class. As attributes now we have two references to the fracture operators `CouplingConditions` and `FluxOperator`:

```
private:
    CouplingConditions    & coupling;
    FluxOperator          & FluxOp;
```

The assembly algorithm with only one loop over the fracture facets is:

- Do a loop over the fracture facets
  (1) Modify M with the contributions of the fracture facet.
  (2) Assemble in C the contributions of the fracture facet.
  (3) Do a loop over the juncture edges of the facet.
  (4) Do a loop over the fracture facets separeted by the edge.
  (5) Compute the transmissibility and assemble it in T.

We have grouped up the steps due to the assembly of coupling conditions and of FV in fratcures.

Now we show the methods, that basically are equivalent to that of the previous class.

```
    void reserve_space(SpMat & S);
    void build(SpMat & S);
    void reserve_space(SpMat & M, SpMat & Bt, SpMat & Tt);
    void build(SpMat & M, SpMat & Bt, SpMat & Tt);
```

The first `reserve_space` and `build` methods reserve the proper space and assemble the modifications of M, the matrices C, C^T and T directly in the `S` argument, that is the monolithic matrix of the system. While the second version of the two methods reserve the proper space and assemble the coupling modification in the M matrix (to obtain the M_c block), the matrix C in the block $\widetilde{\mathsf{B}}$ and the matrix T in the block $\widetilde{\mathsf{T}}$, where the blocks $\widetilde{\mathsf{B}}$ and $\widetilde{\mathsf{T}}$ are the arguments `Bt` and `Tt`.

It follows the `build` method for the iterative solver case.

```
void FractureBuilder::build(SpMat & M, SpMat & Bt, SpMat & Tt)
{
    for (auto& facet_it : M_mesh.getFractureFacetsIdsVector())
    {
        // Modify M
        coupling.assembleFrFace_onM(facet_it,M,0,0);
        // Assemble C-constributions in Bt
        coupling.assembleFrFace(facet_it, Bt,M_mesh.Cells(),0,false);
        // Assemble T-contributions in Tt
        FluxOp.assembleFrFace(facet_it,Tt,M_mesh.Cells(),M_mesh.Cells());
    }
}
```

The two `assembleFrFace` methods of `coupling` perform the (1)-(2) steps; note that in `assembleFrFace` we have a row offset equal to the number of cells of the mesh N_{P}, because of the position of the matrix C in the block $\widetilde{\mathsf{B}}$; the `bool false` indicates to avoid the assembly of C^{T}. The `assembleFrFace` method of `FluxOp` takes care of the steps (3)-(5); there are two offsets equal to N_{P} because of the position of T in $\widetilde{\mathsf{T}}$.

3.2.3 The class `SaddlePoint_StiffMatHandler`

Now we introduce the class that uses all the utilities described up to here to build up the whole matrix of the system. Actually we have two classes that have this goal: the class `StiffMatHandlerMFD`, that builds the matrix in the case of a direct solver, and the class `SaddlePoint_StiffMatHandler`, that does the same thing in the case of an iterative solver. The class `StiffMatHandlerMFD` assembles the matrix of the system in the monolithic form and so a matrix stored as an `SpMat`. The class `SaddlePoint_StiffMatHandler` assembles a matrix that is stored in blocks form and that is handled with a specific class called `SaddlePointMat`. The class `StiffMatHandlerMFD` belongs to an inheritance tree in which also the case of a full FV (both on bulk and fractures) method is implemented through a specific derived class; that's because until a direct solver is used the storage of the matrix is always an `SpMat` and the full FV code supports only direct solvers up to now. We skip this inheritance structure and focus only on the iterative solver case and so on the `SaddlePoint_StiffMatHandler` class.

This class is a builder because it does not store the matrix of the system, but it builds it through a reference. This choice has been done to avoid the copy of the matrix; an alternative would have been using the move semantic but up to now the `Eigen` doesn't support it for sparse matrices. With this choice we can easily avoid to copy or move the matrix, that is built directly in the proper data structure.

We show below the class code.

```
class SaddlePoint_StiffMatHandler
{
public:
    SaddlePoint_StiffMatHandler(const Rigid_Mesh & pmesh, SaddlePointMat
        & Msp, Vector & b, const BoundaryConditions & bc);
    // Copy and default constructors deleted, destructor defaulted

    void setDofs(const UInt Mdim, const UInt Btrow);
    void assemble();

private:
    const Rigid_Mesh                & M_mesh;
    const BoundaryConditions        & M_bc;
    Vector                          & M_b;
    SaddlePointMat                  & M_SP;
};
```

In terms of attributes we see a constant reference to the `Rigid_Mesh`, always needed, and a constant reference to the `BoundaryConditions`, which is a map that stores for every border of the bulk domain the boundary function that must be applied; we do not focus on this latter class, it leads the same principles of the class `BCcond` in [10]. The need of this reference is due to the fact that the class `SaddlePoint_StiffMatHandler` applies also the boundary conditions on the system; for this reason we have an attribute `M_b` that is a reference to the right-hand side vector of the system. The class `Vector` is a `typedef` for an `Eigen` dense vector of dynamic size, .i.e.

```
typedef Eigen::VectorXd Vector;
```

Finally we have `M_SP` that is a reference to the matrix of the system, that is stored as a `SaddlePointMat`.

In terms of constructors, as usual up to here, we have only one constructor that initializes the reference attributes and we have deleted the copy and default constructor, while the destructor is defaulted.

The `setDofs` method sizes with `Mdim` and `Btrow` the `M_SP` matrix; note that to size the blocks M_c, $\widetilde{\mathsf{B}}$ and $\widetilde{\mathsf{T}}$ the dimension of M_c and the number of rows of $\widetilde{\mathsf{B}}$ are enough.

The method `assemble` does all the job, assembling the whole matrix of the system. It is listed here:

```
void SaddlePoint_StiffMatHandler::assemble()
{
    // Rename the block matrices
    auto & M = getM();
    auto & Bt = getBt();
    auto & Tt = getTt();

    // Define the global inner product
    global_InnerProduct gIP(M_mesh, dFacet, dFacet);
    gIP.ShowMe();
    // Define the global divergence operator
    global_Div gDIV(M_mesh, dCell, dFacet);
    gDIV.ShowMe();
    // Define the coupling conditions
    CouplingConditions coupling(M_mesh, dFracture, dFacet,gIP.getMatrix());
    coupling.ShowMe();
    // Define the flux operator
    FluxOperator fluxOP(M_mesh, dFracture, dFracture);
    fluxOP.ShowMe();

    // Define the global bulk builder
    global_BulkBuilder gBulkBuilder(M_mesh, gIP, gDIV);
    // Reserve space for the bulk matrices
    gBulkBuilder.reserve_space(M, Bt);
    // Build the bulk matrices
    gBulkBuilder.build(M, Bt);

    // Define the fracture builder
    FractureBuilder FBuilder(M_mesh, coupling, fluxOP);
    // Reserve space for the fracture matrices
    FBuilder.reserve_space(M, Bt, Tt);
    // Build the fracture matrices
    FBuilder.build(M, Bt, Tt);

    // Define the BCs imposition object
    BCimposition BCimp(M_mesh, M_bc);
    // Impose BCs on bulk
    BCimp.ImposeBConBulk(M, Bt, M_b);
    // Impose BCs on fractures
    BCimp.ImposeBConFracture_onT(Tt, M_b, fluxOP);

    // Compress the saddle point matrix
    M_SP.makeCompressed();
}
```

For a matter of clarity the three matrices stored in M_SP are renamed through three references that represent the blocks M_c, $\widetilde{\mathsf{B}}$ and $\widetilde{\mathsf{T}}$. Then the four numerical operators of the problem are defined in order to make their utilities accessible. We show the basic information (the discretization policies and the number of degrees of freedom) calling the method ShowMe for every operator. A global_BulkBuilder is defined and used to build the bulk problem and, in the same way, a FractureBuilder is defined and used to build the fracture problem. After that the matrix of the system needs to be modified properly in order to impose the boundary conditions; this is done through a class BCimposition. It requires for its construction a reference to the Rigid_Mesh and to the BoundaryConditions. It imposes the boundary conditions on bulk through the method ImposeBConBulk, and on fractures through the method ImposeBConFracture. Imposing the boundary conditions on bulk requires the modification of the blocks M_c and $\widetilde{\mathsf{B}}$ and of the right-hand side of the system; in particular the Dirichlet condition on the pressure affects only M_b, while the Neumann, imposed through the Nitsche method, affects the two blocks and M_b. For the fractures only the block $\widetilde{\mathsf{T}}$ and the right-hand side are modified; we recall that the Neumann condition affects only M_b, while the Dirichlet, imposed through the ghost cell approach, affects both the block and the vector. We do not show here the class BCimposition. Finally the three block matrices M_c, $\widetilde{\mathsf{B}}$ and $\widetilde{\mathsf{T}}$ are compressed through the method makeCompressed of M_SP.

Let us now conduct a brief discussion on the class SaddlePointMat, that implements the storage in block form of the matrix of the system, viewed as a saddle point matrix. The class has three attributes that are the three blocks M_c, $\widetilde{\mathsf{B}}$ and $\widetilde{\mathsf{T}}$, stored as SpMat:

```
private:
    SpMat        M;
    SpMat        Bt;
    SpMat        Tt;
```

This class represents a saddle point matrix and so to make it flexible we have implemented different constructors and clearly it must be copyable. We have a defaulted constructor, one that allows to build a SaddlePointMat, given the three blocks, and another constructor that sizes the three blocks, given the dimension of M_c and the number of rows of $\widetilde{\mathsf{B}}$; this latter is useful to avoid any copy. The copy constructor and the destructor are defaulted. These utilities are shown here:

```
SaddlePointMat() = default;
SaddlePointMat(const SpMat & Mmat, const SpMat & Btmat,
    const SpMat & Ttmat);
SaddlePointMat(const UInt Mdim, const UInt Btrow);
SaddlePointMat(const SaddlePointMat &) = default;
~SaddlePointMat() = default;
```

We do not show the methods of the class, we just say that it contains some get methods to access the blocks and other methods through which the matrices can be resized, compressed, it's possible to have the number of non zeros and so on. Here we show only the overload of the matrix-vector product, which is performed in terms of the three blocks as in formula (2.42).

```
Vector operator * (const Vector & x) const
{
    Vector result(M.rows()+Bt.rows());
    result.head(M.rows()) = M*x.head(M.cols()) + Bt.transpose()*x.tail(Bt.
        rows());
    result.tail(Bt.rows()) = Bt*x.head(M.cols()) + T*x.tail(Bt.rows());
    return result;
}
```

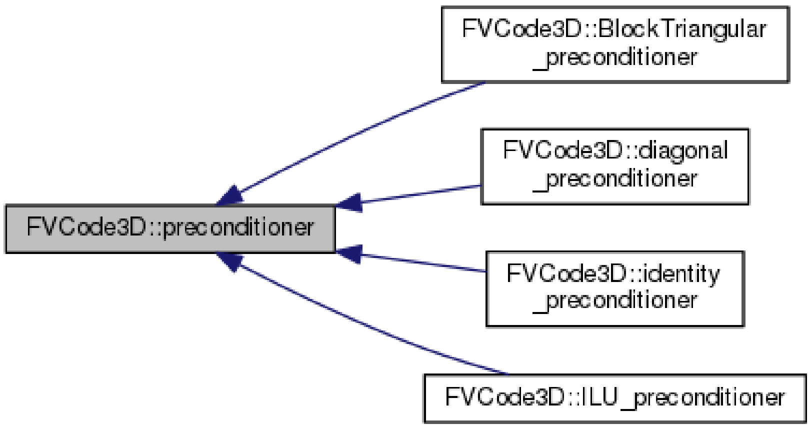
FVCode3D::BlockTriangular_preconditioner
FVCode3D::diagonal_preconditioner
FVCode3D::preconditioner
FVCode3D::identity_preconditioner
FVCode3D::ILU_preconditioner

Figure 3.2: Inheritance graph for preconditioner.

The `Eigen` methods `head` and `tail` allow to extract a top or bottom sub-vector; while the $\widetilde{\mathsf{B}}^{\mathsf{T}}$ is obtained through the `Eigen` method `transpose`. The overload of the matrix-vector product is very important because it's one of the two ingredients necessary for the system preconditioning (see section 2.5 in chapter 2).

3.3 The preconditioners

Now we introduce the classes that implement the preconditioning. We have already shown the overload of the matrix-vector product for the class `SaddlePointMat`, that is the first operation needed to solve a linear system with a preconditioned iterative method. The second operation is solving the linear system $\mathsf{P}\mathbf{z}^k = \mathbf{r}^k$ and basically a preconditioner class must provide a method solve that, given a vector $\mathbf{r}^k$, solves this linear system and outputs the vector $\mathbf{z}^k$. The abstract base class `preconditioner` of the inheritance tree in Figure 3.2 collects just the pure virtual method that performs this operation:

```
virtual Vector solve(const Vector & r) const = 0;
```

The classes `identity_preconditioner` and `diagonal_precondtioner` implement two trivial preconditioners. Using the first means simply solving the linear system without a preconditioner, while the second uses as preconditioner the diagonal part of the matrix of the system. Both of them give usually very poor results in terms of convergence, i.e. a very huge amount of iterations are needed to obtain the convergence with a proper tolerance. This is due to the fact that the matrix of the system is ill-conditioned and a more accurate preconditioner is needed to obtain a faster convergence. In any case these two basic preconditioners are available in the code. The two other preconditioners implemented in the code are the block triangular preconditioner and the ILU preconditioner, both based on the diagonal part of M_c, introduced in the section 2.5 in chapter 2. The first is implemeted in the class `BlockTriangular_preconditioner` and the second in the class `ILU_preconditioner`.

Let us make a comment on the construction of the preconditioner. The code user gives a string that indicates what preconditoner must be used and, at run time, the proper object is selected among the inheritance tree through a factory. The factory is implemented through a class `proconHandler` that is a singleton and builds up a proper map that associates every possible string to the proper preconditioner class. A proxy is used to actually fill the map and it is implemented in the class `preconProxy<T>`, that is a template class on the type of preconditioner. This is a classical factory pattern about which we do not enter the details, see [1] for an in-depth survey on singletons and factory pattern. In any case the preconditioner is defined using the factory and it is defaulted, then its attributes are set through a proper method.

Now we show the `BlockTriangular_preconditioner` class:

```
class BlockTriangular_preconditioner: public preconditioner
{
public:
    BlockTriangular_preconditioner();
    BlockTriangular_preconditioner(const BlockTriangular_preconditioner &)
        = delete;
    ~BlockTriangular_preconditioner() = default;

    void setMaxIt(const UInt itmax);
    void set_tol(const UInt Tol);
    void set(const SaddlePointMat & SP);

    Vector solve(const Vector & r) const;

private:
    const SpMat *                  Btptr;
    DiagMat                        Md_inv;
    SpMat                          ISC;
    UInt                           MaxIt;
    Real                           tol;
    static constexpr UInt          MaxIt_Default = 200;
    static constexpr Real          tol_Default = 1e-2;
};
```

Let us note that to perform the solve operation, that is reported in formula (2.43), we need three matrices: the block $\widetilde{\mathsf{B}}$, the diagonal part of the inner product matrix $\mathsf{D}_{\mathsf{M}_c}$ (or better its inverse) and the inexact Schur complement $\widehat{\mathsf{S}} = -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\mathsf{D}_{\mathsf{M}_c}^{-1}\widetilde{\mathsf{B}}^{\mathsf{T}})$. So we have three attributes that are: `Btptr`, that is a constant pointer to the block $\widetilde{\mathsf{B}}$, `Md_inv`, that is $\mathsf{D}_{\mathsf{M}_c}^{-1}$, and `ISC`, that is $\widehat{\mathsf{S}}$. Note that we have used a `typedef` and `DiagMat` is actually an `Eigen` diagonal matrix, the best format of storage for a diagonal matrix in the `Eigen` framework. The `Btptr` is clearly put to `nullptr` in the default constructor; we have not used a smart pointer because the resource $\widetilde{\mathsf{B}}$ is stored and properly deleted in the class `SaddlePointMat`, which represents the matrix of the system and is stored in the class solver that we describe in the next section. We employs a pointer and not a reference because, using the factory, the preconditioner is first defined through the default constructor and then properly set up, but we could not properly initialize a reference in the default constructor. To set up the preconditioner the following method is used:

```
void set(const SaddlePointMat & SP)
{
    Bptr    = & SP.getB();
    Md_inv = SP.getM().diagonal().asDiagonal().inverse();
    ISC     = - SP.getB() * Md_inv * SP.getB().transpose();
    ISC    += SP.getT();
}
```

It takes the `SaddlePointMat SP`, that is the matrix of the system, and it sets `Bptr`, it extracts the inverse of the diagonal part of M_c and it assemble the $\widehat{\mathsf{S}}$. Note the `Eigen` methods used to compute the $\mathsf{D}_{\mathsf{M}_c}^{-1}$: `diagonal` extracts the diagonal part of M_c as a vector, `asDiagonal` converts it to a diagonal matrix and `inverse` gives the inverse diagonal matrix.

As we can see in formula (2.43) solving the linear system with GMRES requires at each iteration the inversion of a linear system that has $\widehat{\mathsf{S}}$ as matrix. The matrix $\widehat{\mathsf{S}}$ is sparse, symmetric and positive definite and to solve its system the conjugate gradient method of the Eigen is used; the attributes MaxIt and tol refer to the iterative solution of the $\widehat{\mathsf{S}}$ linear system. MaxIt indicates the maximum number of iterations of the CG and tol is the tolerance used. They can be set through the methods setMaxIt and set_tol, in any case they are set in the empty constructor, that gives them deafult values. The default values are implemented via the static constexpr attributes MaxIt_Default and tol_Default. The $\widehat{\mathsf{S}}$ matrix is well-conditioned and so a fast convergence is expected; moreover, having to solve this system at each iteration of the GMRES, the solver must be as fast as possible. The $\widehat{\mathsf{S}}$ system is solved through some iterations of the CG, by default 200 iterations with a tolerance of 10^{-2}.

In terms of constructors we have a default constructor, used to initialize the pointer Btptr and the MaxIt and tol attributes, the copy constructor is deleted and the destructor is defaulted.

We report the method solve that implements the steps of formula (2.43).

```
Vector BlockTriangular_preconditioner::solve(const Vector & r) const
{
    auto & Bt = *Btptr;
    // First step: solve Inexact Schur Complement linear system
    Eigen::ConjugateGradient<SpMat> cg;
    cg.setMaxIterations(MaxIt);
    cg.setTolerance(tol);
    cg.compute(-ISC);
    Vector y2 = cg.solve(r.tail(Bt.rows()));
    // Second step: solve the diagonal linear system
    Vector z(Md_inv.rows()+Bt.rows());
    z.head(Md_inv.rows()) = Md_inv*(r.head(Md_inv.rows())+Bt.transpose()*y2
      );
    z.tail(Bt.rows()) = -y2;
    return z;
}
```

Let us note that for the CG method the Eigen CG solver is used; it employs a diagonal preconditioner as default. An object cg of type Eigen::ConjugateGradient<SpMat> is defined and then the maximum number of iterations and the tolerance are set. The method compute gives to the solver cg the matrix of the system and the solution is obtained through the solve method that takes the right-hand side vetor. The other step involves only matrix-vector product multiplications.

We do not show the class ILU_preconditioner beacuse it is exactly equal to the BlockTrangular_preconditioner class, only the method solve changes in order to implements the formula 2.45.

3.4 The solvers

A solver is a class dedicated to the solution of the linear system arising from the discretization of the problem. It stores the algebraic system, i.e. the matrix, the right-hand side and the solution. We have an abstract base class Solver that represents an asbtract interface of all the possible solvers; its inheritance diagram is shown in Figure 3.3. Solver has two children that are DirectSolver and IterativeSolver; they are again abstract classes that add methods and attributes to represent a generic direct or iterative solver. DirectSolver has three children: EigenUmfPack, which implements the UmfPack solver, EigenLU, which implements the LU sparse factorization and EigenCholesky, which implements the sparse cholesky factorization. Let us note that this latter method cannot be used in our case, because the matrix of the system is not definite positive; it is available in the code for the case of fully FV scheme, that produces a symmetric and definite positive matrix. The IterativeSolver has two children: imlBiCGStab implements the BiCGStab method through the version implemented in the IML++ library and imlGMRES is the same with the GMRES method.

3.4.1 The class Solver

Let us describe the base class Solver, that is reported below.

```
class Solver
{
public:
    Solver(const UInt nbDofs = 0);
    Solver(const Vector & b);
    Solver(const Solver &) = default;
    virtual ~Solver() = default;

    virtual void setDofs(const UInt nbDofs);
    virtual void solve() = 0;

protected:
    Vector          M_b;
    Vector          M_x;
};
```

The class stores as protected attributes the right-hand side and the solution of the linear system; they are both stored as Vector. Note that this class doesn't store the matrix of the system, because its type of storage depends on the type of solver: if the solver is direct then the matrix is stored as an SpMat, if it is iterative the matrix is stored as a SaddlePointMat. So the two children of Solver will store the matrix in the two possible ways.

We have a constructor Solver(const UInt nbDofs = 0) that sizes with nbDofs the two vectors of the class initializing them to zero. The default argument of this constructor is zero and in that case the constructor is the empty one and makes the two vectors of zero dimension. We have another constructor Solver(const Vector & b) that sets the right-hand side M_b to b. The copy constructor makes sense because one may want to copy the system and so it is defaulted. Finally the destructor is virtual because this is a base class and it is defaulted.

The setDofs method resizes with nbDofs the system, it is virtual because it will be overloaded by the derived classes, that will have to resize also the matrix. Finally the

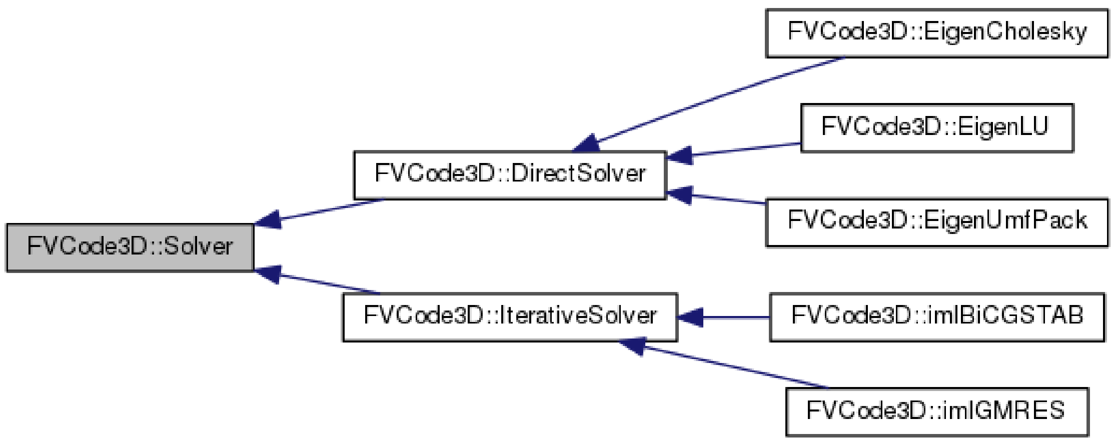

Figure 3.3: Inheritance graph for Solver.

method `solve` is pure virtual and it will be overridden to actually solve the linear system.

Remark 3.1. *If for a generic solver is not clear the type of storage of the matrix (monolithic or block form), in principle one could make `Solver` a template class on the type of matrix. In this case the derived class would have been `DirectSolver<SpMat>` and `IterativeSolver<SaddlePointMat>`. This approach has not been chosen because it makes more difficult referring to the base class (that would be a template class) and employing the polimorphism. We stress that the polimorphism is used also to create the solver, because, given a user defined string, to select the solver among the possible choices a factory is used. The factory is about the same of the one implemented for the preconditoner.*

3.4.2 The class `IterativeSolver`

The inheritance tree of `Solver` is quite complex and we describe only the iterative part, because more interesting. We start from the abstract class `IterativeSolver`:

```
class IterativeSolver : public Solver
{
public:
    IterativeSolver();
    IterativeSolver(const UInt Mdim,const UInt Btrow);
    IterativeSolver(const SaddlePointMat & A, const Vector & b );
    virtual ~IterativeSolver() = default;

    void setDofs(const UInt Mdim, const UInt Btrow);
    void set_precon(const std::string prec);
    void setMaxIter(const UInt maxIter);
    void setTolerance(const Real tol);
    virtual void print() const;

    virtual void solve() = 0;

protected:
    SaddlePointMat              M_A;
    UInt                        M_maxIter;
    UInt                        M_iter;
    Real                        M_tol;
    Real                        M_res;
    UInt                        CIndex;
    preconPtr_Type              preconPtr;
    static constexpr Real       S_referenceTol = 1e-6;
    static constexpr UInt       S_referenceMaxIter = 1000;
};
```

The class collects the matrix of the system stored as a `SaddlePointMat`, i.e. stores the matrix in a blocks form based on the saddle point nature of the system. Then we have some coefficients: `M_maxIter`, which indicates the maximum number of iterations, `M_iter`, which is the number of iteration actually employed to solve the system, `M_tol`, which represents the tolerance required to reach convergence, `M_res`, which is the actual residual error and finally `CIndex`, which indicates whether the convergence has been reached or not (a value of 0 means convergence within `M_maxIter`, a value of 1 means that the convergence has not been reached in the specified `M_maxIter`). The parameters `M_maxIter` and `M_tol` are set through the methods `setMaxIter` and `setTolerance`, in any case they are set to default values in all the constructors. The default values are implemented as `static constexpr` attributes and they are 10^{-6} for the tolerance and 1000 for the maximum number of iterations. The attributes `M_iter`, `M_res` and `CIndex` are set after the resolution of the system, while before the constructors give them zero value.

This class represents an iterative solver and so it must employ a preconditioner; to this purpose it has as attribute a polymorphic object `preconPtr`, that is a smart pointer to the `preconditioner` class. The `typedef` employed is:

```
typedef std::shared_ptr<preconditioner> preconPtr_Type;
```

The pointer is set through the method `set_precon` that, given the string `prec`, employes the preconditioner factory to actually construct the preconditioner. The method is:

```
void set_precon(const std::string prec)
{
    preconPtr = preconHandler::Instance().getProduct(prec);
    preconPtr->set(M_A);
}
```

Note that after the creation of the polymorphic object the preconditioner is set up through the method `set`.

In terms of constructors we have: a default constructor, which gives default values to the parameters of the solver, a constructor `IterativeSolver(const UInt Mdim,const UInt Btrow)`, which sizes the system given the dimension of M and the number of rows of B and a constructor `IterativeSolver(const SaddlePointMat & A, const Vector & b )`, which sets the matrix and the right-hand side of the system. Finally the destructor is `virtual` and defaulted.

Let us comment about the other methods. The method `setDofs` resizes the system given `Mdim` and `Btrow`. Note that while in `solver` (and also in the derived direct solver classes that we don't show here) only an argument `nbDofs` is needed to resize the system, here two argument are needed because of the different storage of the matrix. The `print` method prints some information about the computation as `M_iter`, `M_res` and `CIndex`. It is virtual because for the class `imlBiCGStab` more information are printed out. The method `solve` is again pure virtual.

3.4.3 The class `imlGMRES`

We describe the class `imlGMRES`, that inherits and implements the GMRES iterative method employing the template GMRES method of the `iml++` library. The class `imlBiCGStab` is implemented about in the same way and so we skip it.

The `imlGMRES` adds the attributes:

```
private:
UInt m;
static constexpr UInt Default_m = 300;
```

The `m` indicates the restart value and `Default_m` is the default value of the restart. As usual `m` can be set through a proper method and, in any case, it is set to the default value in the constructors. The GMRES version implemented in the code allows, having performed a certain number of iterations, to restart the computation taking the current solution as the initial data of the new computation. In the GMRES routine the full basis of the Krylov spaces must be stored, so restarting is important to avoid an excessive memory consumption; in practice except in case of fast convergence the restarting is necessary. It must be noted that restarting avoids to store a huge amounts of vectors, but it compromises the convergence of the method and, in general, much more iterations are needed to reach the required tolerance.

The class overrides the method `solve` that actually solves the system:

```
void solve();
```

The method `solve` simply employs the template function of the `IML++` library to solve the system. For more details about the library see [15]. The library that we use has been fully updated for the Eigen framework, in order to avoid the overload of some matrix-vetor operations otherwise required. In any case the usage of the functions of the library is the same. The template function of the GMRES is:

```
template <class Matrix, class Vector, class Preconditioner>
UInt GMRES(const Matrix & A, Vector & x, const Vector & b, const
    Preconditioner & M, UInt & m, UInt & max_iter, Real &tol);
```

We have three templates: the `Matrix`, which is the type of the matrix of the system, the `Vector`, which is the type of vectors of the system (i.e. the right-hand side and the solution), and the `Preconditioner`, which is the class implementing the precondtioner. The `Matrix` class must supply the matrix-vector product and the preconditioner must supply a method solve for the solution of the system $\mathsf{P}\mathbf{z}^k = \mathbf{r}^k$. The arguments that the function takes in input are the matrix of the system `A`, the solution `x`, the right-hand side `b`, the preconditioner `M`, the restart level `m`, the maximum number of iteration `max_iter` and the tolerance required `tol`. The GMRES gives out an `UInt` that indicates if the convergence has been reached in `max_iter` iterations and after the call of the function the vector `x` contains the computed solution, `max_iter` counts the carried out iterations and `tol` the current residual error.

For the the BiCGStab solver we only say that with respect to GMRES the convergence is not ensured and two types of breakdown can happen. In these cases the computation ends up giving out a 2 or 3 value depending on the type of breakdown, so in that case `CIndex` of the class `IterativeSolver` will have a value different from 0 or 1. A patch to the classical BiCGStab has been applied in our code in order to restart the computation if a first type of breakdown happens, i.e. in the case that the current residual becomes orthogonal to the initial one. So in the code it is possible to activate this type of restart, otherwise the algorithm is the classical BiCGStab.

3.5 The class `Problem`

The upper level classes seen up to now are: the `Solver` class, which stores the algebraic system and solves it, and the `SaddlePoint_StiffMatHandler`, which assembles the matrix of the sytem. However these classes are not handled directly in the main program of the software; indeed we have implemented a class `Problem`, which contains the solver, makes use of the `SaddlePoint_StiffMatHandler` to assemble the system and handles the imposition of the source term modifying the right-hand side of the system. It is an abtract base class and the derived classes depend on the type of problem (steady or pseudo-steady), remember that the code handles also the unsteady case for the fully FV scheme. The class `Problem` is a template, where the `QRMatrix` is the quadrature rule used for the forcing term over the bulk and `QRFracture` is the quadrature rule for the forcing term over the fractures.

Before showing the class `Problem` we briefly discuss the treatment of the forcing term. We have two classes that handle the interpolation of the source term: the class `Quadrature` and the class `QuadratureRule`. `QuadratureRule` is an abstract class and every concrete quadrature is its child. It contains the pure virtual methods `clone` and `apply`. The first will be overridden to actually pass the `QuadratureRule` to the class `Quadrature` through a `std::unique_ptr<QuadratureRule>`, while the methods `apply` needs to integrate the forcing term over a cell or a facet using the quadrature rule. The signatures are

```
virtual Real apply(const Cell & cell, const std::function<Real(Point3D)> &
   integrand) const = 0;
virtual Real apply(const Facet & facet, const std::function<Real(Point3D)>
   & integrand) const = 0;
```

Up to now only a derived class `CentroidQuadrature`, that applies the midpoint rule, has been implemented. In the midpoint rule the forcing term is approximated with its value in the barycenter of the cell or facet. Note that the midpoint rule is available on generical polyhedral elements as the ones the code faces up, while it's not easy to accomodate more accurate quadrature rule on these types of elements. Moreover as long as the mimetic scheme is of low order a low order approximation of the forcing term makes sense, otherwise more degrees of freedom for every cell would be necessary.

Let us sketch rapidly the class `Quadrature`, which implements methods to integrate functions over the mesh. Its important attributes are

```
QR_Handler M_quadrature;
QR_Handler M_fractureQuadrature;
```

where `QR_Handler` is a `unique_ptr` to the class `QuadratureRule`. `M_quadrature` is the quadrature rule for the bulk, while `M_fractureQuadrature` is the quadrature rule for the fractures. They are both set up in the constructors of the class using the method `clone` of the `QuadratureRule`. The class collects a lot of methods to integrate the source term over the mesh, even just over the bulk mesh or over the fracture mesh, both for a forcing term stored as a vector or in form of a function. Moreover, also the L2 norm is computable. Here we report two methods, that are the ones actually used for interpolating the source term.

```
Vector cellIntegrateMatrix (const Func & func);
Vector cellIntegrateFractures (const Func & func);
```

These two methods integrate a function and return the integral cell by cell, the first is for the porous matrix, the second is for the fractures. Note that `Func` is a typedef for:

```
typedef std::function<Real(Point3D)> Func;
```

where `Point3D` is a simple class for a point in the three-dimensional space. The utility of the standard library `function` is a class that can wrap any kind of callable element (such as functions and function objects) into a copyable object, and whose type depends solely on its call signature (and not on the callable element type itself).

Let us start showing the class `Problem` with its main features.

```
template <class QRMatrix, class QRFracture>
class Problem
{
public:
    Problem() = delete;
    Problem(const Problem &) = delete;
    Problem(const std::string solver, const Rigid_Mesh & mesh, const
        BoundaryConditions & bc, const Func & func, const DataPtr_Type &
        data);
    virtual ~Problem() = default;

    SpMat & getMatrix() throw();
    SaddlePointMat & getSaddlePointMatrix() throw();
    Vector & getRHS();
    virtual void assembleMatrix() = 0;
    virtual void assembleVector() = 0;
    void assemble();
    virtual void solve() = 0;
    void assembleAndSolve();

protected:
    const Rigid_Mesh & M_mesh;
    const BoundaryConditions & M_bc;
    const Func & M_func;
    const Data::NumericalMethodType M_numet;
    const Data::SolverPolicy M_solvPolicy;
    const Data::SourceSinkOn M_ssOn;
    std::unique_ptr<Quadrature> M_quadrature;
    SolverPtr_Type M_solver;
};
```

The attributes are `protected` in order to make them accessible by the derived classes. We have a constant reference to the `Rigid_Mesh` and to the `BoundaryConditions`, necessary to assemble the system. Then we have a constant reference to the forcing term that is implemented as a `Func`. In this way it is possible to treat the forcing term just as a classic function of three variables. We have also three attributes to indicate the numerical method, the solver policy and where the source term is applied. Each of them is of a particular type that is defined as an `enum`. The three `enum` are:

```
enum NumericalMethodType{FV, MFD}
enum SolverPolicy{Direct, Iterative};
enum class SourceSinkOn{Matrix, Fractures, Both, None};
```

Let us note that the source term can be applied on both matrix or fracture. The `SourceSinkOn` is an `enum class` to prevent name collisions. Then we have a `unique_ptr` to the `Quadrature` class and a `shared_ptr` to the `Solver` class that is

```
typedef std::shared_ptr<Solver> SolverPtr_Type;
```

In terms of constructors both the default and the copy constructors don't make sense and so they are deleted, there's only one constructor that initializes the first three reference attributes, fills the three `enum` attributes, given `data`, and constructs the solver class through the factory, given the `string solver`; the smart pointer to the `Quadrature` is set to `nullptr`. The pointer `data` is a `unique_ptr` to the class `Data`, that in short is a huge collector of all the data of the problem. Finally the destructor is virtual and defaulted.

Let us describe the `assemble` and `solve` methods. The methods `assembleMatrix`, `assembleVector` and `solve` are all pure virtual and must be overridden in the derived classes. The method `assembleMatrix` builds up the numerical scheme by assembling and filling the matrix and the right-hand side with the boundary conditions. To do that it defines and uses the proper builder (like `SaddlePoint_StiffMat` in the case of an iterative solver for the MFD scheme). The method `assembleVector` has to set up the `Quadrature` object and interpolate the source term over the mesh by modifying the right-hand side. The method `assemble` simply calls the latter two methods. Finally the method `solve` has to solve the system using the solver, and `assembleAndSolve` simply calls `assemble` and `solve`.

After the assembly of the system it's possible to get the matrix and the right-hand side through `M_solver`. Regarding that we stress the fact that it's not easy getting the matrix, because the storage format of the matrix is not known at compile time (infact it depends on the type of solver). The problem is that `M_solver` is a pointer to the base `Solver` class, that stores the vector and returns it through an appropriate method, but it cannot return the matrix of the system because it doesn't store it. So to obtain the matrix it is necessary to convert the pointer from `Solver*` to `DirectSolver*`, if the solver is direct, or to `IterativeSolver*`, if the solver is iterative. This can be done through the run-type identification operator `dinamic_cast`, after having checked the type of solver. So, while for the right-hand side we have only one method `getRHS`, for the matrix we have two methods: `getMatrix` and `getSaddlePointMatrix`, the first must be used in the direct solver case and the other in the iterative solver case. Both of them throw an exception if called in the wrong case. So before using the methods for getting the matrix a check on the solver type is necessary.

Regarding the classes derived from `Problem` we are interested only in the steady state case, because the hybrid MFD-FV scheme here presented has still not been set for the unsteady case. The class is `DarcySteady<QRMatrix,QRFracture>` and it overloads the three pure virtual methods of `Problem` to assemble and solve the system. Regarding the methods `assembleMatrix`, it defines the proper builder and assembles the system depending on the scheme (MFD-FV or fully FV) and the type of solver (direct or iterative).

The C++ description of the code is completed, in the following sections we give some practical information on the usage of the code.

3.6 Input parameters

In this section we describe the input data of the software. The program requires:

- an input mesh in `.fvg` format;
- the data file `data.txt`;
- the definition of the forcing function and of the functions of the boundary conditions.

The file format `.fvg` consists of four parts:

- list of POINTS, each of them described by its coordinates x, y, z;
- list of FACETS, each of them described by a list of ids of POINTS, plus some properties if the facet represents a fracture;
- list of CELLS, each of them described by a list of ids of FACETS, plus some properties;
- list of FRACTURE NETWORKS, each of them described by a list of FACETS that belong to the same network.

The properties related to the facets that represent a fracture are the permeability, the porosity and the aperture.The properties related to the cells (porous medium) are the permeability and the porosity. These properties are included directly in the mesh format `.fvg`. It must be noted that in any case it's not smart to generate another mesh every time a change in the properties (in the permeability e.g.) is required. So the properties can be also set in the main program, together with the boundary conditions.

The data file consists of of seven sections:

- **mesh**: the input file parameters such as mesh location and filename;
- **output**: the output directory and filename;

- **problem parameter**: the perameters related to the problem, such as the problem type (steady or unsteady) or if apply or not the source term;
- **numerical method**: the numerical method: MFD or FV;
- **fluid**: mobility and compressibility of the fluid;
- **bc**: the rotation to apply around the z axis, needed to correctly apply the boundary conditions;
- **solver**: the solver to use and some parameters such as the tolerance, the maximum number of iterations and the preconditioner if the solver is iterative.

Below a brief description of each parameter is given:

```
[mesh]
mesh_dir     = ./data/              // mesh directory
mesh_file    = polyGrid3.fvg        // mesh filename
mesh_type    = .fvg                 // mesh format

[output]
output_dir   = ./results/           // output directory
output_file  = sol                  // output prefix of the files

[numet]
method       = MFD                  // FV or MFD

[problem]
type         = steady               // steady or pseudoSteady
fracturesOn  = 1                    // 1 enable fractures, 0 disable fractures
sourceOn     = none                 // where the source is applied: all,matrix
    ,fractures,none
fracPressOn  = 0                    // 1 set pressure inside fracture, 0
    otherwise
fracPress    = 1.                   // if fracPressOn=1, this set the value of
     the pressure
perm_size    = ScalarPermeability   //ScalarPermeability,DiagonalPermeability,
    SymTensorPermeability,FullTensorPermeability
initial_time     = 0.               // if type=pseudosteady, this set the
    initial time
end_time     = 2.e6                 // if type=pseudosteady, this set the
    final time
time_step    = 1.e5                 // if type=pseudosteady, this set the time
     step

[fluid]
mobility     = 1.                   // mobility of the fluid
compressibility  = 1.               // compressibility of the fluid

[bc]
theta        = 0.                   // rotation to apply to grid around the z

[solver]
policy   = Iterative                // Iterative,Direct
type     = imlGMRES                 // EigenCholesky,EigenLU,EigenUmfPack,
    imlCG,imlBiCGSTAB,imlGMRES

    [./iterative]
    maxIt        = 20000            // max iterations of the iterative solver
    tolerance    = 1e-6             // tolerance of the iterative solver
    preconditioner = ILU            // Identity,Diagonal,BlockTriangular,ILU
```

Finally, the functions used to set the forcing term and the boundary conditions are defined in `functions.hpp`. For example, the following line

```
Func SourceDomain = [](Point3D p){return 5.*(p.x()*p.x() + p.y()*p.y() + p
    .z()*p.z()) < 1;};
```

defines a function equal to 5 inside the sphere of radius 1 centered in the origin, zero elsewhere. Instead, these lines define the functions $f(x) = 0$, $f(x) = 1$ and $f(x) = -1$.

```
Func fZero = [](Point3D){return 0.;};
Func fOne = [](Point3D){return 1.;};
Func fMinusOne = [](Point3D){return -1.;};
```

3.7 Tutorial

In this section we describe how to use the code, giving an example of a main program. We stress that, to use the code in practice, it's necessary the employment of some classes that have not been described; e.g. to import the mesh an importer object is needed, while to export the results of the computation an exporter object must be defined. We don't focus on these classes, but we show a practical example of usage.

The classes involved in the usage of the code are:

- **Data**: it collects all the parameters used by the program that can be set through the datafile;
- **Importer**: it reads the input file mesh;
- **PropertiesMap**: it collects the properties of the porous medium and of the fractures;
- **Mesh3D**: it stores the geometrical mesh as Point3D, Facet3D and Cell3D;
- **Rigid_Mesh**: it converts a Mesh3D in a rigid format, suitable to assemble the problem;
- **BoundaryConditions**: it defines the boundary conditions to assign to the problem; the boundary conditions can be Dirichlet or Neumann;
- **Problem**: it defines the problem to solve; it can be a steady problem (DarcySteady) or a time dependent problem (DarcyPseudoSteady); it assembles the matrix and the right-hand side;
- **Exporter**: it exports as `.vtu` files the mesh, the properties, the solution and more.

Now we show an example of usage of the code. Let us suppose to be interested to simulate a steady state flow with the hybrid MFD-FV scheme studied in this report.

Preliminarly we define two typedefs for the problem:

```
typedef Problem<CentroidQuadrature, CentroidQuadrature> Pb;
typedef DarcySteady<CentroidQuadrature, CentroidQuadrature> DarcyPb;
```

The `CentroidQuadrature` is the midpoint quadrature rule, that we use both for bulk and fractures.

First of all we read the data from the file `data.txt`:

```
// Use GetPot to access the datafile
GetPot command_line(argc, argv);
const std::string dataFileName = command_line.follow("data.txt", 2, "-f", "
    --file");
// Read the data
DataPtr_Type dataPtr(new Data(dataFileName));
```

To access the parameters in the datafile the `GetPot` utility has been used; for details on `GetPot` see [14].

We define the `Mesh3D` and the `PropertiesMap`. Then we define the `Importer` and import the mesh and the properties.

```
// Define the mesh and the properties
Mesh3D mesh;
PropertiesMap propMap(dataPtr->getMobility(), dataPtr->getCompressibility()
    );
// Create the importer
Importer * importer = 0;
importer = new ImporterForSolver(dataPtr->getMeshDir() + dataPtr->
    getMeshFile(), mesh, propMap);
// Import the mesh file
importer->import(dataPtr->fractureOn());
```

Note that we create a pointer `importer` to the base class of the importers. This is because in the code different importers are implemented to support different grid formats; with polymorphism it's easy to select the correct importer. In any case in the example we use the importer for the mesh file of `.fvg` extension.

Having read the mesh, it's necessary to perform some operations to process the mesh:

```
// Compute the cells that separate each facet
mesh.updateFacetsWithCells();
// Compute the neighboring cells of each cell
mesh.updateCellsWithNeighbors();
// Set labels on boundary (necessary for BCs)
importer->extractBC(dataPtr->getTheta());
//Compute facet ids of the fractures (creates fracture networks)
mesh.updateFacetsWithFractures();
```

We can define the properties here, in the main program, if necessary:

```
// Define the permeability for bulk and fractures
std::shared_ptr<PermeabilityBase> matrixPerm( new PermeabilityDiagonal );
std::shared_ptr<PermeabilityBase> fracturesPerm( new PermeabilityScalar );
// Set the permability of bulk
matrixPerm->setPermeability( 1., 0 );
matrixPerm->setPermeability( 1., 4 );
matrixPerm->setPermeability( 1., 8 );
// Set the permeability of fractures
const Real kf = 1.e3;
fracturesPerm->setPermeability( kf, 0 );
// Set the aperture (of fractures) and the porosity (of both bulk and
    fractures)
const Real aperture = 1.e-2;
const Real matrixPoro = 0.25;
const Real fracturesPoro = 1;
// Set the properties
propMap.setPropertiesOnMatrix(mesh, matrixPoro, matrixPerm);
propMap.setPropertiesOnFractures(mesh, aperture, fracturesPoro,
    fracturesPerm);
```

Now we define the boundary conditions:

```
// Define boundary condition of each border of the domain
BoundaryConditions::BorderBC leftBC (BorderLabel::Left, Dirichlet, fOne);
BoundaryConditions::BorderBC rightBC(BorderLabel::Right, Dirichlet, fZero);
BoundaryConditions::BorderBC backBC (BorderLabel::Back, Neumann, fZero);
// and so on...

// We define a vector of BorderBC
std::vector<BoundaryConditions::BorderBC> borders;
// and we insert the BCs previously created
borders.push_back(backBC);
borders.push_back(frontBC);
borders.push_back(leftBC);
// and so on...

// We create the boundary conditions
BoundaryConditions BC(borders);
```

We create a `Rigid_Mesh`, necessary to assemble efficiently the problem:

```
//The Rigid_Mesh requires the Mesh3D and the PropertesMap
Rigid_Mesh myrmesh(mesh, propMap);
```

We go ahead defining the problem:

```
// Define the problem
Pb * darcy(nullptr);
darcy = new DarcyPb(dataPtr->getSolverType(), myrmesh, BC, SS, dataPtr);
// Set the iterative parameters in the case of an iterative solver
if(dataPtr->getSolverPolicy() == Data::SolverPolicy::Iterative))
{
    dynamic_cast<IterativeSolver*>(darcy->getSolverPtr())->setMaxIter(
        dataPtr->getIterativeSolverMaxIter());
    dynamic_cast<IterativeSolver*>(darcy->getSolverPtr())->setTolerance(
        dataPtr->getIterativeSolverTolerance());
    dynamic_cast<IterativeSolver*>(darcy->getSolverPtr())->set_precon(
        dataPtr->getpreconType());
}
```

For `Pb` the same consideration on polymorphism, done before about the importer, holds. Note the `dynamic_cast` necessary to convert the pointer to base class to a pointer to derived class if the solver is iterative.

We assemble the problem and we solve it:

```
darcy->assemble();
darcy->solve();
```

We export the solution in terms of pressure:

```
// Define the total number of facets and cells
UInt numFacetsTot  = myrmesh.Facets() + myrmesh.FrFacets.size();
UInt numCellsTot   = myrmesh.Cells() + myrmesh.FrFacets();
//Define the exporter
ExporterVTU exporter;
// Export the solution on bulk
exporter.exportSolution( myrmesh, dataPtr->getOutputDir() + dataPtr->
    getOutputFile() + "_solution.vtu", darcy->getSolver().getSolution()
    .tail(numCellsTot) );
// Export the solution of fractures
exporter.exportSolutionOnFractures(myrmesh, dataPtr->getOutputDir() +
    dataPtr->getOutputFile() + "_solution_f.vtu", darcy->getSolver()
    .getSolution().tail(numCellsTot) );
```

We are exporting the pressure, so we need the tail of the solution vector and we extract it through the `Eigen` method `tail`, taking the last total cells elements.

Finally we delete the instances:

```
delete darcy;
delete importer;
```

Chapter 4

Numerical results

In this chapter we present two numerical examples to test the code. The first one is a single fracture case with a theoretical solution and the goal is to test the order of convergence of the numerical method. This test is carried out with different types of mesh and by varying the permeability of the fractures. Then we show a more realistic simulation in a case of a complex network of fractures; in this case too we vary the permeability, in order to present both a case of a high permeable network and a case of a network of barriers, in addition we make also a comparison between the different solvers of the linear system in order to show the well performance of the block triangular and ILU preconditioner.

4.1 The single fracture test case

This test is a three-dimensional generalization of the one considered in [2], in order to have a solution that depends also from the vertical coordinate z.

Let us consider the domain $\Omega = (-1,1) \times (-1,1) \times (0,1)$, which is the bulk, and the fracture $\Gamma = (-1,1) \times \{0\} \times (0,1)$, which is a vertical plane that cuts the whole domain for $\{y = 0\}$. The bulk permeability tensor is assumed to be the identity matrix, i.e. $\mathsf{K} = \mathsf{I}$, while the fracture permeability tensor is taken as $\mathsf{K}_\Gamma = k_f \mathsf{I}$, where k_f is a positive real number. This type of choice allows to stress the bulk-fractures permability constrast that the code has to reproduce. We take as the forcing term the following function:

$$f(x,y,z) = \begin{cases} (1-k_f)cosh\Big(\dfrac{l_\Gamma}{2}\Big)cos(x)cos(z) & in\ \Omega \\ k_f^2 cos(x) + k_f(1-k_f)cosh\Big(\dfrac{l_\Gamma}{2}\Big)cos(x)cos(z) & in\ \Gamma, \end{cases} \tag{4.1}$$

so that the exact solution is given by

$$p(x,y,z) = \begin{cases} k_f cos(x)cosh(y) + \dfrac{1-k_f}{2}cosh\Big(\dfrac{l_\Gamma}{2}\Big)cos(x)cos(z) & in\ \Omega \\ k_f cos(x) + \dfrac{1-k_f}{2}cosh\Big(\dfrac{l_\Gamma}{2}\Big)cos(x)cos(z) & in\ \Gamma. \end{cases} \tag{4.2}$$

The boundary conditions over the bulk are full Dirichlet and are determined by imposing the exact solution; over the fracture the boundary conditions are imposed accordingly.

Let us note that, in spite of the permeability discontinuity accross Γ, the test case is built to preserve the continuity of the pressure p accross Γ. Morevover if we compute the normal component of the velocity $\mathbf{u} = -\nabla p$ over the fracture we get:

$$\mathbf{u} \cdot \mathbf{n}_\Gamma|_\Gamma = -\nabla p \cdot \mathbf{n}|_\Gamma = -\left.\frac{\partial p}{\partial y}\right|_\Gamma = -\left. k_f cos(x) sinh(y)\right|_{\{y=0\}} = 0.$$

Taking into account the continuity of pressure and the zero normal component of velocity accros Γ the accomplishment of the coupling conditions is trivial and it holds $\forall\ \xi \in [0, 1]$.

We consider three kinds of mesh: hexahedral (cartesian in particular) mesh, tetrahedral mesh and polyhedral mesh; for every type of mesh we consider different refinements to study the convergence order of the numerical method. The Table 4.1 summarizes the characteristics of the meshes in terms of number of cells, facets, fracture facets, minimum diameter, avarage diameter and maximum diameter.

	Hexa1	Hexa2	Hexa3	Hexa4	Tetra1	Tetra2	Tetra3	Tetra4
#Cells	256	2048	16384	131072	224	1211	9456	82407
#Facets	896	6656	51200	401408	564	2782	20344	170900
#Fracs	32	128	512	2048	28	94	360	1510
h_{min}	0.250	0.125	0.0625	0.03125	0.250	0.125	0.0625	0.031
h_{ave}	0.250	0.125	0.0625	0.03125	0.499	0.305	0.163	0.081
h_{max}	0.250	0.125	0.0625	0.03125	1.118	0.563	0.314	0.144

	Poly1	Poly2	Poly3	Poly4
#Cells	896	3592	8761	17459
#Facets	3036	13064	33214	67881
#Fracs	84	282	528	1080
h_{min}	0.036	0.018	0.016	0.009
h_{ave}	0.171	0.095	0.064	0.048
h_{max}	0.559	0.282	0.208	0.157

Table 4.1: Mesh characteristics.

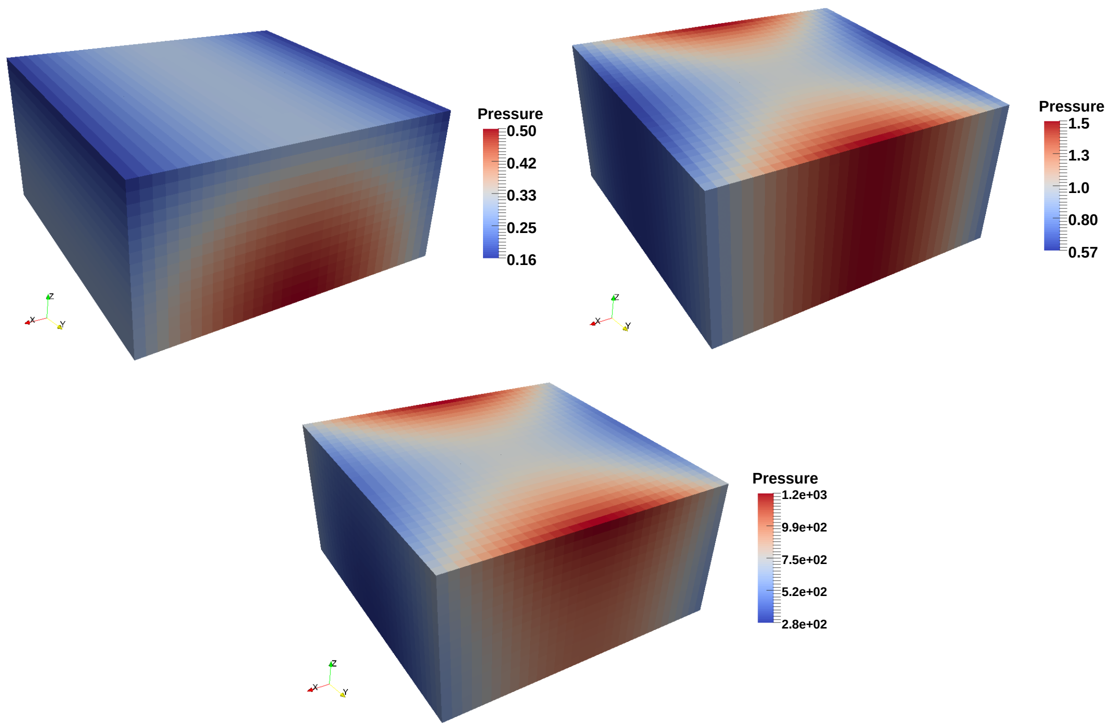

Figure 4.1: Numerical solutions on bulk. From left to right: $k_f = 10^{-3}, 1, 10^3$.

In terms of permeability we consider $k_f = 10^{-3}, 1, 10^3$; the aperture is kept fixed as $l_\Gamma = 10^{-2}$. The numerical solution over the mesh "Hexa3" (see Table 4.1) is shown in

Figure 4.1 for the three different values of the fracture permeability. Let us note that, as the analytical solution, the numerical solution for $k_f = 10^{-3}$ is quite independent from the y coordinate and for $k_f = 1$ the solution is independt from the z coordinate.

k_f	Conv. rate p_h	Conv. rate $p_{\Gamma,h}$	Conv. rate $\mathbf{u}_h$
10^{-3}	1.9860	1.9947	1.8241
1	1.9037	1.9952	1.8029
10^3	1.9544	1.9302	1.8097

Table 4.2: Mean convergence rate for the hexahedral grids.

k_f	Conv. rate p_h	Conv. rate $p_{\Gamma,h}$	Conv. rate $\mathbf{u}_h$
10^{-3}	1.9208	2.0029	1.0476
1	1.9921	1.9977	1.1500
10^3	1.7368	0.9741	1.0161

Table 4.3: Mean convergence rate for the tetrahedral grids.

Error	p_h	$p_{\Gamma,h}$	$\mathbf{u}_h$
Poly1	0.003102	0.005683	0.03510
Poly2	0.006462	0.001739	0.04672
Poly3	0.002741	0.0008369	0.04384
Poly4	0.004723	0.0009230	0.03928

Table 4.4: Relative errors for polyhedral grids in the case $k_f = 10^{-3}$.

We have given a first simple example of computation. Let us now focus on the study of the order of convergence. In [3] the order of convergence of the mimetic scheme has been widely studied and a first order convergence order has been proven for both pressure and velocity; moreover for the pressure a result of superconvergence, a quadratic order in particular, has been proven too. Also the TPFA scheme of FV would show a quadratic order of convergence, provided that the K-orthogonality condition is fullfilled, because otherwise the convergence of the scheme is not guaranteed (see [8]). So that we expect: a quadratic order of convergence for pressure in bulk and fractures and a first order for the velocity.

Regarding the order of convergence we give a brief survey of the norms used to compute the errors. For the pressure in bulk the norm associated to the mimetic space Q_h has been used (see section 2.1 in chapter 2). The formula is

$$||q_h||^2_{Q_h} = \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| q^2_{\mathsf{P}} \qquad \forall\ q_h \in Q_h. \tag{4.3}$$

In practice it is a midpoint approximation of the L^2 norm; for the fracture pressure the same norm has been used. For the velocity the norm used is the one of the mimetic space W_h. The formula is

$$||\mathbf{v}_h||^2_{W_h} = \mathbf{v}_h^{\mathsf{T}} \mathsf{M} \mathbf{v}_h \qquad \forall\ \mathbf{v}_h \in W_h, \tag{4.4}$$

where M is the mimetic inner product matrix. Given the numerical solution $(p_h, \mathbf{u}_h)$, the relative errors with respect the exact solutions $(p, \mathbf{u})$ have been computed as

$$err_p = \frac{||p^I - p_h||_{Q_h}}{||p^I||_{Q_h}}, \qquad err_u = \frac{||\mathbf{u}^I - \mathbf{u}_h||_{W_h}}{||\mathbf{u}^I||_{W_h}}, \tag{4.5}$$

where the apex I indicates proper projection operators over Q_h and W_h (see section 2.1 in chapter 2 for the details).

Note that projecting the velocity $\mathbf{u}$ requires to take into account the decoupling of the fracture facets; in our case it means adding N_Γ zero degrees of freedom at the tail of $\mathbf{u}^I$, because $\mathbf{u} \cdot \mathbf{n}|_\Gamma = 0$.

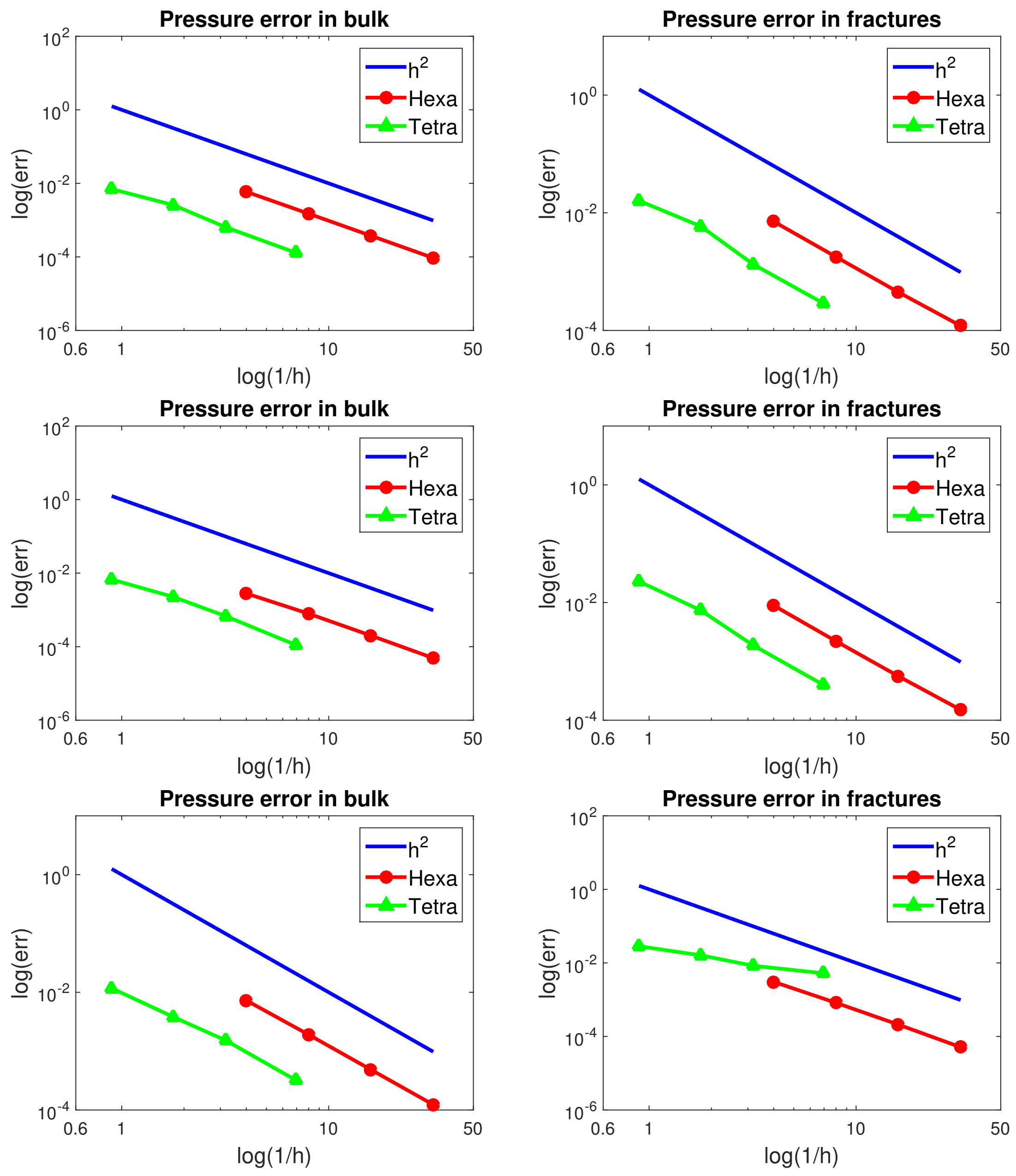

Figure 4.2: Relative errors for pressure both in bulk and fructure as a function of the meshsize (loglog scale). From top to bottom: $k_f = 10^{-3}, 1, 10^3$.

In Table 4.2 and 4.3 the mean convergence orders estimated for hexahedral and tetrahedral grids are reported for the three different values of k_f. Regarding the hexahedral case a second order convergence is clearly achieved for both the pressure in the porous medium, computed with the MFD, and the pressure in the fracture, computed with FV; so the theoretical results are verified. The estimated velocity order in the three cases is about 1.8, so a superconvergence effect is observed also for the velocity in the case of hexahedral grid. Also in the tetrahedral case the results are pretty good for the MFD, because a second order is achieved for the pressure in bulk and a first order for the velocity. For the FV in the fracture a second order convergence is estimated in the cases $k_f = 10^{-3}, 1$ but not for $k_f = 10^3$. We stress the fact that the K-orthogonality condition is quite restrictive and for an isotropic permeability tensor, as $K_\Gamma = k_f \mathsf{I}$ is in our case, this condition is surely fullfilled in a case of a cartesian grid and not for an unstructured triangular grid (indeed a tetrahedral mesh over the bulk induces a triangular mesh over the fracture). Regarding these two cases in Figure 4.2 and 4.3 the relative error as a function of the meshsize $h = \max_{\mathsf{P} \in \Omega_h} diam(\mathsf{P})$ is shown to have a graphical verification of what has been pointed out just before.

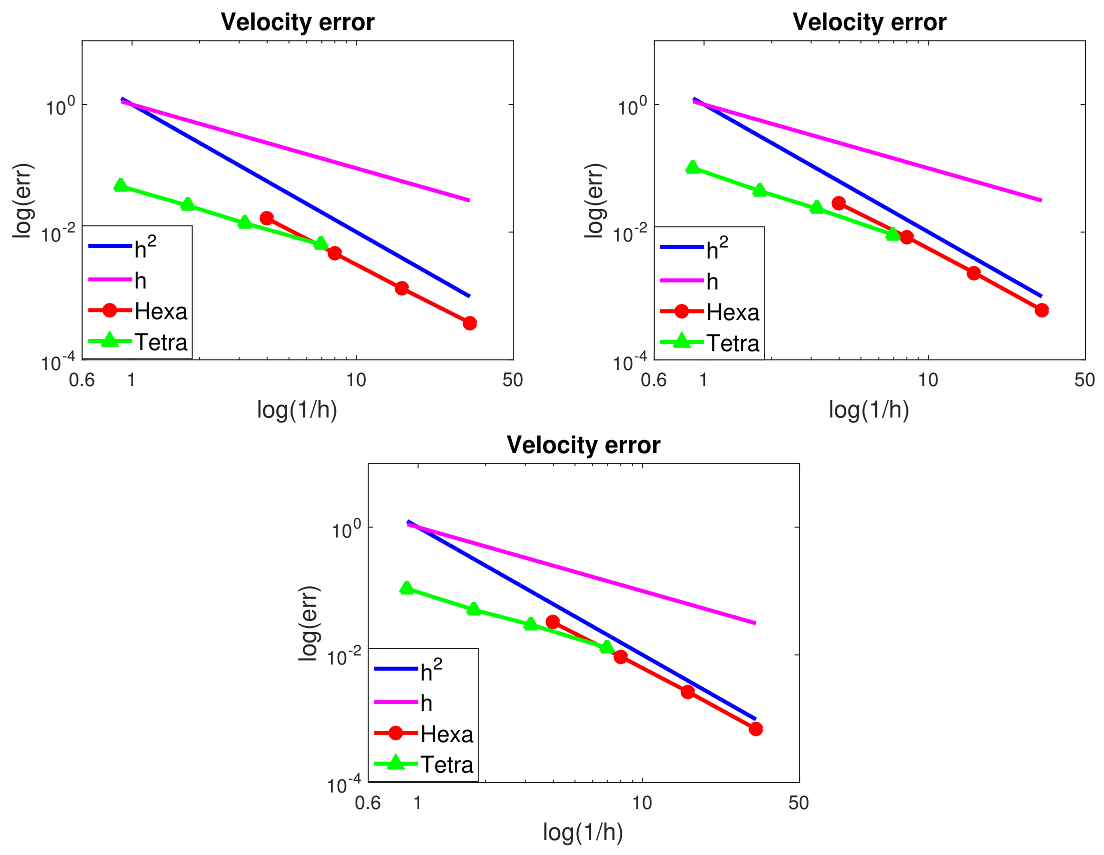

Figure 4.3: Relative errors velocity as a function of the meshsize (loglog scale). From left to right: $k_f = 10^{-3}, 1, 10^3$.

For the case of polyhedral grids the theoretical results in terms of convergence are not verified at all. We do not show the mean convergence order because the estimated convergence orders are both positive and negative and it would be misleading. In Table 4.4 the relative errors are reported in the case $k_f = 10^{-3}$ and it's clear that the method does not converge and the error is not decreasing with the meshsize; about the same behaviour has been observed for the other permeability values. For the FV scheme, as already poitend out, this is probably due to the K-orthogonality that is not fullfilled. For the MFD the matter is more complex. Indeed in principle the MFD requires only weak restrictions on the mesh, in particular the convergence estimates hold with three regularity assumptions that allow also the presence of non convex elements. However, see [3] for the details, the third regularity assumption requires star-shaped elements. The employed polyhedral grids in our case are rather general and they have been developed from tetrahedral grids by agglomeration of cells around each internal vertex. So the presence of non star-shaped elements is possible and, in any case, there are a lot of non convex and distorted elements. This is probably the reason of the non convergence of the method in that case.

4.2 The fracture network test case

Now we present a more realistic case, i.e. a case with a complex network of fractures. The bulk domain is $\Omega = (0,2) \times (0,1) \times (0,1)$ and the network of fractures Γ consists of seven fractures with several intersections; the bulk domain and the network are shown in Figure 4.4. The bulk permeability is assumed to be $\mathsf{K} = \mathsf{I}$, while in the fractures we consider $\mathsf{K}_\Gamma = k_f \mathsf{I}$. On the left and right boundary sides of the domain we consider Dirichlet conditions, in particular we fix $p = 1$ on $\{x = 0\}$ and $p = 0$ on $\{x = 2\}$, while on the top, bottom, front and back boundary sides of the domain we impose homogenous Neumann boundary conditions. We consider an aperture $d = 10^{-2}$ and a coupling parameter $\xi = 0.5$. We have employed a polyhedral mesh generated as the ones of the previous test, its characteristics are: 31745 facets, 9599 cells, 1245 fracture facets, $h_{min} = 0.007498$, $h_{ave} = 0.0605$ and $h_{max} = 0.1925$.

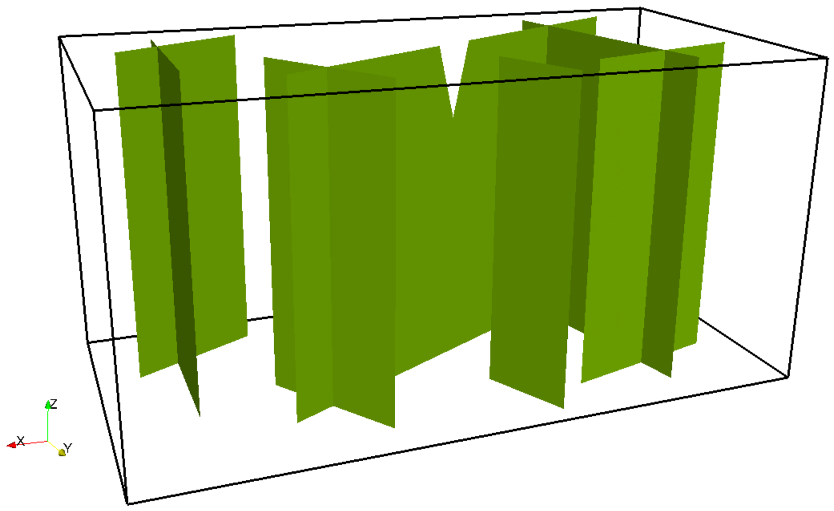

Figure 4.4: Porous medium with fracture network.

We have considered two cases of fracture permeability: $k_f = 10^{-3}, 10^3$ and we have solved the linear system with the direct methods UmfPack and the iterative method GMRES with different preconditioners, in order to show the efficiency of the GMRES with the block triangular and the ILU preconditioners implemented in the code. For the iterative solvers the maximum number of iterations has been fixed to 50000 and the tolerance to 10^{-6}; whereas the restart level for the GMRES is set to 300.

The computing times and the iterations needed to converge in the two permability cases are reported in Table 4.5 for different methods of resolution (clearly for the UmfPack that is a direct method there aren't iterations). Note that solving the linear system with the diagonal preconditioner, i.e. a trivial preconditioner, requires a huge amounts of

k_f	UmfPack	GMRES - Diag.	GMRES - BlockTr.	GMRES - ILU
10^{-3}	22.929 s -	848.823 s - 40798	1.150 s - 37	0.965 s - 29
10^{3}	19.893 s -	834.678 s - 40122	2.875 s - 38	2.433 s - 30

Table 4.5: Computing times (in seconds) to solve the linear system and iterations needed to reach convergence for different methods of resolution and for $k_f = 10^{-3}, 10^3$.

iterations and of computing times, resulting in a strong worsening in terms of performance with respect the direct solver UmfPack. This is due to the fact that the original system is much ill-conditioned and a trivial preconditioner is not enough. The block triangular and ILU preconditioners provide a very good improvement of performance (computing times), not only with respect to the diagonal preconditioner case but also with respect the UmfPack solver; this latter thing is clearly the most important to obtain a good iterative solver that can handle very large linear systems arising from 3D problems. Note that with the block triangular and the ILU preconditioners the restart of the GMRES does not happen, because they converge in few iterations; this is an important point because restarting the GMRES slows down much the convergence of the method. Note that the ILU provides slightly better performances because it converges in less iterations and less time.

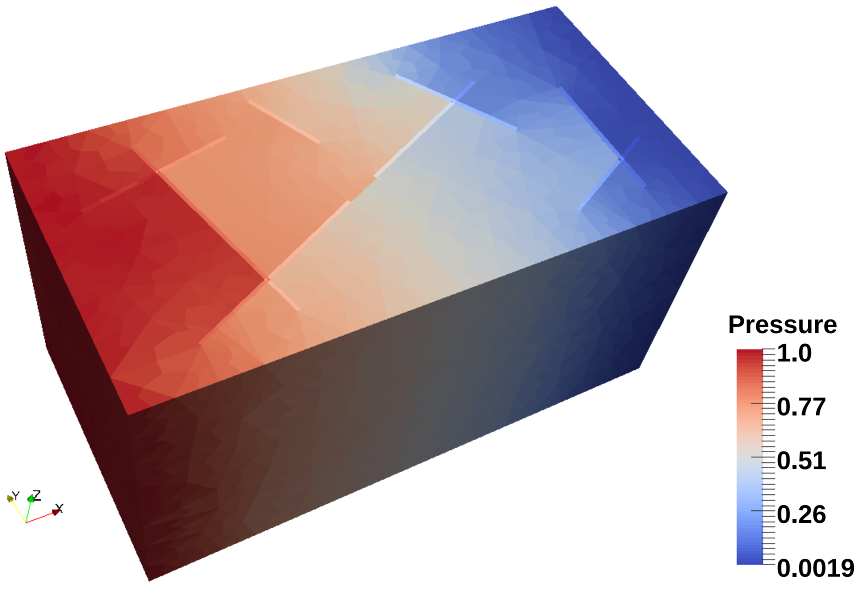

Figure 4.5: Pressure distribution in the porous medium - sealed network case.

Now we comment about the results of the computations in the two cases of permeability. The results that we show refer to computations done with the GMRES employing the ILU preconditioner. When $k_f = 10^{-3}$ the fractures are much less permeable with respect to the bulk and they are called "sealed fractures", i.e. fractures that act as barriers for the flow. If $k_f = 10^3$ the fractures are called "conductive fractures" and the network represents a preferential path for the flow.

Let us start from the case of a network of barriers. The bulk pressure is reported in Figure 4.5. Note that the action of the fractures as barriers implies a strong discontinuity of the pressure accross the fractures, which, phisically, is simply due to a mass accumulation accross the fractures. The pressure inside the fractures is reported in Figure 4.6 from which we can see that also the fracture network is characterized by pressure discontinuities. Finally the pressure inside the bulk and the network for a particular cut of the domain is reported in Figure 4.7.

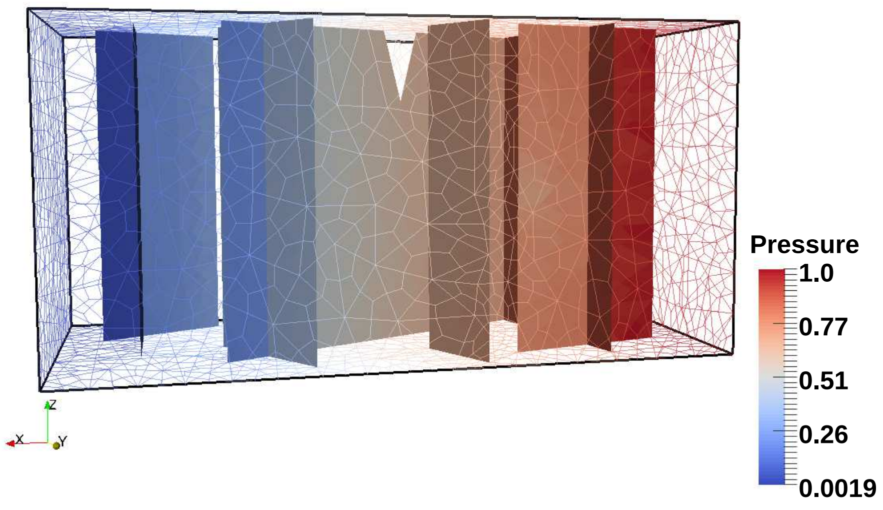

Figure 4.6: Pressure distribution in the fracture network - sealed network case.

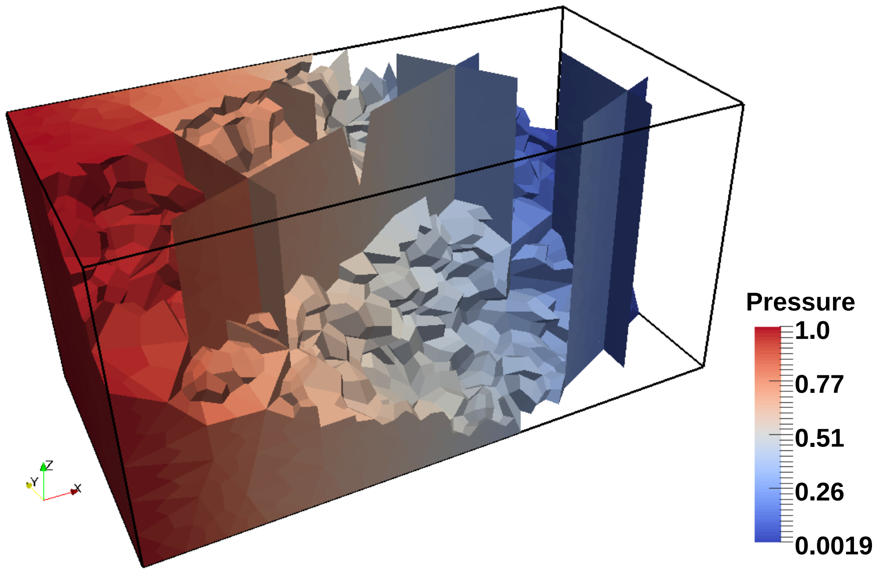

Figure 4.7: Pressure distribution inside the porous matrix and fractures on a cut of the domain - sealed network case.

The bulk pressure in the case of a network of "conductive fractures" is reported in Figure 4.8. Note that in this case the bulk pressure follows the network direction and this is due to the fact that the preferential path of the flow is the network. In Figure 4.9 the pressure in the fractures is reported; note that the pressure in each single fracture shows little gradients, on the other hand in high permeable fractures little gradients are enough to move the flow. While in Figure 4.10 the pressure in bulk and fractures for a particular cut of the domain is reported.

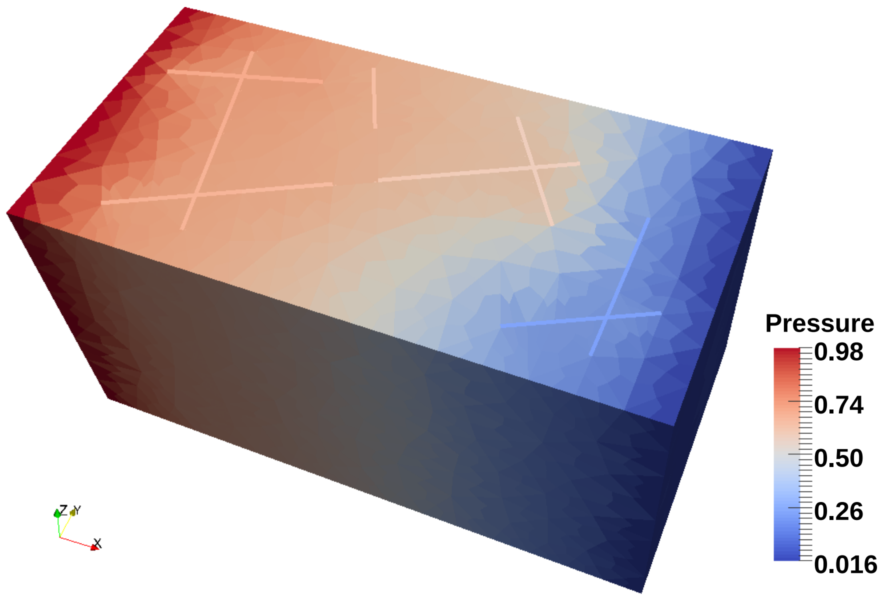

Figure 4.8: Pressure distribution in the porous medium - conductive network case.

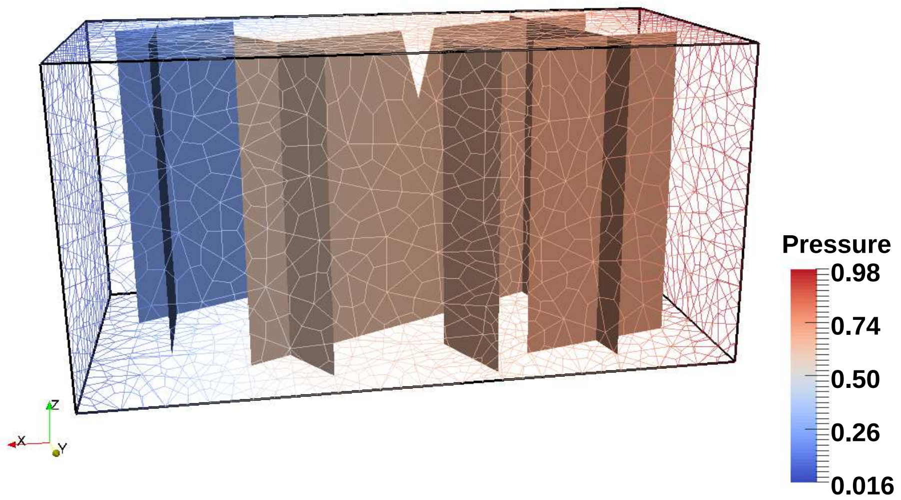

Figure 4.9: Pressure distribution in the fracture network - conductive network case.

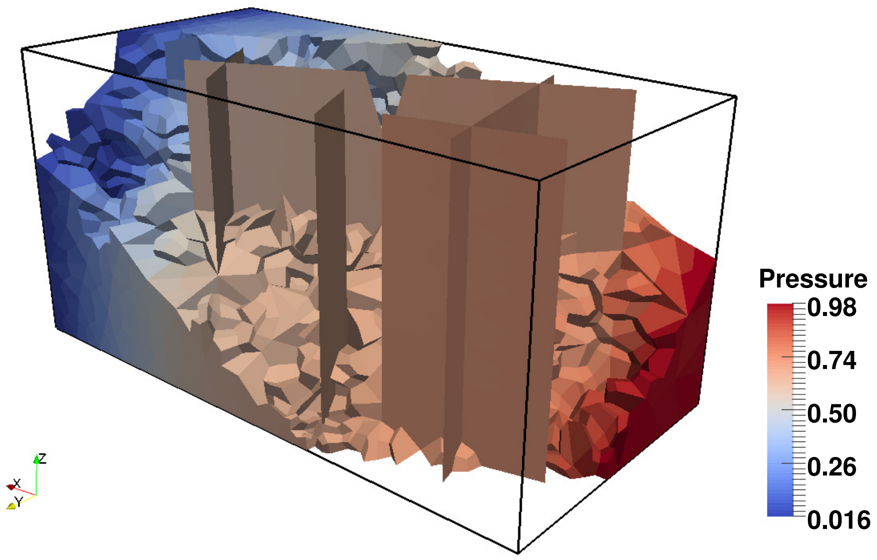

Figure 4.10: Pressure distribution inside the porous matrix and the fractures on a cut of the domain - conductive network case.

Conclusions

In this project we have developed a C++ code that implements a 3D Mimetic Finite Difference Method to solve the Darcy equation in mixed form in the bulk domain. A Finite Volume discretization, based on the two-point flux approximation, has been kept in the fractures and the coupling conditions have been implemented in the most general form. Moreover we have presented and implemented in the software two suitable preconditioners (a block triangular and an ILU preconditioner) to accelerate a Krylov-subspace method, in particular the GMRES. Using an iterative method instead of a direct one is very important in a 3D framework where computations with direct solvers become unfeasible mainly due to memory limitations. We have tested the behaviour of the method in terms of convergence order with different meshes and with different permeabilities. Finally we have assessed the numerical effort to solve a case with a complex network of fractures showing the efficiency of the two preconditioners.

Several improvements of the work have been planned for the thesis. First of all a more accurate study about the order of convergence is needed to recover the theoretical results also in the case of general polyhedral meshes. In this sense we will work on the shape and the regularity of the polyhedra. This is important because the MFD have been developed to handle generic polyhedral meshes. From a theoretical point of view a generalization to the case of network of fractures of the well-posedness results given in [2] will be probably given in the thesis. Regarding the preconditioners we will try to develop some theory in order to investigate the well numerical results in terms of computational effort. Then a numerical study on the heterogenuity of the bulk permeability tensor will be done with the goal to have the capability of the code to treat realistic cases in which the permability in the porous media can vary also of some orders of magnitude.

There are several extensions of this work. For instance, one may set the mimetic code to solve time dependent problems or extend the mimetic approximation also in the fractures to have a better approximation of the velocity in fractures. A more general study of polyhedral discretization based on virtual element methods can be carried out, this could open a possibility of implementing higher order approximations. The porous matrix can be considered compressible and the problem can be coupled with the equations of the poroelasticity.

Bibliography

[1] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, C++ in depth, Addison-Wesley, 2001.

[2] P. F. Antonietti, L. Formaggia, A. Scotti, M. Verani, N. Verzotti, *Mimetic finite difference approximation of flows in fractured porous media*, ESAIM: Mathematical Modelling and Numerical Analysis, 2015.

[3] L. Beirão da Veiga, K. Lipnikov, G. Manzini, *The mimetic Finite Difference Method for Elliptic Problems*, Springer, 2014.

[4] M. Benzi, G. H. Golub, J. Liesen, *Numerical solution of saddle point problems*, Acta Numerica, 14: pages 1-137, 2005.

[5] E. Burman, P. Zunino, *Numerical Approximation of Large Contrast Problems with the Unfitted Nitsche Method*, In J. Blowely, Max Jensen, editors, Frontiers in Numerical Analysis - Durham 2010, pages 227-282, Springer, 2012.

[6] C. D'Angelo, A. Scotti, *A mixed finite element method for Darcy flow in fractured porous media with non-matching grids*, ESAIM, 46(2): pages 465-489, 2012.

[7] J. Droniou, *Finite volume schemes for diffusion equations: Introduction to and review of modern methods*, Mathematical Models and Methods, in Applied Sciences, 24(08):1575–1619, 2014.

[8] G.T. Eigestad, R. A. Klausen *On the convergence of the multi-point flux approximation o-method: Numerical experiments for discontinuous permeability*. Numerical Methods in Partial Differential Equations, 21(6):1079–1098, 2005.

[9] V. Martin, J. Jaffré, and J. E. Roberts, *Modeling fractures and barriers as interfaces for flow in porous media*, SIAM Journal on Scientific Computing, 26(5):1667–1691, 2005.

[10] F. Sottocasa, *Formulazione mista per flussi in mezzi porosi fratturati: approssimazione con le Differenze Finite Mimetiche*, Tesi Magistrale, Politecnico di Milano, 2015.

[11] N. Verzotti, *Flusso in un mezzo poroso fratturato: approssimazione numerica tramite le Differenze Finite Mimetiche*, Tesi Magistrale, Politecnico di Milano, 2014.

[12] CGAL, http://www.cgal.org.

[13] Eigen v3, http://eigen.tuxfamily.org.

[14] GetPot: Input File and Command Line Parsing, http://getpot.sourceforge.net.

[15] IML++, http://math.nist.gov/iml++/