POLITECNICO DI MILANO

School of Industrial and Information Engineering
Master of Science in Mathematical Engineering

Preconditioning techniques for fractured porous media discretized by mimetic finite differences

Master's thesis in Computational Science and Engineering

ADVISOR:
Prof. Luca Formaggia

CO-ADVISORS:
Prof. Paola F. Antonietti
Dr. Anna Scotti

Submitted by:
Jacopo De Ponti
ID: 853013

Academic Year 2017-2018

Abstract

In the mathematical modelling for flows in porous media the treatment of fractures is of great interest for a wide range of engineering applications. For the fractures we consider a reduced problem in which they are modelled as $(d-1)$-dimensional interfaces. The governing equations for the porous matrix are discretized by mimetic finite differences, while in fractures a two-point flux approximation version of the finite volume scheme is considered. The resulting linear system can be formulated as a generalized saddle point problem, leading the way to several preconditioning techniques. In this thesis we have studied different preconditioners based only on the block structure of the matrix of the system, in order to accelerate Krylov-subspace methods. The problem of preconditioning and the selected techniques are described theoretically, and several numerical experiments are presented to study the behaviour of the preconditioners with respect to the mesh size and the model parameters. Moreover the spectral properties of the system have been analysed, in particular a theoretical estimate for the condition number of the matrix of the system has been proved. From the numerical tests we can conclude the efficiency and robustness, with respect to both the mesh size and the model parameters, of two of the selected preconditioners, and conjecture the suboptimality of the theoretical estimate of the condition number. Also a C++ software has been developed, focusing on the object-oriented architecture of the code for the mimetic scheme and the preconditioners.

Keywords: Preconditioning; Saddle point problems; Flows in fractured porous media; Mimetic finite differences; C++ programming for scientific computing.

Sommario

Nella modellazione matematica per flussi in mezzi porosi il trattamento delle fratture è di grande interesse per una vasta gamma di applicazioni in ambito ingegneristico. Per le fratture si considera un problema ridotto in cui sono modellate come interfacce $(d-1)$-dimensionali. Le equazioni che governano il flusso nella matrice porosa vengono discretizzate con le differenze finite mimetiche, mentre nelle fratture si considera una versione a due punti dello schema volumi finiti. Il sistema lineare risultante dal modello discreto può essere formulato come un problema punto sella generalizzato e ciò apre la strada a diverse tecniche di precondizionamento. In questa tesi abbiamo studiato alcuni precondizionatori basati solo sulla struttura a blocchi della matrice del sistema, al fine di accelerare metodi basati su iterazioni in spazi di Krylov. Il problema del precondizionamento e le diverse tecniche adottate vengono descritte a livello teorico, e molti esperimenti numerici vengono presentati per studiare il comportamento dei precondizionatori rispetto alla griglia e ai parametri di modello. Inoltre sono state studiate le proprietà spettrali del sistema, in particolare si presenta e si dimostra una stima teorica del numero di condizionamento della matrice del sistema. Nei test numerici due dei precondizionatori studiati si sono rivelati efficienti e robusti sia rispetto alla griglia che ai parametri di modello, mentre, riguardo al numero di condizionamento, dai test possiamo congetturare che la stima teorica sia subottimale. In aggiunta è stato sviluppato un codice in C++, concentrandosi sulla struttura orientata a oggetti per l'implementazione dello schema mimetico e dei precondizionatori.

Parole chiave: Precondizionamento; Problemi punto sella; Flussi in mezzi porosi fratturati; Differenze finite mimetiche; Programmazione in C++ per il calcolo scientifico.

Contents

Introduction **1**

1 The mathematical model **5**

1.1 Governing equations 5

1.2 Weak formulation and well-posedness 9

2 Numerical approximation of the problem **13**

2.1 State of the art 13

2.2 Mimetic finite differences approximation 15

2.2.1 The mesh and the degrees of freedom 15

2.2.2 The mimetic divergence operator 17

2.2.3 The mimetic inner products 18

2.2.4 Discretization of the bulk problem 21

2.3 Finite volume approximation for the fracture network 22

2.4 Algebraic formulation 25

2.5 Practical imposition of Neumann boundary conditions 27

3 Preconditioners for the discrete problem **29**

3.1 Krylov subspace methods and GMRES 30

3.2 A brief introduction to preconditioning 33

3.3 Preconditioning for saddle point problems 35

3.4 Preconditioning techniques 36

3.4.1 Block Triangular preconditioner 37

3.4.2 Block ILU preconditioner 39

3.4.3 Hermitian and skew-Hermitian preconditioner 40

3.5 Spectral analysis of the governing matrix 43

3.5.1 Characterization of the governing matrix 43

3.5.2 Preliminary lemmas and mesh assumption 45

3.5.3 Estimate of the condition number 48

4 The C++ code **53**
4.1 The numerical operators 54
4.2 The builders of the system 63
4.2.1 The class `global_BulkBuilder` 64
4.2.2 The class `FractureBuilder` 65
4.2.3 The class `SaddlePoint_StiffMatHandler` 67
4.3 The preconditioners 70
4.4 The solvers 73
4.4.1 The class `Solver` 74
4.4.2 The class `IterativeSolver` 75
4.4.3 The class `imlGMRES` 76
4.5 The class `Problem` 77
4.6 Input parameters 81
4.7 Tutorial 83

5 Numerical results **87**
5.1 Verification of the order of convergence 88
5.2 Testing the preconditioners 95
5.2.1 The single fracture test case 95
5.2.2 A first case of network of fractures 102
5.2.3 A second case of network of fractures 108
5.3 Test on the spectral properties of the system 109
5.3.1 Spectral properties of the transmissibility matrix 111
5.3.2 Numerical estimate of the condition number 113

Conclusions **117**

Bibliography **121**

List of Figures

1.1 Fracture configurations 6
1.2 Sketch of the mathematical model 7

2.1 Admissible elements in MFD 16
2.2 Degrees of freedom of the MFD bulk scheme 16
2.3 Geometrical quantities involved in TPFA 24
2.4 Sketch of the boundary for the ghost cell approach 25

4.1 Inheritance graph for `global_Operator` 55
4.2 Inheritance graph for `preconditioner` 71
4.3 Inheritance graph for `Solver` 73

5.1 Discrete pressure in bulk - test on the order of convergence 89
5.2 Relative errors of bulk pressure - test on the order of convergence . 92
5.3 Relative errors of fracture pressure - test on the order of convergence 93
5.4 Relative errors of velocity - test on the order of convergence 94
5.5 Number of iterations and computing times - single fracture case . . 97
5.6 Pressure in fracture by varying the coupling coefficient - single fracture case 100
5.8 Configuration of the first network of fractures 102
5.7 Pressure field in the bulk domain - single fracture case 103
5.9 Pressure distribution in the porous medium - sealed fractures - first network of fractures 105
5.10 Pressure distribution in the fractures network - sealed fractures - first network of fractures 105
5.11 Pressure distribution inside the porous matrix and fractures on a cut of the domain - sealed fractures - first network of fractures 106
5.12 Pressure distribution in the porous medium - conductive fractures - first network of fractures 106
5.13 Pressure distribution in the fractures network - conductive fractures - first network of fractures 107
5.14 Pressure distribution inside the porous matrix and the fractures on a cut of the domain - conductive fractures - first network of fractures 107
5.15 Configuration of the second network of fractures 108
5.16 Pressure distribution inside the porous matrix and the fractures on a cut of the domain - sealed fractures - second network of fractures 110
5.17 Pressure distribution inside the porous matrix and the fractures on a cut of the domain - no influence of the network - second network of fractures 110

5.18 Pressure distribution inside the porous matrix and the fractures on a cut of the domain - conductive fractures - second network of fractures 111
5.19 Condition number of the transmissibility matrix - spectral analysis of TPFA 112
5.20 Condition number of the system - spectral analysis of the system . . 114

List of Tables

5.1 Diameters of the grids - test on the order of convergence 90
5.2 Mean convergence rate - test on the order of convergence 90
5.3 Convergence details and computing times by varying the mesh size - single fracture case 96
5.4 Optimal α coefficient of HSS - single fracture case 98
5.5 Convergence details and computing times by varying the fracture parameters - single fracture case 99
5.6 Convergence details by varying the coupling coefficient - single fracture case 101
5.7 Convergence details and computing times by varying the fracture parameters - first network of fractures 104
5.8 Convergence details and computing times by varying the fracture parameters - second network of fractures 109
5.9 Spectral properties of the transmissibility matrix - spectral analysis of TPFA 112
5.10 Condition number of the system - spectral analysis of the system . . 113

Introduction

The simulation of underground flows is of great interest for a wide range of engineering applications: hydrocarbon exploitation, deep geological repositories for radioactive waste, utilisation of geothermal energy, and the study of groundwater contamination in aquifers are just a set of examples. In geophysical applications porous media are often characterized by strong heterogeneities that have a major impact on the flow. A classical situation is the presence of layers in the ground with different geometries, filled by different materials with permeability that can vary several orders of magnitude. Moreover, tectonic stresses, but also human activities, may generate fractures at very different space scales, ranging from lengths of microns for micro-fractures to hundreds of kilometres for faults. Fractures typically form a network of rock discontinuities with apertures having widths ranging over scales from microns to centimetres. They are filled by materials with a permeability that may differ of several orders of magnitude with respect to the surrounding rock. Depending on their permeability, fractures can act as a preferential path for the flow (high permeable fractures) or as barriers with a sealing effect on the flow (low permeable fractures).

In the past decades, the most common techniques for simulating flows in fractured porous media have been dual porosity models [9], so that fractures were taken into account via homogenization. However this approach has several limitations, e.g. it is inadequate to simulate cases of disconnected fractures or cases with a few large fractures. For these reasons discrete fracture models, in which fractures are explicitly represented, have been studied in last few years. Because of their small aperture, fractures are commonly modelled via $(d-1)$-dimensional interfaces immersed in a d-dimensional domain representing the surrounding rock (which is called "bulk"). From the mathematical point of view, a suitable reduced model [52] has to be adopted to describe flow along the fracture surfaces, with coupling conditions that take into account the flow exchange between bulk and fractures.

Different models can be adopted for flow in the fractures. For human-induced fractures Stokes [38] or Darcy-Forchheimer [40] models are used to better account for frictional forces or inertial effects. For flow along faults, Darcy's model [52] is usually accepted since faults are normally fields with debris. Other important modelling assumptions are single or multiphase flow, compressible versus incompressible fluid, depending on the application. Due to the methodological nature of this work, we focused on single-phase incompressible flow, with a constitutive law of Darcy's type.

Regarding the discretization of a fractured porous medium, the common approach consists of requiring the conformity of the grid to the fractures. Mesh conformity gives rise to several difficulties in the mesh generation procedure, in

particular the low-quality of the elements that may be distorted with small angles and non-convex. Special techniques have been developed to simplify a fracture network and ease the mesh generation process [53, 44], but they are far from general and the effect of the simplification on the flow is difficult to foresee. An alternative way of handling this problem is to couple the grid discretization with a numerical method that is designed for general polyhedral meshes and is robust with respect to mesh anisotropy, like the Mimetic Finite Difference method (MFD) [14]. In this thesis MFD are employed in the bulk, which is modelled in mixed form in order to compute both pressure and velocity accurately, whereas in fractures, which are modelled with a primal formulation, a Finite Volume (FV) approximation is considered, in particular a Two-Point Flux Approximation (TPFA) scheme [45]. The choice of a primal formulation in fractures is motivated by the fact that the coupling conditions do not involve fracture velocity and that TPFA allows a simplified treatment of fractures intersections. We stress that the thesis deals only with steady state problems.

This thesis fits a line of research on mimetic finite differences for fractured porous media, started in 2013 with the thesis work of N. Verzotti [61] and continued with the work [7] and the more recent developments [60, 34]. The original work of N. Verzotti solved the 2D problem with a mixed mimetic formulation in bulk and a TPFA scheme in fractures. In works [60, 34] a further development on mathematical and numerical modelling has been carried out. In particular a mixed mimetic formulation both in bulk and fractures has been widely studied, for a single immersed fracture [60] and for an immersed network of fractures [34]. In all the aforementioned works only 2D computations were presented and the linear system arising from the discretization was solved with direct methods. The specific goal of this thesis is the development of an efficient mimetic finite difference code for 3D flows in fractured porous media. This work can be viewed as a three-dimensional extension of [61, 7], because the same hybrid MFD-FV numerical scheme has been adopted.

In 3D problems however, direct computations are often unfeasible, mainly due to memory limitations, and the use of an iterative solver becomes mandatory. It is well-known that linear systems arising in numerical approximation of PDEs are usually bad-conditioned and so the construction of a proper preconditioner is the only option to obtain convergence in a reasonable amount of time. In our case the bad conditioning may be due to two main causes. First, the heterogeneity of the medium: even if the type of material in the domain was uniform, the presence of fractures induces strong variations in the effective permeability. The resulting strong variations in the coefficients of the governing PDE's induce a worsening of the condition number. The second cause is intrinsic of the discretization process: the algebraic system has a condition number that degrades with mesh spacing going to zero.

The development of suitable preconditioners for the discrete problem and their analysis, testing their performances and sensitivity with respect to the mesh size and the model parameters, is one of the main topic and the original part of the thesis. The linear system is sparse and in block form and can be formulated as a saddle point problem. All the preconditioning techniques proposed in this thesis take advantage of the saddle point block partition of the matrix of the linear

system. We present three preconditioners: a block triangular preconditioner, an inexact block LU (ILU) preconditioner and an Hermitian/skew-Hermitian splitting preconditioner. No work has been carried out on the mesh generation aspects, for which we have referred to [62].

The goal of preconditioning is to transform the linear system in another linear system with more favourable spectral properties, which allow a faster convergence of the iterative method. In this sense the condition number of the matrix of the system plays a fundamental role. Therefore an estimate of the condition number has been proved and this is another original contribute of this thesis. The proof is a generalization of the one presented in [51] for a diffusion problem in mixed form.

We stress that the thesis has also an important part of implementation in C++. The programming work starts from an existing code implementing a full FV numerical scheme both in bulk and fractures, with a first simple implementation of MFD in bulk in primal form. We have implemented the MFD in mixed form in bulk with more general coupling conditions. Actually most of the work in this sense has been carried out in order to have a mimetic code in a C++ object-oriented framework. The other important programming effort consists of the implementation through a well-designed class structure of the iterative solvers and of the preconditioners. The code makes a wide use of C++11 programming techniques and it employs several libraries, like the `Eigen` for an efficient storage of the sparse matrix of the system and the `IML++` for flexible implementations of preconditioned iterative solvers.

The thesis thus covers theoretical and implementative aspects, supported by extensive numerical tests. Its structure is the following.

Chapter 1. It is devoted to the presentation of the governing equations of the problem, i.e. the bulk equations, the reduced model for the fractures and the coupling conditions. Moreover a weak formulation of the bulk model and a result of well-posedness of the problem are presented, following [7].

Chapter 2. First, a state of the art to frame the thesis work within the current literature on numerical methods for fractured porous media is presented. Then the theoretical background of MFD is described and the discrete model of the bulk problem is presented. The TPFA scheme is then introduced for the discretization of the fracture problem. Finally, a result for the well-posedness of the linear system arising from the discretization is presented, following [7].

Chapter 3. First, a general introduction to Krylov subspace methods and preconditioning is presented. Then, after the description of several issues for preconditioning saddle point problems, three preconditioning techniques are described. Finally, we state some preliminary results and then we present and prove an estimate of the condition number of the matrix of the system.

Chapter 4. It is devoted to the implementative details of the C++ code. In the first part of the chapter the classes that handle the assembly of the linear system are described, while in the second part the class structure of iterative solvers and preconditioners is presented. Finally, a tutorial illustrates a practical use of the code.

Chapter 5. This chapter is devoted to numerical results: the validation of the convergence order of the discrete model; the study of the preconditioners, in terms of performance and sensitivity with respect to the mesh size and the fracture model parameters, both for a single fracture and a network of fractures. We also present

a validation of the theoretical estimate of the condition number.

Chapter 1

The mathematical model

In this chapter we present the mathematical model that governs the problem. The three ingredients for a model of an incompressible flow in a fractured porous medium are: the equations that govern the flow in the porous matrix, the equations for the fracture flow and suitable conditions that couple the two flows along the interfaces between the bulk and the fractures. Then the weak formulation of the bulk problem is presented, since our MFD discretization is carried out on the bulk model in weak form. Finally, we recall some results on the well-posedness of the model.

1.1 Governing equations

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

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

where $\mathbf{u}$ is the velocity of the flow, p is the pressure, K is a (symmetric positive and definite) permeability tensor and f is the source/sink term. We have assumed that the fluid and the medium are incompressible and we have also neglected the gravitational effects. We do not enter into the details and the physics of the Darcy law, regarding this see [11]. We keep the model in mixed form because we want to approximate jointly both pressure and velocity. The model is complemented by boundary conditions which will be detailed later.

The treatment of the fractures is more delicate, because they have a much smaller thickness with respect to the other dimensions and the dimension of the reservoir Ω. Moreover, the fractures are typically filled with a porous material whose permeability can differ greatly with respect to that of the bulk. The use of model (1.1) also in the fractures would require a very refined grid in the zones of fracture with a strong increase of the computational cost. So the thickness of the fractures is here considered negligible and a reduced model is used in the fractures. The model has been derived in [2] in the case of high permeable fractures; it has been generalized to include also the case of low permeable fractures, i.e. sealed

Figure 1.1: From left to right: an immersed fracture, a partially immersed fracture and a fracture that completely cuts the porous medium. Taken by [60].

fractures, in [52], to which we refer for the derivation and the details. We also mention that the model has been recently extended to describe transport [36] and two-phase flows [37, 43] in porous media.

We assume a single fracture that is modelled via a $(d-1)$-dimensional manifold $\Gamma \subset \mathbb{R}^{d-1}$. There are three possible fracture configurations: a fracture that cuts the whole domain in two disjoint sets, a partially immersed fracture and a completely immersed fracture; see Figure 1.1. The model is presented with the only hypothesis of a single fracture without specifying its configuration.

The fracture Γ splits the domain in two Lipschitz subdomains Ω^+ and Ω^- such that $\overline{\Omega} = \overline{\Omega^+} \cup \overline{\Omega^-}$. For the partially or fully immersed fracture, we assume that Γ can be suitably extended to obtain this partition. By $\mathbf{n}_\Gamma$ we indicate the unit normal vector to Γ, whose orientation must be considered fixed from Ω^+ to Ω^-, and with $\boldsymbol{\tau}_\Gamma$ the matrix whose columns form an orthonormal basis for the tangent space at each $\mathbf{x} \in \Gamma$.

Let us define the bulk domain as $\Omega_\Gamma = \Omega \setminus \overline{\Gamma}$. We consider the partition of the boundary of the domain $\partial\Omega = \Gamma_D \cap \Gamma_N$ such that $\Gamma_D \cap \Gamma_N = \emptyset$ and $\Gamma_D \neq \emptyset$. By $\mathbf{n}$ we denote the boundary unit outward normal. We prescribe a function g on the Dirichlet boundary Γ_D and a function ϕ on the Neumann boundary Γ_N. The model for the bulk is then

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

In Figure 1.2 a sketch of the mathematical model is reported.

Regarding the fracture, we denote the aperture (i.e. the thickness) by $l_\Gamma = l_\Gamma(\mathbf{x}) \geq l_{\min} > 0$ and we assume for the fracture flow a permeability tensor with the block diagonal structure

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

where $k_\Gamma^n > 0$ and $\mathsf{K}_\Gamma^\tau \in \mathbb{R}^{2\times 2}$ is a symmetric and positive definite tensor. Requiring a fracture permeability tensor of this type means supposing that the normal flow across the fracture does not depend on the pressure gradients along the tangent space.

Now let us define the jump and average of a function v in Ω_Γ (that must be

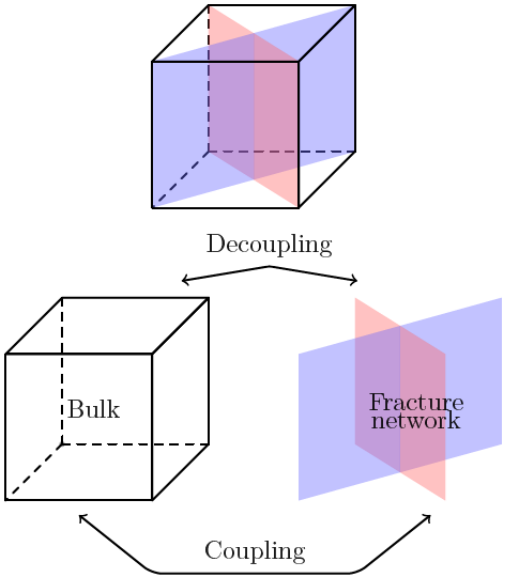


Figure 1.2: Sketch of the mathematical model. Taken by [7].

regular enough) across Γ:

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

where v^i is the trace of v on Γ taken within Ω^i, with $i = \pm$.

The pressure and the velocity in the fracture are indicated with p_Γ and $\mathbf{u}_\Gamma$ and are defined by integrating the real pressure p_f and tangential velocity $\mathbf{u}_{f,\tau}$ along the thickness of the fracture, i.e.

$$p_\Gamma = \frac{1}{l_\Gamma}\int_{-l_\Gamma/2}^{l_\Gamma/2} p_f \, dn, \qquad \mathbf{u}_\Gamma = \int_{-l_\Gamma/2}^{l_\Gamma/2} \mathbf{u}_{f,\tau} \, dn.$$

The governing equations of the fracture flow are the mass balance and the Darcy law written in tangential components; see [52] for the details. Both of them must be properly integrated across the thickness of the fracture and, after some calculations, they can be written as:

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

where $k_\Gamma = l_\Gamma \mathsf{K}_\Gamma^\tau$ and f_Γ is the source/sink term in the fracture. The differential operators have to be intended acting in the tangential direction of the fracture Γ, in particular $\nabla_\tau \cdot \mathbf{v} = \nabla \cdot \mathbf{v} - \nabla(\mathbf{v} \cdot \mathbf{n}_\Gamma) \cdot \mathbf{n}_\Gamma$ and $\nabla_\tau q = \nabla q - (\nabla q \cdot \mathbf{n}_\Gamma)\mathbf{n}_\Gamma$. Note the presence of the aperture deriving from the integration along the thickness and the velocity jump, which represents the flow exchange between the bulk and the fracture.

Let us set $\partial\Gamma_N = \Gamma \cap \Gamma_N$, $\partial\Gamma_D = \Gamma \cap \Gamma_D$ and $\partial\Gamma_{N_0} = \partial\Gamma \setminus (\partial\Gamma_N \cup \partial\Gamma_D)$. Eliminating the velocity variable to have a primal formulation, we obtain the model

for the fracture:

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

Note that on the internal boundary of the fracture an homogeneous Neumann condition is imposed, because we assume that the flow exchange between the bulk and the fracture takes place 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 [52], let $\xi \in [0, 1]$, the coupling conditions are given by

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

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 possibilities of closing the system. If $\xi = 0.5$ we are simply assuming that the pressure in the fracture is the algebraic average of the bulk pressures across the two sides of the fracture; if $\xi = 1$ the coupling conditions are equivalent to a first order approximation of Darcy law across each half of the fracture; if $\xi = 0.75$ the coupling conditions are equivalent to a second order approximation of Darcy law across each half of the fracture, indeed $\xi = 0.75$ is the optimal value.

We stress the fact that the bulk problem is in mixed form, suitable for the mimetic discretization, and the fracture model is in primal form, because it is discretized with a primal FV formulation. The choice of a primal form in the fracture is due to the fact that the coupling conditions do not depend on the fracture velocity. Moreover using FV in fractures allows a easier treatment of fractures intersections in the code implementation. Obviously there are other approaches, e.g. in [4] a primal formulation is adopted both in bulk and fracture and in [52, 60, 34] a mixed formulation both in bulk and fracture is employed.

Let us point out that in the case of a network of intersecting fractures the model is exactly the same of the one presented above, but we have to require in addition the continuity of pressure and velocity across the intersections. In such a case the notation becomes more complex, because we have to handle the intersections between fractures. Although the software developed in this thesis handles complex networks of fractures and in Chapter 5, devoted to the numerical results, we present some cases of networks, for the model we have assumed here only one fracture to ease the presentation. For a sketch of the problem with a network of fractures see [7, 60] and for a detailed analysis see [34].

Remark 1.1. *The model for the fracture depends actually on two physical parameters: $l_\Gamma k_\Gamma^\tau$ and k_Γ^n / l_Γ, which can be considered equivalent permeabilities along the tangential and normal direction. The coefficient $l_\Gamma k_\Gamma^\tau$ is related to the velocity jump across the fracture and describes the flow exchanges within the fracture. The*

dependency on the aperture is an interesting point, because physically the aperture is important as well as the permeability in determining the behaviour of the fracture flow. Think to a fracture filled by permeable grains but characterized by a very small aperture, i.e. a fracture that is a "film" in the porous medium: in spite of its high permeability probably there was no significant flow along the fracture; the factor $l_\Gamma k_\Gamma^\tau$ takes into account just this effect. If $l_\Gamma k_\Gamma^\tau \sim K$ (here K is the bulk permeability), then we can expect a conductive fracture that is a preferential path for the flow, so the bulk flow interact with the fracture and we can expect a non zero velocity jump.

The other important coefficient is k_Γ^n / l_Γ. It is a factor that indicates how much the fracture represents an obstacle for the flow and it is related to the flow exchanges across the fracture. Also for k_Γ^n / l_Γ the aperture, in addition to the permeability, plays an important role. If $k_\Gamma^n / l_\Gamma \sim K$, then we can expect that the flow avoids the fracture with a resulting non zero pressure jump.

1.2 Weak formulation and well-posedness

In this section we present the weak formulation of the bulk problem (1.2) and then we give some results on the well-posedness of the whole model. Taking into account that the mimetic discretization is carried out on the weak formulation of the bulk model, in which we introduce the coupling conditions (1.6), we start presenting it leaving the fracture model in strong form. This is because the fracture equation will be discretized with FV, so that for the moment we consider the pressure in the fracture p_Γ as given.

First, we introduce some regularity assumptions on data. In particular we suppose the ellipticity of the permeability tensor that means

$$0 < \mathsf{k}_* \leq \zeta^\mathsf{T} \mathsf{K} \zeta \leq \mathsf{k}^* \qquad \forall\, \zeta \in \mathbb{R}^d \setminus \{\mathbf{0}\} \;\; a.e.\; \mathbf{x} \in \overline{\Omega}.$$

For the forcing term and the boundary data we suppose $f \in L^2(\Omega_\Gamma)$, $g \in L^2(\Gamma_D)$ and ϕ enough regular such that there exists a lifting $R_\phi \in H(\mathrm{div}, \Omega_\Gamma)$ whose trace on Γ_N coincides with ϕ; see [59] for the definitions of these spaces.

Now let us introduce the following functional spaces:

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

with the associated norms

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

Note that we are requesting more regularity on the velocity $\mathbf{v}$ than $H(\mathrm{div}, \Omega_\Gamma)$ in order to accommodate the coupling conditions (1.6) in the weak formulation. We mention that, to have $\mathbf{v}^+$ and $\mathbf{v}^-$ well-defined, in a case of a fracture that cuts the whole domain, the standard trace operator can be used, but, in a case of an immersed fracture, suitable trace operators must be defined; regarding this topic see [4]. It can be shown that the spaces Q and W are Hilbert spaces with scalar

products inducing the above stated norms. To deal with the Neumann boundary conditions we define also the spaces:

$$W_{\phi,\Gamma_N} = \{\mathbf{v} \in W : \mathbf{v}\cdot\mathbf{n} = \phi \text{ on } \Gamma_N\}$$
$$W_{0,\Gamma_N} = \{\mathbf{v} \in W : \mathbf{v}\cdot\mathbf{n} = 0 \text{ on } \Gamma_N\}.$$

Note that W_{ϕ,Γ_N} is an affine space, i.e. $W_{\phi,\Gamma_N} = R_\phi + W_{0,\Gamma_N}$.

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 [60]. It reads as follows: find $(\mathbf{u},p) \in W_{\phi,\Gamma_N}\times Q$ such that

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

Note the essential imposition of the Neumann conditions in the velocity space and the natural imposition of Dirichlet conditions on pressure that appear directly in the formulation.

To summarize: the mathematical model, which will be discretized by the hybrid MFD-FV scheme, is represented by the model in weak form of the bulk flow, which takes into account also the coupling conditions, and by the model for the fracture in strong form. The final coupled model is given by

$$\begin{cases} a_\xi(\mathbf{u},\mathbf{v}) - B(\mathbf{v},p) + \displaystyle\int_\Gamma [\![\mathbf{v}\cdot\mathbf{n}_\Gamma]\!]p_\Gamma\ dS = -\int_{\Gamma_D} g(\mathbf{v}\cdot\mathbf{n})\ dS & \forall\ \mathbf{v}\in W_{0,\Gamma_N} \\ B(\mathbf{u},q) = \displaystyle\int_{\Omega_\Gamma} fq\ dV & \forall\ q\in Q \end{cases}$$
$$\begin{cases} -\nabla_\Gamma\cdot(k_\Gamma\nabla_\Gamma p_\Gamma) = l_\Gamma f_\Gamma + [\![\mathbf{u}\cdot\mathbf{n}_\Gamma]\!] & \text{in } \Gamma \\ p_\Gamma = g_\Gamma & \text{on } \partial\Gamma_D \\ -k_\Gamma\nabla_\Gamma p_\Gamma\cdot\boldsymbol{\tau}_\Gamma = \phi_\Gamma & \text{on } \partial\Gamma_N \\ -k_\Gamma\nabla_\Gamma p_\Gamma\cdot\boldsymbol{\tau}_\Gamma = 0 & \text{on } \partial\Gamma_{N_0}. \end{cases}$$

Now we focus on the well-posedness of the model. The fracture configuration is important for the well-posedness of the problem, because, depending on the configuration, the procedure and the complexity of the proof are different. The case of a fracture that cuts the whole domain in two disjoint sets has been treated in [52], for a mixed scheme in bulk and fractures, and in [7], for a mixed-primal model. The case of an immersed fracture has been treated in [4], for a primal scheme both in bulk and fracture, and in [60], for a mixed model both in bulk and fracture. The case of a network of fractures has been treated for a mixed-mixed model in [34].

We sketch out the main ideas and report the main results of [7], without entering into the details.

Following [7], we suppose a single fracture that cuts the whole domain in two disjoint sets and we assume $\partial\Gamma_D \neq \emptyset$, i.e. that the fracture touches the Dirichlet boundary of the domain. The first thing consists of converting in weak form also the fracture equation. We assume for the fracture permeability tensor K_Γ the ellipticity already introduced and, regarding the other fracture data, we consider $f_\Gamma \in L^2(\Gamma)$ and $g_\Gamma \in H^{1/2}(\partial\Gamma_D)$ (this ensures the existence of a suitable lifting). The Neumann data in fracture is supposed to be zero, i.e. $\phi = 0$, $\phi_\Gamma = 0$. Now define the functional space $V_{0,\partial\Gamma_D}$ and its associated norm:

$$V_{0,\partial\Gamma_D} = \{v \in H^1(\Gamma) : v = 0 \text{ on } \partial\Gamma_D\}, \qquad ||v||_{V_{0,\partial\Gamma_D}} = ||v||_{H^1}.$$

The model becomes: find $(\mathbf{u}, p, p_\Gamma) \in W_{0,\Gamma_N} \times Q \times V_{0,\partial\Gamma_D}$ such that

$$\begin{cases} a_\xi(\mathbf{u},\mathbf{v}) - B(\mathbf{v},p) + \displaystyle\int_\Gamma [\![\mathbf{v}\cdot\mathbf{n}_\Gamma]\!] p_\Gamma \, dS = -\int_{\Gamma_D} g(\mathbf{v}\cdot\mathbf{n}) \, dS & \forall\ \mathbf{v} \in W_{0,\Gamma_N} \\ B(\mathbf{u},q) = \displaystyle\int_{\Omega_\Gamma} fq \, dV & \forall\ q \in Q \\ a_\Gamma(p_\Gamma,\varphi) - \displaystyle\int_\Gamma [\![\mathbf{u}\cdot\mathbf{n}_\Gamma]\!]\varphi \, dS = \int_\Gamma l_\Gamma f_\Gamma \varphi \, dS & \forall\ \varphi \in V_{0,\partial\Gamma_D}, \end{cases} \tag{1.8}$$

where

$$a_\Gamma(\omega,\varphi) = \int_\Gamma \mathsf{K}_\Gamma \nabla_\tau \omega \cdot \nabla_\tau \varphi.$$

The procedure to prove the well-posedness of the model (1.8) starts by noting that $\forall\ \psi \in L^2(\Gamma)$ $\exists!\ q_\Gamma \in V_{0,\partial\Gamma_D}$ solution of

$$a_\Gamma(q_\Gamma,\varphi) = \int_\Gamma \psi\varphi \, dS + \int_\Gamma l_\Gamma f_\Gamma \varphi \, dS \qquad \forall\ \varphi \in V_{0,\partial\Gamma_D}.$$

This is a standard result of elliptic PDEs; see Lax-Milgram theorem in [59]. The important point is that q_Γ can be always decomposed as $q_\Gamma = q_\Gamma^0 + q_\Gamma^1(\psi)$, where

$$\begin{aligned} a_\Gamma(q_\Gamma^0,\varphi) &= \int_\Gamma l_\Gamma f_\Gamma \varphi \, dS \qquad \forall\ \varphi \in V_{0,\partial\Gamma_D} \\ a_\Gamma(q_\Gamma^1(\psi),\varphi) &= \int_\Gamma \psi\varphi \, dS \qquad \forall\ \varphi \in V_{0,\partial\Gamma_D} \end{aligned}$$

and we have

$$\begin{aligned} ||q_\Gamma^0||_{H^1(\Gamma)} &\leq C_1 ||l_\Gamma f_\Gamma||_{L^2(\Gamma)} \\ ||q_\Gamma^1(\psi)||_{H^1(\Gamma)} &\leq C_2 ||\psi||_{L^2(\Gamma)}. \end{aligned}$$

From this latter relation it follows that $\psi \to q_\Gamma^1(\psi)$ is a linear and continuous operator.

Let us define the bilinear form $A_\xi(\cdot,\cdot) : W \times W \to \mathbb{R}$ as

$$A_\xi(\mathbf{u},\mathbf{v}) = a_\xi(\mathbf{u},\mathbf{v}) + \int_\Gamma q_\Gamma^1([\![\mathbf{u}\cdot\mathbf{n}_\Gamma]\!])[\![\mathbf{v}\cdot\mathbf{n}_\Gamma]\!] \, dS.$$

We reformulate problem (1.8) as: find $(\mathbf{u}, p) \in W_{0,\Gamma_N} \times Q$ such that

$$\begin{cases} A_\xi(\mathbf{u}, \mathbf{v}) - B(\mathbf{v}, p) = F_\xi(\mathbf{v}) & \forall\ \mathbf{v} \in\ W_{0,\Gamma_N} \\ B(\mathbf{u}, q) = G(q) & \forall\ q \in\ Q, \end{cases} \tag{1.9}$$

where

$$F_\xi(\mathbf{v}) = \int_{\Gamma_D} g[\![\mathbf{v} \cdot \mathbf{n}_\Gamma]\!]\ dS - \int_\Gamma q_\Gamma^0 [\![\mathbf{v} \cdot \mathbf{n}_\Gamma]\!]\ dS$$
$$G(q) = \int_{\Omega_\Gamma} fq\ dS.$$

Note that (1.9) is the weak formulation of a classical saddle point problem. Now we can make use of a standard result of saddle point problems, see e.g. [55], to prove the well-posedness with the continuity and the coercivity of $A_\xi(\cdot,\cdot)$, the continuity of $B(\cdot,\cdot)$, the inf-sup condition and the continuity of the operators $F_\xi(\cdot)$ and $G(\cdot)$. We have the following results.

Proposition 1.1. *The $A_\xi : W \times W \to \mathbb{R}$ form is bilinear and continuous on $W \times W$. Moreover, it is coercive on*

$$\widetilde{W} = \{\mathbf{v} \in W_{0,\Gamma_N} :\ B(\mathbf{v}, q) = 0 \quad \forall\ q \in\ Q\},$$

provided that $\xi \in (1/2, 1]$.

Proposition 1.2. *The $B(\cdot,\cdot) : W_{0,\Gamma_N} \times Q \to \mathbb{R}$ form is bilinear and continuous. Moreover, it satisfies the inf-sup condition, i.e., there exists a constant $C_B > 0$ such that*

$$\inf_{q \in Q} \sup_{\mathbf{w} \in W} \frac{B(\mathbf{w}, q)}{||q||_Q ||\mathbf{w}||_W} > C_B.$$

For the proof of these results we refer to [7] and [34]. For the coercivity and the continuity of $A_\xi(\cdot,\cdot)$ we only point out that the standard norm of W previously introduced is equivalent to the norm

$$|||\mathbf{v}|||_W^2 = ||\mathbf{v}||_{L^2(\Omega_\Gamma)}^2 + ||\nabla \cdot \mathbf{v}||_{L^2(\Omega_\Gamma)}^2 + |||[\![\mathbf{v} \cdot \mathbf{n}_\Gamma]\!]|||_{L^2(\Gamma)}^2 + ||\{\mathbf{v} \cdot \mathbf{n}_\Gamma\}||_{L^2(\Gamma)}^2.$$

Whereas, for the inf-sup condition, we point out that the proof is based on the adjoint equation and on the elliptic regularity [55], provided that Ω is enough regular, like a domain of class C^2 or a convex polygon.

Finally, the regularity of the data ensures the continuity of the operators $F_\xi(\cdot)$ and $G(\cdot)$. So we conclude the well-posedness of the problem.

Chapter 2

Numerical approximation of the problem

In this chapter we describe the numerical discretization of the mathematical model introduced in Chapter 1. First we present a state of the art to describe the main current literature concerning the numerical approximation of flows in fractured porous media, and to explain how this thesis fits this particular branch of scientific computing literature. Then we describe the MFD method for the discretization of the bulk problem entering into the details of the mimetic theory. We devote a section to present the TPFA version of the finite volume scheme used to discretize the fracture model. Then we present the resulting linear system and its reduction to the classical algebraic form of a saddle point problem. We also describe a result that guarantees a unique solution of the linear system, without entering into the details of the proof. Finally we make some considerations on the imposition of the Neumann boundary conditions implemented in the code.

2.1 State of the art

As pointed out in Chapter 1, fractures are considered as $(d-1)$-dimensional interfaces, since the thickness is considered negligible, and a reduced model is adopted to simulate the fracture flow. The traditional approach for the discretization requires the conformity of the mesh, i.e. the cells must have facets (or edges in 2D) that are aligned with the fractures. This conformity gives rise to several difficulties with standard methods, because in realistic cases, where a lot of intersecting fractures may occur, the mesh generation has to handle small intersection angles, nearly coincident fractures, high disparity of traces lengths in the network of fractures (the traces are the intersections). The main problems are the very high number of elements, which may be too much for the required level of accuracy, and the low-quality of the elements (small angles, high aspect ratios, non-convex cells), and it is well-known that for most numerical methods the quality of the grid is fundamental to ensure the accuracy of the solution.

Several ways to handle these difficulties have been proposed in literature in the past years. In [48] the Boundary Element Method (BEM) is applied to a 3D network of fractures to reduce the total number of unknowns minimizing the

core memory. A possible approach to overcome the aforementioned difficulties is working on the mesh generation. A technique consists of performing suitable simplifications on the network, e.g. in [53] an hybrid Gabriel Delaunay triangulation is proposed, while in [44] large scale fractures and small scale ones are treated in different ways. Whereas, in [49] an adaptive mesh refinement method is described for the generation of a high resolution mesh that preserves the good quality of the elements. An alternative is to avoid requiring the conformity and to use a fairly regular and coarse grid, which is cut arbitrarily by the fracture, and to couple the discretization with an appropriate numerical method like the eXtended Finite Element Method (XFEM), see [28, 37]. Otherwise a possible choice is keeping the conformity requirement, but employing a numerical method that is robust, in terms of accuracy and convergence, with respect to the mesh anisotropy and distortion, like the MFD [14] or their last generalization represented by the Virtual Element Method (VEM) [12]. These method are designed for generic polyhedral/polygonal grids and require weak regularity assumptions on the elements of the mesh that could also have non-convex cells.

The usage of the MFD has greatly increased in the last years, also because, in addition to their flexibility with respect to the mesh, they have the capability to preserve several properties of the underlying physical-mathematical problem, like mass conservation and maximum principle. We mention the application of the MFD in diffusion-type problems [24, 13], electromagnetism [23, 25], non-linear and control problems [5, 6]. We stress that recently MFD and VEM have been employed in fractured porous media, for the MFD see [1, 7, 34] and for VEM see [15, 16]. Our approach, which consists of discretizing the bulk domain with the MFD in mixed form and the fractures with the TPFA version of the finite volume scheme (so primal form for the fracture model), is the 3D extension of the work developed in [7]; whereas in [1, 34] a full mimetic mixed-mixed formulation is used. We stress that in [15, 16] the bulk is considered impervious and only the pressure in fractures is reconstructed; this allows to treat more complex networks easily but it neglects the interaction with the surrounding rock. Finally we mention a novel approach based on a PDE-constrained optimization algorithm [21, 22] that, in the same framework of [15, 16], allows to accommodate different meshing procedures for each fracture and allows to decouple the global problem in several local problems, in order to have a great potential for parallel implementations.

We remark that using a mixed formulation in bulk and a primal scheme in fractures is different from the classical approach available in literature, which consists of using the same type of formulation both in bulk and fractures. Our choice is motivated by the fact that the coupling conditions at the interfaces do not involve the fracture velocity. Moreover we stress that discretizing the fractures model with TPFA allows some important simplifications in the treatment of fracture intersections. Indeed a connectivity transformation, called "star-delta" and introduced in [45], is employed to handle intersections: this simplification is used during the transmissibility evaluation step and it is very easy to be introduced in the flow simulator. A mixed formulation in fractures would require to enforce pressure and flux continuity in the discrete formulation, with a resulting more complex treatment of intersections, see [34]. As already pointed out, this thesis is a 3D generalization of [7] and the key point is the development of suitable

preconditioners to make convenient the usage of an iterative method with respect to a direct one, which is an important task in 3D, where direct computations become unfeasible mainly because of memory limitations.

2.2 Mimetic finite differences 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 discrete spaces, then we introduce the projection operators and the mimetic divergence operator, and finally describe the construction of the mimetic inner products. For a comprehensive overview on the MFD method we refer to [14].

2.2.1 The mesh and the degrees of freedom

As in Chapter 1 we consider only one fracture. Let Ω_h be a partition of Ω into non-overlapping polyhedra P that are aligned with the fracture Γ. We define 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 $\Omega_h = \Omega_h^+ \cup \Omega_h^-$ (if Γ does not cut the whole domain it must be extended to have this partition). Moreover the mesh Ω_h induces a natural partition of Γ that we indicate with Γ_h, which is the fracture bi-dimensional grid of polygons.

We number the fracture facets $\hat{\mathsf{f}}_i \in \Gamma_h$, $i = 1, ..., N_\Gamma$, where N_Γ is the number of the fracture facets. In order to deal with the flux discontinuity across the fracture, in the bulk problem every fracture facet $\hat{\mathsf{f}}$ is replaced by two distinct facets f^+, f^- geometrically identical to $\hat{\mathsf{f}}$, which will be associated to Ω_h^+ and Ω_h^-, respectively, so that each subset Ω_h^i, $i = \pm$ is complemented with its own fracture facets. The total set of facets for the bulk problem is

$$\mathscr{F}_h = \mathscr{F}_h^0 \cup \mathscr{F}_{h,+}^\Gamma \cup \mathscr{F}_{h,-}^\Gamma \cup \mathscr{F}_h^{\Gamma_D} \cup \mathscr{F}_h^{\Gamma_N} = \mathscr{F}_h^0 \cup \mathscr{F}_h^\Gamma \cup \mathscr{F}_h^{\partial\Omega},$$

where $\mathscr{F}_h^{\partial\Omega} = \mathscr{F}_h^{\Gamma_D} \cup \mathscr{F}_h^{\Gamma_N}$ collects the boundary facets, i.e. Dirichlet and Neumann facets, $\mathscr{F}_h^0$ collects the internal bulk facets and $\mathscr{F}_h^\Gamma = \mathscr{F}_{h,+}^\Gamma \cup \mathscr{F}_{h,-}^\Gamma$ the decoupled fracture facets. Note that we are assuming a partition Ω_h which respects the boundary conditions. Moreover we denote by $\mathscr{F}_h(\mathsf{P}) \subset \mathscr{F}_h$ the facets of the polyhedron P. Hereafter we denote with f a facet corresponding to the bulk problem, i.e. $\mathsf{f} \in \mathscr{F}_h$.

For the mesh, following [14], we assume three shape-regularity assumptions summarized in the following condition.

Condition (AM). There exist two positive real numbers N^s and ρ_s such that Ω_h admits a conforming sub-partition T_h of tetrahedra such that:

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

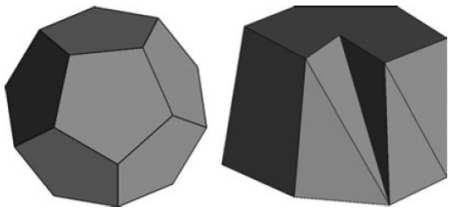


Figure 2.1: Admissible elements: shape-regular convex (left) and degenerate non-convex (right). Taken by [14].

- every tetrahedron $\mathsf{T} \in \mathsf{T}_h$ is regular, i.e. the ratio between the radius r_T of the inscribed sphere and the diameter h_T is bounded from below by

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

- 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 to a point $\overline{\mathbf{x}}_\mathsf{f} \in \mathsf{f}$. Moreover the tetrahedral sub-partition T_h is simple, i.e., it is build in the following way. First, each facet f is subdivided into triangles by connecting each vertex of f with the point $\overline{\mathbf{x}}_\mathsf{f}$. Second, each element P is decomposed into tetrahedra by connecting each vertex of P and each point $\overline{\mathbf{x}}_\mathsf{f}$, $\mathsf{f} \in \mathscr{F}_h(\mathsf{P})$, with the point $\overline{\mathbf{x}}_\mathsf{P}$.

These assumptions require weak restrictions on the shape of the admissible elements; indeed the grid Ω_h may contain generally shaped elements, like non-convex cells, see Figure 2.1.

We denote by $|\mathsf{P}|$ and $|\mathsf{f}|$ the volume of the polyhedron P and the area of the facet f, respectively. 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. In particular, for every couple of fracture facets $\{\mathsf{f}^+, \mathsf{f}^-\}$, we consider the same normal $\mathbf{n}_\mathsf{f} = \mathbf{n}_\Gamma$, where we remind that $\mathbf{n}_\Gamma$ is orientated from Ω^+ to Ω^-. Then, we denote by $\mathbf{n}_{\mathsf{P},\mathsf{f}}$ the unit normal vector of the facet $\mathsf{f} \in \mathscr{F}_h(\mathsf{P})$ outward orientated with respect to the polyhedron P. We define the orientation of a facet as $\alpha_{\mathsf{P},\mathsf{f}} = \mathbf{n}_\mathsf{f} \cdot \mathbf{n}_{\mathsf{P},\mathsf{f}}$ and clearly it holds $\alpha_{\mathsf{P},\mathsf{f}} \in \{-1, 1\}$. Note that, given $\mathsf{f} \in \mathscr{F}_h^\Gamma$, we have $\alpha_{\mathsf{P},\mathsf{f}} = 1$, if $\mathsf{f} \in \mathscr{F}_{h,+}^\Gamma$, and $\alpha_{\mathsf{P},\mathsf{f}} = -1$, if $\mathsf{f} \in \mathscr{F}_{h,-}^\Gamma$. We indicate with $\mathbf{x}_\mathsf{P}$ and $\mathbf{x}_\mathsf{f}$ the barycentres of the polyhedron P and of the facet f.

Now we introduce the mimetic spaces. We denote by Q_h the space of the discrete pressures. In particular, we associate a pressure degree of freedom $q_\mathsf{P} \in \mathbb{R}$ to every cell of the mesh, so that for $q_h \in Q_h$ we have $q_h = \{q_\mathsf{P}\}_{\mathsf{P} \in \Omega_h}$. Fixed an enumeration

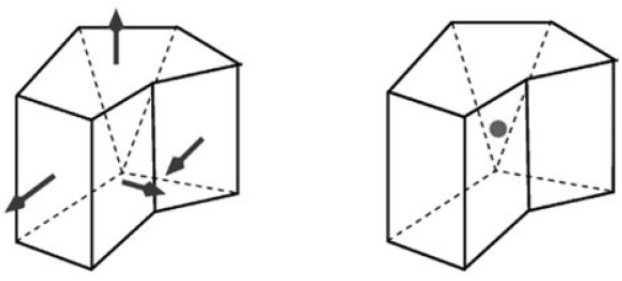


Figure 2.2: Degrees of freedom: velocity (left), pressure (right). Taken by [14].

of the cells, a discrete field q_h can be represented by an algebraic vector $q_h \in \mathbb{R}^{N_\mathsf{P}}$, where N_P is the number of cells of the mesh. Let us now introduce the space W_h of the discrete velocities. We associate a velocity degree of freedom $v_\mathsf{f} \in \mathbb{R}$ to every facet $\mathsf{f} \in \mathscr{F}_h$, so that, given $\mathbf{v}_h \in W_h$, we have $\mathbf{v}_h = \{v_\mathsf{f}\}_{\mathsf{f}\in\mathscr{F}_h}$. Fixed an enumeration of the facets of $\mathscr{F}_h$, a discrete vector field $\mathbf{v}_h \in W_h$ can be represented through an algebraic vector $\mathbf{v}_h \in \mathbb{R}^{N_\mathsf{f}}$, where N_f is the cardinality of $\mathscr{F}_h$, i.e. the total number of facets that takes into account the decoupling of fracture facets. We stress that, having decoupled the fracture facets, a discrete vector field $\mathbf{v}_h$ allows to reproduce velocity discontinuities across the fracture. In Figure 2.2 a sketch of the degrees of freedom for discrete velocity and pressure is shown. The discrete subspace $W_h^{\phi,\Gamma_N} \subset W_h$ is defined by incorporating the Neumann boundary conditions in W_h:

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

We also introduce the jump and the average operators across the fracture facet $\hat{\mathsf{f}} \in \Gamma_h$ as

$$[\![\mathbf{v}_h]\!]_{\hat{\mathsf{f}}} = v_{\mathsf{f}^+} - v_{\mathsf{f}^-}, \qquad \{\mathbf{v}_h\}_{\hat{\mathsf{f}}} = \frac{v_{\mathsf{f}^+} + v_{\mathsf{f}^-}}{2}.$$

Remark 2.1. *A cell-based discrete function $q_h \in Q_h$ can be defined to be constant inside each element, so that its pointwise value will be well-defined for every interior point of every mesh element. Similarly a facet-based discrete function $\boldsymbol{v}_h \in W_h$ can be defined to be constant over each mesh facet.*

2.2.2 The mimetic divergence operator

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

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

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

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 $\mathsf{P} \in \Omega_h$:

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

where $\mathbf{n}_\mathsf{P}$ is the outward unit normal of the element P. From the above formula, we can define the discrete divergence as

$$(\mathrm{div}_h\mathbf{v}_h)_\mathsf{P} = \frac{1}{|\mathsf{P}|}\sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} \alpha_{\mathsf{P},\mathsf{f}}|\mathsf{f}|v_\mathsf{f}, \tag{2.3}$$

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

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

2.2.3 The mimetic inner products

The next goal is to define suitable inner products in Q_h and W_h. For 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 complicated. It is defined by assembling element-wise 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 $\mathsf{M} \in \mathbb{R}^{N_{\mathsf{f}} \times N_{\mathsf{f}}}$ is a symmetric and positive definite matrix representing the mimetic inner product that approximates the one in L^2 weighted 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 and, as M, is symmetric and positive definite. With $N_{\mathsf{P}}^{\mathsf{f}}$ we have denoted the number of facets of the cell P.

Let us review some theoretical tools on the construction of a mimetic inner product; for more details see [14]. 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 $\displaystyle\int_{\mathsf{P}} qv \, dV$ can be computed exactly using the degrees of freedom, i.e. the components of the vector $\Pi_{\mathsf{P}}^{\mathscr{S}}(v)$.

Assumption (1) states that $S_{h,\mathsf{P}}$ is rich enough, assumption (2) is related to the accuracy of the mimetic scheme, whereas condition (3) is problem dependent and allows the practical construction of the discrete inner 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

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

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

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

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

with C_* and C^* that are two positive constants independent of P and v_P.

In the latter definition $||.||$ is a discrete norm that is problem dependent. The stability condition ensures the positive definiteness of the mimetic inner product and is important for the well-posedness of the discrete problem. 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 a discrete inner product that approximates the one in L^2 weighted with K^{-1}. Let K_P be an 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. Let us start from the stability condition.

Condition (**S**). There exist two positive constants σ_* and σ^*, independent from the mesh size h, such that for every element P it holds

$$\sigma_*|\mathsf{P}| \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} |v_\mathsf{f}|^2 \leq [v_\mathsf{P}, v_\mathsf{P}]_\mathsf{P} \leq \sigma^*|\mathsf{P}| \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} |v_\mathsf{f}|^2 \qquad \forall\ v_{h,\mathsf{P}} \in W_{h,\mathsf{P}}. \tag{2.9}$$

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

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

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

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

where $\mathbb{P}_1(\mathsf{P})$ are the linear polynomials defined on P. It is trivial to verify that $S_{h,\mathsf{P}}$ ensures conditions (1) and (2).

Now we state the consistency condition.

Condition (**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 that here we do not simplify K_P in order to stress that the natural L^2 inner product in the velocity space is defined with the tensorial weight $\mathsf{K}_\mathsf{P}^{-1}$.

Now we show that the right-hand side of (2.10) 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

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

since $\mathrm{div}_{\mathsf{P}}\mathbf{v}_{\mathsf{P}}^I = (\mathrm{div}\mathbf{v})|_{\mathsf{P}} = \text{const}$ and $v_{\mathsf{f}}^I = \mathbf{v}\cdot\mathbf{n}_{\mathsf{f}} = \text{const}$, 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 average normal components of $\mathbf{v}$ on the facets $\mathsf{f} \in \mathscr{F}_h(\mathsf{P})$ are all what is needed to compute the integral and so condition (3) about $S_{h,\mathsf{P}}$ holds.

Condition (**C**) holds for every linear polynomial q. Now let us restrict to the linear polynomials in P of zero mean, i.e., we require $q \in \widetilde{\mathbb{P}}_1(\mathsf{P}) = \{z \in \mathbb{P}_1(\mathsf{P}) : \int_{\mathsf{P}} z\ dV = 0\}$. In this way the volume integral in the equation (2.11) is zero and the consistency condition (**C**) becomes

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

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 (**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 barycentre of the cell P. Let us define, for $j = 1, 2, 3$, a vector $\mathsf{N}_j \in \mathbb{R}^{N_{\mathsf{P}}^{\mathsf{f}}}$. The definition is the following:

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

Let us define the matrix $\mathsf{N}_{\mathsf{P}} \in \mathbb{R}^{N_{\mathsf{P}}^{\mathsf{f}}\times 3}$ 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 basis of $\mathbb{R}^3$, the N_{P} formula is

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

Condition (**C**) can be reformulated as

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

where the vector $\mathsf{R}_j \in \mathbb{R}^{N_{\mathsf{P}}^{\mathsf{f}}}$ 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 $\mathsf{R}_{\mathsf{P}} \in \mathbb{R}^{N_{\mathsf{P}}^{\mathsf{f}}\times 3}$ as $\mathsf{R}_{\mathsf{P}} = [\mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3]$. Because $\mathbf{v}_{\mathsf{P}}^I$ 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 $\mathsf{R_P}$. The face integral (see (2.15)) of a linear function equals its value at the barycentre $\mathbf{x_f}$ times the facet area $|\mathsf{f}|$. Thus,

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

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

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

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

Lemma 2.3. *For any polyhedral cell* P *we have*

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

For the proof of this result we refer to [14]. Using Lemma 2.3 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 $Ker(\mathsf{M_P^1}) = Range(\mathsf{N_P})$; this allows to keep the consistency (**C**). The important point to make $\mathsf{M_P}$ stable is to ensure that $\mathsf{M_P^1}$ has the same scaling of $\mathsf{M_P^0}$. An effective choice is given by the scaled orthogonal projector. So that finally the $\mathsf{M_P}$, which satisfies 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

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

2.2.4 Discretization of the bulk 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

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

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

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

where

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

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

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

The coefficient $\eta_{\Gamma,\hat{\mathsf{f}}}$ is an average value of η_Γ over the fracture facet $\hat{\mathsf{f}}$.

2.3 Finite volume approximation for the fracture network

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 approximation we refer to [30]. 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 $\boldsymbol{\tau}_\Gamma$, which form an orthonormal basis of the plane. With this coordinate system the fracture equation is

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

where clearly the differential operators collect derivatives with respect to 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 $\hat{\mathsf{f}}_i \in \Gamma_h$ the latter equation:

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

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

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

where $\mathbf{u}_h \in W_h^{g,\Gamma_N}$ is the discrete velocity, whereas $f_{\Gamma,i}$ and $l_{\Gamma,i}$ are average values over the fracture facet $\hat{\mathsf{f}}_i$ of the forcing term and the aperture. So the equation

becomes

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

Applying the divergence theorem and splitting the resulting boundary integral over the edges e_{ij} of $\hat{\mathsf{f}}_i$, we get

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

where $\mathbf{n}_{ij}$ is the normal to the edge e_{ij}, outward orientated with respect $\hat{\mathsf{f}}_i$, and N_e^i is the number of edges of the facet $\hat{\mathsf{f}}_i$.

We consider a cell-centred scheme and so we locate the degrees of freedom of the fracture pressure in the barycentres of the fracture facets. Every edge integral in the equation (2.28) represents an outward flux from the fracture cell $\hat{\mathsf{f}}_i$ 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 $\hat{\mathsf{f}}_i$, where, the summation collects the contributions between $\hat{\mathsf{f}}_i$ and the adjacent fracture cells, and the integral with the velocity jump collects the contributions between $\hat{\mathsf{f}}_i$ 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 barycentres of two adjacent fracture cells.

Following [45], let us consider two cells that share an edge and introduce the following geometrical quantities: the barycentres C_1 and C_2 of the cells; the midpoint C_0 of the edge; the 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; the unit vectors $\mathbf{f}_1$ from C_0 to C_1 and $\mathbf{f}_2$ from C_0 to C_2. To have a graphical view of the situation see Figure 2.3. 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 symmetry 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 cell $\hat{\mathsf{f}}_i$ (see [30] 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

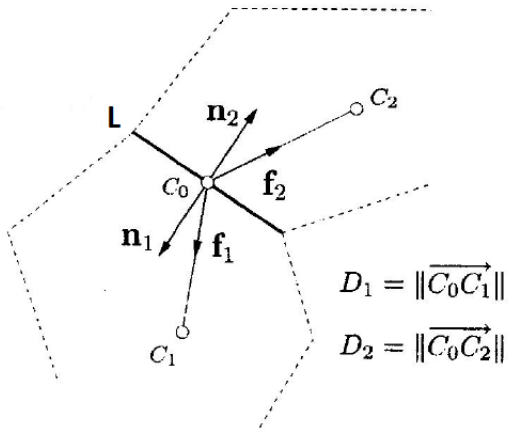


Figure 2.3: Geometrical quantities involved in TPFA. Taken by [61].

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_{12} is the transmissibility between cells 1 and 2. More generally 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 across the edge e_{ij}. We define the coefficient

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

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 2.4 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_g and s_j 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

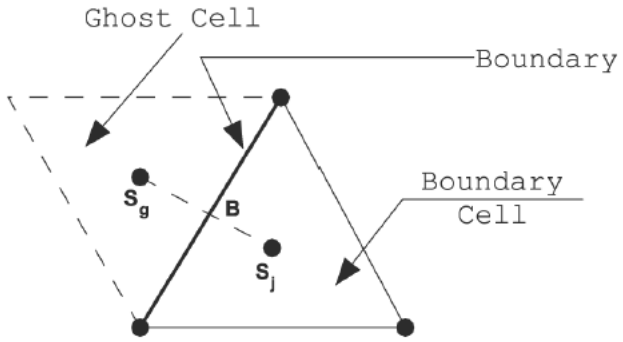


Figure 2.4: Sketch of the boundary for the ghost cell approach. Taken by [61].

flux that can be computed as $\phi_j L_j$, where ϕ_j is determined through the boundary condition and L_j is the length of the boundary edge.

Now we focus on the treatment of fractures intersections. Up to now we have supposed only one fracture to ease the presentation of the theoretical part, but the code handles network of fractures and so we briefly describe the treatment of fractures 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 a network of fractures 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 follows:

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

For more details on the "star-delta" approach see [45].

Remark 2.2. *We have considered a local coordinates 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 coordinates system is not needed, because the TPFA scheme of FV involves only geometrical quantities that do not depend on the coordinates system. This simplifies things since the knowledge of a local coordinates system may be complex in a network of fractures.*

2.4 Algebraic formulation

We start describing the linear system arising from the hybrid MFD-FV numerical scheme. From the mimetic formulation of the bulk problem, i.e. from 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: $\mathsf{M}_c \in \mathbb{R}^{N_\mathsf{f} \times N_\mathsf{f}}$ represents the weighted mimetic inner product plus 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 $-[p_h, \mathrm{div}_h \mathbf{v}]_{Q_h}$; $\mathsf{C} \in \mathbb{R}^{N_\Gamma \times N_\mathsf{f}}$ takes into account the other term of the coupling conditions $\sum_{\hat{\mathsf{f}} \in \Gamma_h} |\hat{\mathsf{f}}| p_{\Gamma,\hat{\mathsf{f}}} [\![\mathbf{v}_h]\!]_{\hat{\mathsf{f}}}$; $\mathbf{g} \in \mathbb{R}^{N_\mathsf{f}}$ takes into account the Dirichlet and Neumann boundary conditions on bulk; $\mathbf{h} \in \mathbb{R}^{N_\mathsf{P}}$ 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 bulk pressure p_h, respectively.

From the discretization of the fracture problem with the TPFA version 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 the fracture. 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:

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

Now we reduce the linear system to a saddle point algebraic problem. 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. [20]. 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

$$\widetilde{\mathsf{B}} = \begin{bmatrix} \mathsf{B} \\ \mathsf{C} \end{bmatrix}, \quad \widetilde{\mathsf{T}} = \begin{bmatrix} 0 & 0 \\ 0 & \mathsf{T} \end{bmatrix}. \tag{2.36}$$

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

$$\begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^\mathsf{T} \\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \widetilde{\mathbf{p}} \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ \widetilde{\mathbf{h}} \end{bmatrix}, \tag{2.37}$$

which is the form of a 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.

Now, following [7], we report a non-singularity result for the matrix of the system in the case of a single fracture that cuts the whole domain. For this purpose, let us review a non-singularity result for generalized saddle-point matrices; see [20] for the details.

Theorem 2.4. *Let us consider the following block matrix*

$$\mathcal{A} = \begin{bmatrix} \mathsf{A} & \mathsf{B}^\mathsf{T} \\ \mathsf{B} & -\mathsf{C} \end{bmatrix}.$$

where $\mathsf{A} \in \mathbb{R}^{n\times n}$ is a symmetric positive definite matrix, $\mathsf{B} \in \mathbb{R}^{m\times n}$ and $\mathsf{C} \in \mathbb{R}^{m\times m}$ is a symmetric positive semi-definite matrix. If $Ker(\mathsf{C}) \cap Ker(\mathsf{B}^\mathsf{T}) = \{\mathbf{0}\}$, then the saddle point matrix $\mathcal{A}$ is non-singular. In particular, $\mathcal{A}$ is invertible if B has full rank.

For our problem we have the following result.

Proposition 2.1. *The linear system (2.37) admits a unique solution.*

Proof. We start by describing the properties of matrices M_c and $\widetilde{\mathsf{T}}$. The matrix M_c is symmetric positive definite by construction. The transmissibility matrix T is symmetric and is a Z-matrix with positive diagonal elements by construction. If $\partial\Gamma_D = \emptyset$, then $\mathsf{T}_{ii} = \sum_j \mathsf{T}_{ij} \ \forall \ i = 1, \dots, N_\Gamma$. In this case T is diagonally dominant with the vector $\begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix}$ in the kernel and it is symmetric semi-positive definite. Whereas, if $\partial\Gamma_D \neq \emptyset$, we have, at least for one row, $\mathsf{T}_{ii} > \sum_j \mathsf{T}_{ij}$. In this case T is symmetric positive definite. Anyway the block $\widetilde{\mathsf{T}}$ is symmetric positive semi-definite. Regarding the matrix $\widetilde{\mathsf{B}}$, we have that $Ker(\widetilde{\mathsf{B}}^{\mathsf{T}}) = \{\mathbf{0}\}$; see [7] for the proof. Therefore we can apply Theorem 2.4 and conclude that linear system (2.37) admits a unique solution. □

Remark 2.3. *We stress that the positive definiteness of* T *is not necessary for the non-singularity of the linear system. Therefore we can conjecture that the hypothesis* $\partial\Gamma_D \neq \emptyset$ *for the well-posedness at the continuous level can be relaxed. Indeed the well-posedness for a single immersed fracture, with no Dirichlet conditions imposed on its boundary, has been proved in [4, 60], even though different formulations have been adopted in these works.*

Remark 2.4. *In the case of network of fractures the linear system is still of the form (2.35) and the saddle point reduction can be done exactly in the same way. The proof of the well-posedness of the discrete model, in the case of an immersed fractures network, for a mimetic mixed formulation both in bulk and fractures, can be found in [34].*

2.5 Practical imposition of Neumann boundary conditions

In the theoretical analysis presented up to here we have supposed a strong imposition of Neumann boundary conditions in bulk. In particular the velocity degrees of freedom corresponding to Neumann facets are prescribed averaging the Neumann function ϕ. However in the code a different approach has been adopted. Indeed, from an implementative point of view, the strong imposition of velocity boundary conditions requires the modification of the rows of the system corresponding to Neumann facets and this implies having to handle a non symmetric matrix of the system. However a symmetric matrix turns out in good properties in terms of iterative solvers and preconditioning and, moreover, 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 resolution techniques like the UmfPack direct solver are unavailable. So it has been decided, concerning the implementation of the code, to impose the boundary conditions through a Nitsche technique that preserves the symmetry of the system and can be implemented well with a column storage of the sparse matrix of the system.

Roughly speaking, it consists of a modification of the penalty approach, which would convert the Neumann boundary condition in a Robin one with a penalty term, to recover the consistency of the numerical scheme. For a detailed description of the Nitsche method we refer to [26]. Here we show the discrete bulk scheme with the Nitsche imposition of Neumann boundary conditions on velocity. The formulation reads as follows: find $(\mathbf{u}_h, p_h) \in W_h \times Q_h$ such that

$$\begin{cases} [\mathbf{u}_h, \mathbf{v}_h]_{W_h} + [\mathbf{u}_h, \mathbf{v}_h]_\Gamma + \displaystyle\sum_{\mathsf{f}\in\Gamma_h^N} \gamma h|\mathsf{f}| u_\mathsf{f} v_\mathsf{f} - [p_h, \mathrm{div}_h \mathbf{v}_h]_{Q_h} - \sum_{\mathsf{f}\in\Gamma_h^N} |\mathsf{f}| p_\mathsf{P} v_\mathsf{f} \\ \qquad + \displaystyle\sum_{\hat{\mathsf{f}}\in\Gamma_h} |\hat{\mathsf{f}}| p_{\Gamma,\hat{\mathsf{f}}} [\![\mathbf{v}_h]\!]_{\hat{\mathsf{f}}} = L_h(\mathbf{v}_h) + \sum_{\mathsf{f}\in\Gamma_h^N} \gamma h|\mathsf{f}| \phi_\mathsf{f} v_\mathsf{f} \qquad \forall\ \mathbf{v}_h \in W_h \\ -[\mathrm{div}_h \mathbf{u}_h, q_h]_{Q_h} - \displaystyle\sum_{\mathsf{f}\in\Gamma_h^N} |\mathsf{f}| u_\mathsf{f} q_\mathsf{P} = G_h(q_h) - \sum_{\mathsf{f}\in\Gamma_h^N} |\mathsf{f}| \phi_\mathsf{f} q_\mathsf{P} \qquad \forall\ q_h \in Q_h, \end{cases} \tag{2.38}$$

where $\phi_\mathsf{f} = 1/|\mathsf{f}| \int_\mathsf{f} \phi \, dS$, γ is the penalty coefficient (which must be sufficiently high). The h in the penalty terms is a piecewise constant function defined on the Neumann facets and such that $h|_\mathsf{f} = diam(\mathsf{f})$, i.e. evaluated on a Neumann facet its value is the facet diameter. The utility of h is to make the scaling of the penalty term comparable to that of the inner product. As we can see from the discrete model (2.38) the Nitsche imposition of Neumann boundary conditions requires additional boundary terms in the formulation, more precisely, from an algebraic point of view, these boundary terms affect the matrices M_c, B, B^T and the vectors $\mathbf{g}$, $\mathbf{h}$.

Chapter 3

Preconditioners for the discrete problem

In this chapter we present some techniques for preconditioning the linear system arising from the discrete problem in order to develop an efficient iterative solver. In previous works that make use of MFD in fractured porous media, see [7, 61, 34, 60], the linear system was solved with direct methods and so the problem of preconditioning was not studied; on the other hand only 2D configurations were considered in these works. In 3D problems, direct computations become unfeasible mainly due to memory limitation and the usage of an iterative solver becomes mandatory. However in most cases, especially those concerning the numerical resolution of PDEs, the linear system is not well-conditioned and a preconditioner is needed to obtain an effective technique. Our work takes shape as a 3D generalization of [7] and focuses on the development of suitable preconditioners for solving iteratively the resulting linear system.

Only a few preconditioning proposals are available in literature concerning fractured porous media, also with respect to other discretization techniques. For instance, in [16], where a VEM formulation is used, a FETI preconditioner [55] is employed because the problem, which involves only the fractures network and not the surrounding rock, is formulated as a domain decomposition problem and the FETI preconditioner is one of the standard techniques in this context. Another case is the one presented in [28], a XFEM formulation for non-matching grids, where a block diagonal preconditioner is adopted. We stress that for Finite Elements Methods several proposals of optimal preconditioners, with respect to both the mesh size and the PDE's coefficients, have been developed and are available in literature for a wide range of problems. For instance, for Mixed Finite Elements applied to diffusion problems in mixed form, like in [28], two optimal preconditioners have been proposed in [54]. However for MFD, which is a more recent technique, again under development from different point of views, only a few preconditioning proposals are available in literature and most of them suggest multigrid preconditioners. For example, in [8] a two-level preconditioner for a vertex-based mimetic discretization of a diffusion problem in primal form is presented, while in [47] an algebraic multigrid preconditioner is employed for a diffusion problem in mixed form. The development of coarsening procedures with polyhedral grids are quite complex in itself, and in fractured porous media, especially with complicated networks of fractures, this

may be a very complex task because the mesh has to keep the conformity to the fractures. Therefore, even if multigrid looks promising, in this work we preferred to focus on more classic preconditioners.

More precisely, following [20], we have developed suitable preconditioners working on the saddle point block partitioning of the matrix of the linear system. In particular we present a block triangular preconditioner, an inexact block LU (ILU) preconditioner and an Hermitian/skew-Hermitian splitting (HSS) preconditioner. Moreover, we also present and prove an estimate for the condition number of the matrix of the system, extending the analysis presented in [51] for a diffusion problem in mixed form (i.e. a porous medium without fractures). Characterizing the spectral properties of the matrix of the system is very important for preconditioning, because the main goal of a preconditioner is to reduce the condition number and/or to cluster the eigenvalues, to better the convergence properties of the iterative solver.

In this chapter we firstly describe a brief review of Krylov subspace methods, focusing on the GMRES. Then, we present a general introduction to the problem of preconditioning and some considerations on preconditioning a saddle point problem. Finally we describe the three preconditioning techniques and, after having presented several preliminary results and assumptions, we prove an estimate of the condition number of the matrix of the system.

The preconditioners have been widely tested in Chapter 5, by comparing their performances with direct computations and by assessing their sensitivity with respect to the mesh size and the problem parameters.

3.1 Krylov subspace methods and GMRES

In this section, following [20], we present a brief review of Krylov subspace methods, without entering into the implementation details and focusing on the main properties and in particular on the GMRES, which is the solver used in this thesis for all the numerical simulations. For more details on this class of iterative methods we refer to [32, 58].

Let us consider a generic linear system

$$\mathcal{A}\mathbf{x} = \mathbf{b}, \tag{3.1}$$

where $\mathcal{A} \in \mathbb{R}^{n\times n}$ is non-singular and typically large and sparse, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^n$.

Let us denote with $\mathbf{x}_0$ the initial guess for the solution $\mathbf{x}$ of system (3.1). The initial residual is defined as $\mathbf{r}_0 = \mathbf{b} - \mathcal{A}\mathbf{x}_0$. Krylov subspace methods are iterative methods that look for a k-th solution $\mathbf{x}_k$ satisfying

$$\mathbf{x}_k \in \mathbf{x}_0 + \mathcal{K}_k(\mathcal{A}, \mathbf{r}_0), \qquad k = 1, 2, \dots, \tag{3.2}$$

where

$$\mathcal{K}_k(\mathcal{A}, \mathbf{r}_0) = \text{span}\{\mathbf{r}_0, \mathcal{A}\mathbf{r}_0, \dots, \mathcal{A}^{k-1}\mathbf{r}_0\}$$

denotes the k-th Krylov subspace generated by $\mathcal{A}$ and $\mathbf{r}_0$. Krylov subspaces form a nested sequence that ends with dimension $d = \dim\mathcal{K}_n(\mathcal{A}, \mathbf{r}_0) \leq n$. More precisely, it holds

$$\mathcal{K}_1(\mathcal{A}, \mathbf{r}_0) \subset \cdots \subset \mathcal{K}_d(\mathcal{A}, \mathbf{r}_0) = \cdots = \mathcal{K}_n(\mathcal{A}, \mathbf{r}_0)$$

and, for every $k \leq d$, we have that $\dim\mathcal{K}_k(\mathcal{A}, \mathbf{r}_0) = k$.

Since k degrees of freedom are involved in the search for $\mathbf{x}_k$, k constraints have to be prescribed to determine uniquely $\mathbf{x}_k$. This is achieved by requiring that the k-th residual is orthogonal, in the Euclidean sense, to a constraint space $\mathcal{C}_k$, i.e.

$$\mathbf{r}_k = \mathbf{b} - \mathcal{A}\mathbf{x}_k \in \mathbf{r}_0 + \mathcal{A}\mathcal{K}_k(\mathcal{A}, \mathbf{r}_0), \qquad \mathbf{r}_k \perp \mathcal{C}_k. \tag{3.3}$$

We mention that Krylov subspace methods fall under a wide class of iterative techniques called *projection methods*, see [58] for more details. Indeed relations (3.2) and (3.3) show that Krylov subspace methods are just based on a *projection process*.

The constraint space $\mathcal{C}_k$ is selected depending on the properties of the matrix of the linear system $\mathcal{A}$. The two most important cases are summarized in the following theorem, see [58] for the proof.

Theorem 3.1. *Suppose that the Krylov subspace $\mathcal{K}_k(\mathcal{A}, \mathbf{r}_0)$ has dimension k. If*

(C) *$\mathcal{A}$ is symmetric and positive definite and $\mathcal{C}_k = \mathcal{K}_k(\mathcal{A}, \mathbf{r}_0)$, or*

(M) *$\mathcal{A}$ is non-singular and $\mathcal{C}_k = \mathcal{A}\mathcal{K}_k(\mathcal{A}, \mathbf{r}_0)$,*

then there exists an unique iterate solution $\mathbf{x}_k$ of the form (3.2) whose residual $\mathbf{r}^k$ satisfies (3.3).

The (**C**) case characterizes the Conjugate Gradient method (CG) for symmetric and positive definite matrices. If $\mathcal{A}$ is not symmetric and positive definite, an approximate solution $\mathbf{x}_k$ satisfying (3.2) and (3.3) may not exist. Nevertheless, there are several implementations of this projection process, like the Full Orthogonalization Method (FOM).

The (**M**) case characterizes the Minimum Residual Method (MINRES) for non-singular symmetric matrices and the Generalized Minimum Residual Method (GMRES) for generic non-singular matrices.

Several other choices of $\mathcal{C}_k$ can be found in literature. For instance, if $\mathcal{A}$ is unsymmetric, an alternative is to consider $\mathcal{C}_k = \mathcal{K}_k(\mathcal{A}^\mathsf{T}, \mathbf{r}_0)$. This technique represents a generalization of the case (**C**) and identifies the Biconjugate Gradient Method (BiCG). However for a general unsymmetric matrix $\mathcal{A}$ the iterate $\mathbf{x}_k \in \mathbf{x}_0 + \mathcal{K}_k(\mathcal{A}, \mathbf{r}_0)$ satisfying the constraint of orthogonality with $\mathcal{C}_k = \mathcal{K}_k(\mathcal{A}^\mathsf{T}, \mathbf{r}_0)$ may not exist and, in practice, an irregular convergence may occur. This is fixed with an additional minimization step, in order to stabilize the convergence, in the Stabilized Biconjugate Gradient Method (BiCGStab), which avoids to use the matrix $\mathcal{A}^\mathsf{T}$.

In exact arithmetic, the methods defined by (**C**) and (**M**) terminate with the exact solution in $d = \dim\mathcal{K}_n(\mathcal{A}, \mathbf{r}_0)$ steps. This property is called *finite termination property*. In practice, the method is stopped for $k < d$ with $\mathbf{x}_k$ that is a sufficiently accurate approximation of the exact solution $\mathbf{x}$.

To study how fast a Krylov method can reach a given level of accuracy, it is important to note that the orthogonality condition defined in (3.3) is often equivalent to an optimality condition that involves the error $\mathbf{x} - \mathbf{x}_k$ or the residual $\mathbf{r}_k$ for a certain norm. Hereafter we consider only the GMRES for general non-singular matrices $\mathcal{A}$, since we have used this method for all the computations in Chapter 5.

From the orthogonality constraint defined in (**M**), it follows the optimality condition

$$||\mathbf{r}_k||_2 = ||\mathbf{b} - \mathcal{A}\mathbf{x}_k||_2 = \min_{\mathbf{z}\in\mathbf{x}_0+\mathcal{K}_k(\mathcal{A},\mathbf{r}_0)} ||\mathbf{b} - \mathcal{A}\mathbf{z}||_2 = \min_{p\in\Pi_k} ||p(\mathcal{A})\mathbf{r}_0||_2, \tag{3.4}$$

where Π_k denotes the set of polynomials of degree at most k with value 1 at the origin. Therefore the GMRES is based on computing the solution $\mathbf{x}_k$ that minimizes the 2-norm of the residual $\mathbf{r}_k$ in the subspace $\mathcal{K}_k(\mathcal{A}, \mathbf{r}_0)$.

Now let us suppose that $\mathcal{A}$ is diagonalizable, i.e., $\mathcal{A} = \mathsf{Q}\mathsf{D}\mathsf{Q}^{-1}$, $\mathsf{D} = \mathtt{diag}(\lambda_i)$, where λ_i identifies the i-th eigenvalue of $\mathcal{A}$. The Euclidean norm of the k-th residual satisfies

$$\begin{aligned} ||\mathbf{r}_k||_2 &= \min_{p\in\Pi_k} ||p(\mathcal{A})\mathbf{r}_0||_2 \leq \min_{p\in\Pi_k} ||\mathsf{Q}p(\mathsf{D})\mathsf{Q}^{-1}||_2||\mathbf{r}_0||_2 \\ &\leq ||\mathsf{Q}||_2||\mathsf{Q}^{-1}||_2 \min_{p\in\Pi_k} ||p(\mathsf{D})||_2||\mathbf{r}_0||_2 \end{aligned}.$$

The resulting estimate is given by

$$||\mathbf{r}_k||_2 \leq ||\mathbf{r}_0||_2 K(\mathsf{Q}) \min_{p\in\Pi_k} \max_{\lambda_j} |p(\lambda_j)|. \tag{3.5}$$

So we have that the Euclidean norm of the normalised k-th residual $||\mathbf{r}_k||_2/||\mathbf{r}_0||_2$ is bounded from below by the value of a polynomial approximation problem of the eigenvalues of $\mathcal{A}$, multiplied by the condition number of the eigenvector matrix Q of $\mathcal{A}$.

If $\mathcal{A}$ is symmetric (normal), then Q is a unitary matrix and $K(\mathsf{Q}) = 1$. In this case the estimate (3.5) is sharp, in the sense that, for every non-singular symmetric matrix $\mathcal{A}$ and for every iteration k, there exists a $\mathbf{r}_0^{(k)}$ such that equality holds, see [39] for the proof. So the "worst-case" behaviour of the GMRES is completely determined by the eigenvalues of the matrix. Moreover, the sharp estimate suggests some intuitive considerations on how the distribution of the eigenvalues influences the convergence of the GMRES. For example, if the eigenvalues form a tight cluster around a single point far away from the origin, fast convergence can be expected; in general a clustered spectrum is favourable in terms of convergence of GMRES. Widely spread eigenvalues, on the other hand, will potentially lead to slow convergence. We stress that for the CG the same estimate holds and, being $\mathcal{A}$ also positive definite, an estimate involving only the condition number of $\mathcal{A}$ can be derived, see [58].

If $\mathcal{A}$ is unsymmetric (non-normal) $K(\mathsf{Q})$ may be very large and the inequality (3.5) may strongly overestimate the worst-case relative residual norm. In such a case the eigenvalues of the matrix $\mathcal{A}$ may also give misleading information on the convergence behaviour of the GMRES. In general, no convergence analysis based only on the matrix of the system is really exhaustive in the non-normal case and also the initial residual $\mathbf{r}_0$ should be included in the analysis [20].

Now we point out some important features of GMRES without entering into the implementation details. At each iteration k the method has to store the basis of the k-th Krylov subspace $\mathcal{K}_k(\mathcal{A}, \mathbf{r}_0)$. So its memory requirement increases with the number of iterations. A common way to reduce the expensive memory cost of the standard version of GMRES is to introduce a restart, i.e., after a prescribed number

of iterations, the method is stopped and restarted taking the current solution as initial guess. In practice, except in case of fast convergence, the restart is necessary. It must be noted that restarting avoids to store a too large number of vectors, but it may worsen the convergence of the method. So a reasonable compromise must be set between robustness and memory requirement.

If a preconditioned GMRES is employed and the preconditioner changes from one iteration to another, the flexible version of the method is recommended to enhance the robustness of the iterative routine. The FGMRES is an extension of the GMRES that accounts for the variations of the preconditioner. For that purpose it requires to store also the basis of the "preconditioned" Krylov subspace, thus doubling the memory requirement with respect to GMRES. See [57] for details on FGMRES. A common situation in which the preconditioner is "not constant" is when another Krylov subspace method is used as a preconditioner.

3.2 A brief introduction to preconditioning

Let us consider the generic linear system (3.1). The solution of a large linear system is usually the most time-consuming part of numerical simulations in science and engineering and the discretization of PDEs is a main topic where large and sparse linear systems arise and need to be solved.

Direct methods, based on the factorization of the matrix $\mathcal{A}$ into easily invertible matrices, are very robust and require an amount of time and memory that is typically predictable. For sparse matrices there exist several direct methods that are robust and reliable, see [29] for a survey on this topic. We mention for instance multi-frontal methods like UmfPack or MUMPS, implemented in several scientific libraries. However direct methods scale poorly, in terms of number of operations and especially of memory consumption, with respect to the problem size. If most of 2D computations in numerical resolution of PDEs can be solved in a reasonable amount of time with direct methods, in 3D their usage becomes often unfeasible and the employment of an iterative solver becomes mandatory. Iterative solvers require less memory storage and often less operations, but they have less reliability, and this is particularly true in numerical discretization of PDEs where the matrix $\mathcal{A}$ is often badly conditioned. In such cases the development of a suitable preconditioner for the matrix of the system is the only option to obtain convergence in a reasonable amount of time. We focus on preconditioning for accelerating and improving reliability of Krylov-subspace methods. The development of effective preconditioners is a very active area of research, because an optimal-general purpose preconditioner does not exist and for saddle point problems the situation is even more problematic.

The term *preconditioning* refers to transforming the linear system (3.1) in another linear system that has more favourable properties for being solved in an iterative way. The *preconditioner* is the matrix that realizes such transformation. In general the aim of preconditioning is the improvement of the spectral properties of $\mathcal{A}$. If the matrix is symmetric and positive definite, the CG method is typically used and, since its rate of convergence depends only on the condition number of $\mathcal{A}$, in such cases the goal of preconditioning is to reduce the condition number of the matrix of the system. If the matrix of the system is only symmetric (normal), we

have described in the previous section that the spectral properties of the matrix determine completely the worst-case behaviour of GMRES, so that a clustered spectrum turns out in fast convergence. Therefore, if $\mathcal{A}$ is only symmetric, the aim of preconditioning can be considered to cluster the eigenvalues of the matrix of the system. For general unsymmetric (non normal) matrices the situation is more complicated and not only the matrix spectrum may affect the rate of convergence of GMRES, also the initial residual vector has to be taken into account. In any case a clustered spectrum far away from the origin often turns out in rapid convergence. So the fundamental aim of preconditioning can be considered to improve the spectral properties of the matrix of the system in this regard. For a more detailed discussion on this topic see [20].

Let $\mathcal{P}$ be a non-singular matrix approximating $\mathcal{A}$ in some sense, then we can reformulate the linear system (3.1) as

$$\mathcal{P}^{-1}\mathcal{A}\mathbf{x} = \mathcal{P}^{-1}\mathbf{b}, \tag{3.6}$$

where $\mathcal{P}$ is the preconditioner. This modified linear system has the same solution of (3.1) but, depending on $\mathcal{P}$, an accurate iterative solution may be more easily computable. The system (3.6) is preconditioned from the left, otherwise also a preconditioning from the right is possible:

$$\mathcal{A}\mathcal{P}^{-1}\mathbf{y} = \mathbf{b}, \quad \mathbf{x} = \mathcal{P}^{-1}\mathbf{y}, \tag{3.7}$$

or a split preconditioning:

$$\mathcal{P}_1^{-1}\mathcal{A}\mathcal{P}_2^{-1}\mathbf{y} = \mathcal{P}_1^{-1}\mathbf{b}, \quad \mathbf{x} = \mathcal{P}_2^{-1}\mathbf{y}, \tag{3.8}$$

where $\mathcal{P} = \mathcal{P}_1\mathcal{P}_2$. The choice depends mainly on the iterative solver, e.g. the GMRES and the minimizing residual methods are typically implemented with right preconditioning. We stress that the three preconditioned matrices $\mathcal{P}^{-1}\mathcal{A}$, $\mathcal{A}\mathcal{P}^{-1}$, $\mathcal{P}_1^{-1}\mathcal{A}\mathcal{P}_2^{-1}$ are similar and so they have the same eigenvalues. Note also that the matrix of the preconditioned system, e.g. $\mathcal{P}^{-1}\mathcal{A}$ for left preconditioning, may have different properties of the original matrix $\mathcal{A}$, e.g. preconditioning may destroy the symmetry of the system, and this must be taken into account for selecting a proper iterative method.

It is very important to note that computing $\mathcal{P}^{-1}\mathcal{A}$ explicitly is not required. When Krylov-subspace methods are employed the iterative algorithm involves only the two following operations:

- matrix-vector product $\mathbf{y} = \mathcal{A}\mathbf{x}$,
- solution of the linear system $\mathcal{P}\mathbf{z} = \mathbf{r}$.

Note that often the complete storage of $\mathcal{P}$ can be avoided, because solving $\mathcal{P}\mathbf{z} = \mathbf{r}$ may not require the explicit construction of $\mathcal{P}$. Clearly the need of solving the linear system with $\mathcal{P}$, which must be done at each iteration and maybe more then once per iteration, requires the development of preconditioners that can be rapidly inverted. But a preconditioner, for damping the condition number of the system, has to be an approximation of $\mathcal{A}$, and the more $\mathcal{P}$ approaches $\mathcal{A}$, the more inverting

$\mathcal{P}$ becomes expensive. The trade off is between the need to have an iterative method that shows a rapid converge and the need to perform iterations that are not too expensive. A favourable balance must be found and it is the key of developing an efficient preconditioner. We note that another aspect to be taken into account is the cost of constructing the preconditioner.

In general there are two ways of constructing preconditioners. The first approach consists of purely algebraic techniques, like incomplete factorizations or sparse algebraic inverses, which are independent from the problem that has originated the linear system, see [56, 17]. They are useful for the implementation of general purpose solvers and they do not require any type of a priori knowledge of the original problem. The second approach is based on developing preconditioners that are tailored to the problem at hand. They require more information about the problem that has originated the linear system (e.g. for PDEs details on the discretization, on the geometry, on the coefficients of the model and so on) but typically this approach leads to more effective preconditioners.

3.3 Preconditioning for saddle point problems

In this section we make some preliminary considerations on iterative solvers and preconditioning for saddle point linear systems. We consider linear systems of the type arising from our problem, but our considerations hold for a wider class of saddle point problems. The system we consider is of the form

$$\begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \widetilde{\mathbf{p}} \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ \widetilde{\mathbf{h}} \end{bmatrix}, \quad \text{or} \quad \mathcal{A}\mathbf{x} = \mathbf{b}, \tag{3.9}$$

where the matrix M_c is symmetric and positive definite, $\widetilde{\mathsf{B}}^{\mathsf{T}}$ has empty null space and $\widetilde{\mathsf{T}}$ is symmetric semi-positive definite; see (2.36) for the block definitions. Note that, because of the block $\widetilde{\mathsf{T}} \neq 0$, the problems we are considering are called more rigorously "generalized saddle point problems".

With formulation (3.9) the matrix of the system is symmetric and indefinite. Another possible formulation consists of changing the sign of the second block equation of the system, in order to have an unsymmetric and positive definite matrix:

$$\begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ -\widetilde{\mathsf{B}} & \widetilde{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \widetilde{\mathbf{p}} \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ -\widetilde{\mathbf{h}} \end{bmatrix}, \quad \text{or} \quad \widehat{\mathcal{A}}\mathbf{x} = \widehat{\mathbf{b}}. \tag{3.10}$$

We stress that in any case the matrix of the system cannot be symmetric and positive definite and solving iteratively the system may be challenging. With the second version of the system we have that the eigenvalues of $\widehat{\mathcal{A}}$ lie in the right-half plane of $\mathbb{C}$, this may be (but not necessarily) a favourable property in solving the system with a Krylov subspace method. However in this way we loose the symmetry of the system and the convergence analysis of the Krylov subspace methods is more complex, because not only the matrix spectrum affects the convergence. What approach using depends on the selected Krylov subspace method, on the type of preconditioner, on the particular problem at hand and so on. In our case we will

use the version (3.9) for the block triangular and block ILU preconditioners and the system (3.10) for the HSS preconditioner.

In terms of Krylov subspace methods for solving our problem, the CG is not suitable, and the main options are: the MINRES, if the matrix is symmetric; the GMRES, which is the most popular method for generic non-singular matrices; and the BiCGStab, which is designed for generic non-singular matrices and often converges faster with respect to the GMRES, but which may present an irregular convergence and some types of breakdowns. For a detailed survey of these methods we refer to [32, 58].

For our purpose we do not make use of MINRES because it requires the constraint of a symmetric and positive definite preconditioner, which is quite restrictive for saddle point linear systems. In the code we have included the BiCGStab method and the restarted GMRES and FGMRES. In any case, for the scopes of this thesis we restrict our numerical analysis to the GMRES. For the HSS preconditioner that is implemented with a CG inner cycle, the FGMRES is used.

For saddle point problems general-purpose algebraic preconditioners often perform poorly, because of the indefiniteness and lack of diagonal dominance of the matrix of the system, see [20]. We have developed three preconditioners tailored for our problem, but, for reasons that we have already pointed out, only the saddle point algebraic block partitioning of the matrix has been used to design them. Indeed the saddle point nature of the problem leads the way to many possible choices in terms of preconditioning, see [20]. Regarding this, we stress that most of proposals in literature concern the case $\widetilde{\mathsf{T}} \neq 0$ and it is not always easy to accommodate the term $\widetilde{\mathsf{T}} \neq 0$ in preconditioners for standard saddle point problems.

An important aspect of solving iteratively a saddle point linear system is that the storage of the whole matrix of the system is not necessary. Indeed in an iterative solver algorithm only the matrix-vector product is required, and this can be performed by storing only the single block matrices, saving a memory amount corresponding to the non-zeros of $\widetilde{\mathsf{B}}^{\mathsf{T}}$. So, depending on the type of linear system that we choose, the matrix-vector product is performed as

$$\begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathsf{M}_c\mathbf{x} + \widetilde{\mathsf{B}}^{\mathsf{T}}\mathbf{y} \\ \widetilde{\mathsf{B}}\mathbf{x} - \widetilde{\mathsf{T}}\mathbf{y} \end{bmatrix} \tag{3.11}$$

or as

$$\begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ -\widetilde{\mathsf{B}} & \widetilde{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathsf{M}_c\mathbf{x} + \widetilde{\mathsf{B}}^{\mathsf{T}}\mathbf{y} \\ -\widetilde{\mathsf{B}}\mathbf{x} + \widetilde{\mathsf{T}}\mathbf{y} \end{bmatrix}. \tag{3.12}$$

Finally consider that also the construction of the chosen preconditioners is based on the block structure of the matrix. This is convenient in terms of memory and time.

3.4 Preconditioning techniques

In this section we present the three preconditioners that have been implemented in the code to accelerate the convergence of iterative methods, in particular of GMRES. We start presenting a block triangular preconditioner, then we describe the block ILU case, which is a similar technique, and then we describe the HSS

preconditioner, which is a more recent technique based on the Hermitian and skew-Hermitian splitting of the matrix of the system. A survey on these techniques and of several others for saddle point problems can be found in [20].

3.4.1 Block Triangular preconditioner

Let us consider the following preconditioner

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

where $\mathsf{S} = -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\mathsf{M}_c^{-1}\widetilde{\mathsf{B}}^{\mathsf{T}})$ is the Schur Complement. For the matrix of the system $\mathcal{A}$ the following factorization holds:

$$\mathcal{A} = \begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} \mathsf{I} & 0 \\ \widetilde{\mathsf{B}}\mathsf{M}_c^{-1} & \mathsf{I} \end{bmatrix} \begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ 0 & \mathsf{S} \end{bmatrix}. \tag{3.13}$$

It is straightforward noting that, thanks to this factorization, we have the optimal spectral result $\sigma(\mathcal{A}\mathcal{P}^{-1}) = \{1\}$. Moreover the preconditioned matrix has a minimal polynomial of degree 2, which in the Krylov subspace theory (see [58]) means convergence in at most two steps.

Obviously in practice suitable approximations $\widehat{\mathsf{M}}_c$ and $\widehat{\mathsf{S}}$ of M_c and S have to be used because otherwise inverting the preconditioner would be unfeasible. Therefore, the typical form of a block triangular preconditioner is:

$$\mathcal{P} = \begin{bmatrix} \widehat{\mathsf{M}}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ 0 & \widehat{\mathsf{S}} \end{bmatrix}, \tag{3.14}$$

where $\widehat{\mathsf{M}}_c \simeq \mathsf{M}_c$ and $\widehat{\mathsf{S}} \simeq \mathsf{S}$.

Finding good approximations $\widehat{\mathsf{M}}_c$ and $\widehat{\mathsf{S}}$ is the most important point and it is strongly problem dependent. There are several proposals in literature, especially for standard Finite Elements and Mixed Finite Elements (see [20]); however, as already mentioned, regarding this topic there are no proposals available in literature for mimetic finite differences. The matrix M_c represents a standard L^2 inner product with a tensorial weight K^{-1} and a typical approach in such cases is diagonal lumping. In particular we consider two approximations: the classical diagonal lumping given by

$$\widehat{\mathsf{M}}_c = \mathtt{diag}([\overline{m_c}]_{ii}), \qquad [\overline{m_c}]_{ii} = \sum_j [m_c]_{ij} \tag{3.15}$$

and the the diagonal part of M_c given by

$$\widehat{\mathsf{M}}_c = \mathtt{diag}([m_c]_{ii}). \tag{3.16}$$

Regarding the approximation of S, the situation is more delicate. Again there are several proposals in literature for Finite Elements. For instance, for Stokes problems, where the block (1,1) represents a second order diffusion operator, in that

case the Schur Complement is spectrally equivalent to a mass matrix, see [55]. For what concerns us, we have adopted the simple approximation based on $\widehat{\mathsf{M}}_c$ given by

$$\widehat{\mathsf{S}} = -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\widehat{\mathsf{M}}_c^{-1}\widetilde{\mathsf{B}}^{\mathsf{T}}). \tag{3.17}$$

Note that, being $\widehat{\mathsf{M}}_c$ a diagonal matrix, its inverse $\widehat{\mathsf{M}}_c^{-1}$ is readily available and so $\widehat{\mathsf{S}}$ is a sparse matrix that can be computed and stored explicitly. To summarize, the block triangular preconditioners we consider are of two types: one based on the mass lumping of M_c and one based on the diagonal part of M_c.

Solving $\mathcal{P}\mathbf{z} = \mathbf{r}$ can be implemented basing on the factorization

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

so that, fixed an iteration and 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:

$$\boxed{\begin{aligned} -\widehat{\mathsf{S}}\mathbf{y} &= \mathbf{r}_2 \\ \boldsymbol{\gamma} &= \mathbf{r}_1 + \widetilde{\mathsf{B}}^{\mathsf{T}}\mathbf{y} \\ \mathbf{z}_1 &= \widehat{\mathsf{M}}_c^{-1}\boldsymbol{\gamma} \\ \mathbf{z}_2 &= -\mathbf{y}. \end{aligned}} \tag{3.18}$$

Matrix-vector products and the resolution of a linear system with matrix $-\widehat{\mathsf{S}}$, which is the most demanding operation, are required for the inversion of the preconditioner $\mathcal{P}$, whose explicit construction is not necessary. Let us note that $-\widehat{\mathsf{S}}$ is sparse, symmetric and positive definite and so we can exploit some efficient methods like a sparse Cholesky factorization or the CG. Our code constructs the matrix $-\widehat{\mathsf{S}}$ and solves the system with a sparse Cholesky factorization [29]. More precisely, we pre-factorize the matrix $-\widehat{\mathsf{S}}$ and we solve a couple of upper and lower triangular systems for every iteration of GMRES.

Another possible approach is solving the linear system involving $-\widehat{\mathsf{S}}$ with the CG, giving raise to an "inner cycle", and using the FGMRES as "outer" solver to take into account the preconditioner modifications. This technique is more effective when $-\widehat{\mathsf{S}}$ is very large (i.e. in case of very refined grids) and allows memory savings, because $-\widehat{\mathsf{S}}$ must be not stored explicitly. However the efficacy of this approach depends on the conditioning of $-\widehat{\mathsf{S}}$, which is sensitive on both the mesh size and the problem parameters. Unless an effective preconditioner of $-\widehat{\mathsf{S}}$ is available, the calibration of the maximum number of iterations and the tolerance for the CG will depend on the mesh size and the parameters. We have tested such a scheme with a simple diagonal preconditioner for $-\widehat{\mathsf{S}}$ and it is significantly less efficient than the scheme with the Cholesky pre-factorization already for medium size problems. Finding a good preconditioner for $-\widehat{\mathsf{S}}$, to handle huge size problems where the factorization of $-\widehat{\mathsf{S}}$ is inconvenient, is still an open issue.

Remark 3.1. *Note that from the factorization (3.13) it follows that $\mathcal{A}\mathcal{P}^{-1}$ is non symmetric. This turns out in a complex convergence analysis in which not only the spectrum of the preconditioned matrix affects the rate of convergence of the GMRES.*

In any case, if good approximations of M_c *and* S *are available, this aspect may not compromise a rapid convergence, as we will see in Chapter 5.*

Remark 3.2. *Another possible preconditioner for a saddle point system is a block diagonal preconditioner with* $\widehat{\mathsf{M}}_c$ *and* $\widehat{\mathsf{S}}$ *as the (1,1) and (2,2) diagonal blocks, respectively. Indeed the matrix* $\mathcal{A}$ *and the block diagonal preconditioner with the exact blocks* M_c *and* S *are similar. This preconditioning option in general is less performing of the block triangular one in which the coupling between velocity and pressure is taken into account because of the non zero extra-diagonal block* $\widetilde{\mathsf{B}}^{\mathsf{T}}$*, with the additional cost of only one matrix-vector product, see (3.18).*

3.4.2 Block ILU preconditioner

Now we introduce a preconditioner based on an inexact block LU factorization. For the matrix of the system $\mathcal{A}$ the following block LU factorization holds:

$$\mathcal{A} = \begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} \mathsf{M}_c & 0 \\ \widetilde{\mathsf{B}} & \mathsf{S} \end{bmatrix} \begin{bmatrix} \mathsf{I} & \mathsf{M}_c^{-1}\widetilde{\mathsf{B}}^{\mathsf{T}} \\ 0 & \mathsf{I} \end{bmatrix}, \tag{3.19}$$

where $\mathsf{S} = -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\mathsf{M}_c^{-1}\widetilde{\mathsf{B}}^{\mathsf{T}})$ is the Schur Complement. The idea is again to substitute M_c and S with proper approximations $\widehat{\mathsf{M}}_c$ and $\widehat{\mathsf{S}}$, so that the preconditioner is now

$$\mathcal{P} = \begin{bmatrix} \widehat{\mathsf{M}}_c & 0 \\ \widetilde{\mathsf{B}} & -\widehat{\mathsf{S}} \end{bmatrix} \begin{bmatrix} \mathsf{I} & \widehat{\mathsf{M}}_c^{-1}\widetilde{\mathsf{B}}^{\mathsf{T}} \\ 0 & \mathsf{I} \end{bmatrix}. \tag{3.20}$$

Multiplying the factors we get

$$\mathcal{P} = \begin{bmatrix} \widehat{\mathsf{M}}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ \widetilde{\mathsf{B}} & -\widehat{\widetilde{\mathsf{T}}} \end{bmatrix}, \qquad \widehat{\widetilde{\mathsf{T}}} = -(\widetilde{\mathsf{B}}\widehat{\mathsf{M}}_c\widetilde{\mathsf{B}}^{\mathsf{T}} + \widehat{\mathsf{S}}). \tag{3.21}$$

We consider the same approximations of $\widehat{\mathsf{M}}_c$ and $\widehat{\mathsf{S}}$ introduced for the block triangular preconditioner: the mass lumping of M_c (see (3.15)) and the diagonal part of M_c (see (3.16)) and the corresponding inexact Schur Complement $\widehat{\mathsf{S}} = -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\widehat{\mathsf{M}}_c^{-1}\widetilde{\mathsf{B}}^{\mathsf{T}})$. Therefore we have two versions of the ILU preconditioner: one based on the lumping and the other on the diagonal of M_c. Note that with this simple approximation of the Schur Complement the formula (3.21) reduces to

$$\mathcal{P} = \begin{bmatrix} \widehat{\mathsf{M}}_c & \widetilde{\mathsf{B}}^{\mathsf{T}} \\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}} \end{bmatrix}, \tag{3.22}$$

because $\widehat{\widetilde{\mathsf{T}}} = \widetilde{\mathsf{T}}$. The preconditioner in (3.22) is the classical form of an indefinite preconditioner, more precisely it falls under a wide class of preconditioners called *Constraint and Indefinite preconditioners*, see [20] for more details. This technique is based on the principle that the preconditioner matrix should have the same block structure of the original saddle point matrix. Therefore the saddle point matrix is preconditioned by another saddle point matrix that can be inverted in an easier way.

It is easy to recover from (3.20) that, for a given residual $\mathbf{r} = [\mathbf{r}_1, \mathbf{r}_2]^\mathsf{T}$, computing the preconditioned residual $\mathbf{z} = [\mathbf{z}_1, \mathbf{z}_2]^\mathsf{T} = \mathcal{P}^{-1}\mathbf{r}$ requires the following steps:

$$\boxed{\begin{aligned} \mathbf{y}_1 &= \widehat{\mathsf{M}}_c^{-1}\mathbf{r}_1 \\ -\widehat{\mathsf{S}}\mathbf{y}_2 &= \widetilde{\mathsf{B}}\mathbf{y}_1 - \mathbf{r}_2 \\ \mathbf{z}_1 &= \mathbf{y}_1 - \widehat{\mathsf{M}}_c^{-1}\widetilde{\mathsf{B}}^\mathsf{T}\mathbf{y}_2 \\ \mathbf{z}_2 &= \mathbf{y}_2. \end{aligned}} \tag{3.23}$$

The required operations are basically the same as in the block triangular preconditioner with an additional multiplication by $\widehat{\mathsf{M}}_c^{-1}$ and $\widetilde{\mathsf{B}}$. As already pointed out for the block triangular preconditioner, the matrix $-\widehat{\mathsf{S}}$ is sparse, symmetric and positive definite. The code solves the linear system with a sparse Cholesky pre-factorization, so that a couple of upper and lower triangular systems is solved for every GMRES iteration.

Remark 3.3. *The block ILU preconditioner based on the diagonal part of M_c is the preconditioning scheme for fractured porous media corresponding to the well-known SIMPLE preconditioning (see [32]) in computational fluid dynamics.*

3.4.3 Hermitian and skew-Hermitian preconditioner

This technique has been introduced firstly as a linear stationary iterative method for generalized saddle point linear systems, then it has been applied and studied as a preconditioner to accelerate Krylov subspace methods for generic non-singular matrices, the GMRES in particular. We start presenting the HSS iterative scheme and then we deal with the HSS preconditioning.

This technique is based on the Hermitian and skew-Hermitian splitting of the matrix of the system $\widehat{\mathcal{A}}$, which is the saddle point matrix in unsymmetric and positive definite form, see (3.10). If $\mathsf{H} = 1/2(\mathsf{M}_c + \mathsf{M}_c^\mathsf{T})$ is the hermitian part of M_c and $\mathsf{K} = 1/2(\mathsf{M}_c - \mathsf{M}_c^\mathsf{T})$ is the skew-Hermitian part of M_c, we have the following Hermitian/skew-Hermitian splitting of $\widehat{\mathcal{A}}$:

$$\widehat{\mathcal{A}} = \begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^\mathsf{T} \\ -\widetilde{\mathsf{B}} & \widetilde{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} \mathsf{H} & 0 \\ 0 & \widetilde{\mathsf{T}} \end{bmatrix} + \begin{bmatrix} \mathsf{K} & \widetilde{\mathsf{B}}^\mathsf{T} \\ -\widetilde{\mathsf{B}} & 0 \end{bmatrix} = \mathcal{H} + \mathcal{K}, \tag{3.24}$$

where $\mathcal{H}$ is symmetric and semi-positive definite. The HSS technique requires only the positive semi-definiteness of the M_c; in our case M_c is symmetric and positive definite and so we have $\mathsf{H} = \mathsf{M}_c$ and $\mathsf{K} = \mathbf{0}$.

Let us consider a parameter $\alpha > 0$ and the following splitting of $\widehat{\mathcal{A}}$:

$$\widehat{\mathcal{A}} = (\mathcal{H} + \alpha\mathsf{I}) - (\alpha\mathsf{I} - \mathcal{K}), \quad \widehat{\mathcal{A}} = (\mathcal{K} + \alpha\mathsf{I}) - (\alpha\mathsf{I} - \mathcal{H}),$$

where I indicates the identity matrix of proper size. The HSS iterative method proceeds by alternating these two splits of $\widehat{\mathcal{A}}$: given the initial guess $\mathbf{x}_0$, the HSS iteration looks for the sequence $\{\mathbf{x}_k\}$ such that

$$\begin{cases} (\mathcal{H} + \alpha\mathsf{I})\mathbf{x}_{k+1/2} = (\alpha\mathsf{I} - \mathcal{K})\mathbf{x}_k + \widehat{\mathbf{b}} \\ (\mathcal{K} + \alpha\mathsf{I})\mathbf{x}_{k+1} = (\alpha\mathsf{I} - \mathcal{H})\mathbf{x}_{k+1/2} + \widehat{\mathbf{b}}. \end{cases} \tag{3.25}$$

Note that $(\mathcal{H} + \alpha \mathsf{I})$ is symmetric and positive definite and $(\mathcal{K} + \alpha \mathsf{I})$ is non singular.

Erasing the intermediate unknown $\mathbf{x}_{k+1/2}$, we can rewrite the two steps scheme as a fixed point iteration

$$\mathbf{x}_{k+1} = \mathcal{T}_\alpha \mathbf{x}_k + \mathbf{c}, \tag{3.26}$$

where

$$\mathcal{T}_\alpha = (\mathcal{K} + \alpha \mathsf{I})^{-1}(\alpha \mathsf{I} - \mathcal{H})(\mathcal{H} + \alpha \mathsf{I})^{-1}(\alpha \mathsf{I} - \mathcal{K})$$

and

$$\mathbf{c} = (\mathcal{K} + \alpha \mathsf{I})^{-1}[\mathsf{I} + (\alpha \mathsf{I} - \mathcal{H})(\mathcal{H} + \alpha \mathsf{I})^{-1}]\widehat{\mathbf{b}}.$$

This is the classical form of a linear stationary iterative scheme with iteration matrix $\mathcal{T}_\alpha$ (see [56]) and, for arbitrary initial guess $\mathbf{x}_0$ and right-hand side $\widehat{\mathbf{b}}$, it converges to the solution $\mathbf{x} = \widehat{\mathcal{A}}^{-1}\widehat{\mathbf{b}}$ if and only if $\rho(\mathcal{T}_\alpha) < 1$, where $\rho(\mathcal{T}_\alpha)$ indicates the spectral radius of the iteration matrix. It has been shown (see [19]) that the HSS iteration scheme converges for all $\alpha > 0$ if H is positive definite, $\widetilde{\mathsf{B}}$ is full rank and $\widetilde{\mathsf{T}}$ is symmetric semi-positive definite. The $\rho(\mathcal{T}_\alpha)$ represents the asymptotic rate of convergence of the HSS iterative method and an optimal value of α that minimizes an upper bound for $\rho(\mathcal{T}_\alpha)$ can computed as

$$\alpha_{opt} = \sqrt{\lambda_{\min}(\mathcal{H})\lambda_{\max}(\mathcal{H})}, \tag{3.27}$$

provided that $\mathcal{H}$ is positive definite, see [10] on this topic.

To recover a preconditioner for the linear system (3.10), we observe that there is a unique splitting $\widehat{\mathcal{A}} = \mathcal{P} - \mathcal{Q}$, with $\mathcal{P}$ non singular, such that $\mathcal{T}_\alpha = \mathsf{I} - \mathcal{P}^{-1}\widehat{\mathcal{A}}$, i.e. $\mathcal{T}_\alpha$ is the iteration matrix induced by that splitting. Basing on this observation, the iteration scheme (3.26) can be rewritten as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathcal{P}^{-1}\mathbf{r}_k, \quad \mathbf{r}_k = \widehat{\mathbf{b}} - \widehat{\mathcal{A}}\mathbf{x}_k,$$

where the preconditioner matrix $\mathcal{P}$ is given by

$$\mathcal{P} = \mathcal{P}_\alpha = \frac{1}{2\alpha}(\mathcal{H} + \alpha \mathsf{I})(\mathcal{K} + \alpha \mathsf{I}). \tag{3.28}$$

Note that, from $\mathcal{T}_\alpha = \mathsf{I} - \mathcal{P}_\alpha^{-1}\widehat{\mathcal{A}}$ and $\rho(\mathcal{T}_\alpha) < 1$, it follows that $\sigma(\mathcal{P}_\alpha^{-1}\widehat{\mathcal{A}}) \subset \{z \in \mathbb{C} : |z - 1| < 1\}$. Thanks to this property, we have that the spectra of $\mathcal{P}_\alpha^{-1}\widehat{\mathcal{A}}$ is fully contained in $\mathbb{C}_+$, so that $\mathcal{P}_\alpha^{-1}\widehat{\mathcal{A}}$ is positive definite.

The linear stationary method can be used to solve a saddle point linear system, but, also employing the α_{opt} given by (3.27), a slow convergence is observed [19]. An interesting alternative, which is important from our point of view, is using a suitable Krylov subspace method, e.g. the GMRES, with $\mathcal{P}_\alpha$ as a preconditioner. Clearly the rate of convergence of the GMRES depends on the coefficient α and we can observe that, if $\rho(\mathcal{T}_\alpha)$ is small enough, the eigenvalues of $\mathcal{P}_\alpha^{-1}\widehat{\mathcal{A}}$ tends to form a cluster around 1. However the optimal value of α given by (3.27) is valid only if $\mathcal{H}$ is positive definite, which in general is quite restrictive and in our case is not true, because $\widetilde{\mathsf{T}}$ is only semi-positive definite. Moreover, the optimal value of α for minimizing the iterations of the preconditioned GMRES may be different from the one that optimizes the iterations of the HSS stationary scheme. Regarding this

aspect we have the following result (see [42]): as $\alpha \to 0$ the eigenvalues of $\mathcal{P}_\alpha^{-1}\widehat{\mathcal{A}}$ are real and fall within two small intervals $(0, \epsilon_1)$ and $(2-\epsilon_2, 2)$ with $\epsilon_1, \epsilon_2 > 0$ such that $\epsilon_1, \epsilon_2 \to 0$ as $\alpha \to 0$. So a coefficient α small enough ensures a favourable clustering result of the eigenvalues of the preconditioned matrix. Moreover, a Fourier analysis of the HSS preconditioning has been conducted for the saddle point formulation of a Poisson problem and it shows that a sufficiently small α turns out in h-independent convergence of GMRES [18]. However h-independence is not guaranteed in general, indeed it is not observed in Stokes problem [19].

Now we deal with the steps required to solve $\mathcal{P}_\alpha \mathbf{z} = \mathbf{r}$. We present them taking into account that for our problem M_c is also symmetric and so $\mathsf{H} = \mathsf{M}_c$ and $\mathsf{K} = \mathbf{0}$. As a preconditioner we consider $\mathcal{P}_\alpha = (\mathcal{H} + \alpha\mathsf{I})(\mathcal{K} + \alpha\mathsf{I})$ without the normalization factor $1/2\alpha$ that has no effect on the preconditioned system. Fixed an iteration and given the residual $\mathbf{r}$, to compute the preconditioned residual $\mathbf{z}$ we have to solve

$$\begin{cases} (\mathcal{H} + \alpha\mathsf{I})\boldsymbol{\omega} = \mathbf{r} \\ (\mathcal{K} + \alpha\mathsf{I})\mathbf{z} = \boldsymbol{\omega}. \end{cases}$$

Splitting the vectors as $\mathbf{r} = [\mathbf{r}_1, \mathbf{r}_2]^\mathsf{T}$, $\boldsymbol{\omega} = [\boldsymbol{\omega}_1, \boldsymbol{\omega}_2]^\mathsf{T}$ and $\mathbf{z} = [\mathbf{z}_1, \mathbf{z}_2]^\mathsf{T}$, the steps are the following:

$$\boxed{\begin{aligned} (\mathsf{M}_c + \alpha\mathsf{I})\boldsymbol{\omega}_1 &= \mathbf{r}_1 \\ (\widetilde{\mathsf{T}} + \alpha\mathsf{I})\boldsymbol{\omega}_2 &= \mathbf{r}_2 \\ (\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T} + \alpha^2\mathsf{I})\mathbf{z}_2 &= \widetilde{\mathsf{B}}\boldsymbol{\omega}_1 + \alpha\boldsymbol{\omega}_2 \\ \mathbf{z}_1 &= \alpha^{-1}(\boldsymbol{\omega}_1 - \widetilde{\mathsf{B}}^\mathsf{T}\mathbf{z}_2). \end{aligned}} \tag{3.29}$$

The main operations consist of solving three linear systems. Starting from the one involving $\mathsf{M}_c + \alpha\mathsf{I}$, the matrix is symmetric and positive definite and the term $\alpha\mathsf{I}$ improves the condition number of M_c. This linear system is solved with CG, preconditioned with the diagonal part of $\mathsf{M}_c + \alpha\mathsf{I}$, and not with a sparse Cholesky pre-factorization. The factorization of this matrix is too expensive and solving this system with a direct method turns out in a much slower scheme. This is not surprising because M_c is the largest block matrix of the system: the number of facets of the mesh is much greater than the number of cells.

For the second linear system we stress that, because of the definition (2.36) of the block $\widetilde{\mathsf{T}}$, we have

$$\widetilde{\mathsf{T}} + \alpha\mathsf{I} = \begin{bmatrix} \alpha\mathsf{I} & 0 \\ 0 & \mathsf{T} + \alpha\mathsf{I} \end{bmatrix}.$$

Therefore, only the system involving $\mathsf{T} + \alpha\mathsf{I}$ must be actually solved. The matrix $\mathsf{T} + \alpha\mathsf{I}$ is symmetric and positive definite and this linear system is solved in the code through a sparse Cholesky pre-factorization. Note that $\mathsf{T} + \alpha\mathsf{I}$ arises from a 2D problem and its dimension (which coincides with the number of fracture facets) is much lower with respect to the other block matrices of the system. So in this case using a direct solver is surely the best approach.

Regarding the third linear system involving $\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T} + \alpha^2\mathsf{I}$, the matrix is again symmetric, positive definite and sparse. It is solved in the code with a sparse Cholesky pre-factorization.

3.5 Spectral analysis of the governing matrix

In this section we study the spectral properties of the matrix of the system, for which we prove an estimate of the condition number. To this purpose, we start describing in details the matrices involved in the algebraic formulation of the problem. Next, we present some preliminary results and assumptions on the mesh and the transmissibility matrix. Finally, we state and prove the estimate of the condition number.

We assume, as in Chapter 1 and 2, a single fracture which may be also partially or fully immersed in the domain. In addition, we assume only Dirichlet boundary conditions for the bulk problem.

For the notation of the mesh, which is important in this section, we refer to Section 2.2 of Chapter 2. We also enrich the notation introducing the following sets of mesh objects: $\Omega_h(\mathsf{f})$ collecting the bulk cells separated by the facet f; $\mathscr{F}_h^\Gamma(\hat{\mathsf{f}}) = \{\mathsf{f}^+, \mathsf{f}^-\}$ denoting the decoupled facets of the bulk problem associated with the fracture facet $\hat{\mathsf{f}}$; $\Gamma_h(\mathsf{f})$ for the fracture facet $\hat{\mathsf{f}}$ corresponding to the bulk facet f. Note that, if $\mathsf{f} \in \mathscr{F}_h^\Gamma$, then $\Omega_h(\mathsf{f})$ identifies only one cell: $\mathsf{P}^+ \in \Omega_h^+$ if $\mathsf{f} \in \mathscr{F}_{h,+}^\Gamma$, $\mathsf{P}^- \in \Omega_h^-$ if $\mathsf{f} \in \mathscr{F}_{h,-}^\Gamma$. Observe also that $\Gamma_h(\mathsf{f}) \neq \emptyset$ if and only if $\mathsf{f} \in \mathscr{F}_h^\Gamma$. Moreover, we define the maximum number of facets of a cell as

$$N^* = \max_{\mathsf{P} \in \Omega_h} |\mathscr{F}_h(\mathsf{P})|,$$

where $|\cdot|$ indicates the cardinality.

3.5.1 Characterization of the governing matrix

The linear system arising from the numerical discretization is

$$\begin{bmatrix} \mathsf{M}_c & \mathsf{B}^\mathsf{T} & \mathsf{C}^\mathsf{T} \\ \mathsf{B} & 0 & 0 \\ \mathsf{C} & 0 & -\mathsf{T} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \\ \mathbf{p}_\Gamma \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ \mathbf{h} \\ \mathbf{h}_\Gamma \end{bmatrix}, \tag{3.30}$$

see Section 2.4 of Chapter 2 for more details on the derivation of the linear system.

The matrix M_c represents the discrete bilinear form

$$a_h(\mathbf{u}_h, \mathbf{v}_h) = [\mathbf{u}_h, \mathbf{v}_h]_{W_h} + [\mathbf{u}_h, \mathbf{v}_h]_\Gamma, \tag{3.31}$$

where $[\cdot,\cdot]_{W_h}$ is the weighted mimetic inner product and $[\cdot,\cdot]_\Gamma$ takes into account the coupling conditions involving the velocity jump and average across the fracture facets. In particular we have have

$$a_h(\mathbf{u}_h, \mathbf{v}_h) = \mathbf{u}^\mathsf{T}\mathsf{M}_c\mathbf{v} = \mathbf{u}^\mathsf{T}\mathsf{M}\mathbf{v} + \sum_{\hat{\mathsf{f}} \in \Gamma_h} \eta_{\Gamma,\hat{\mathsf{f}}} |\hat{\mathsf{f}}| \{\mathbf{u}_h\}_{\hat{\mathsf{f}}} \{\mathbf{v}_h\}_{\hat{\mathsf{f}}} + \sum_{\hat{\mathsf{f}} \in \Gamma_h} \xi_0 \eta_{\Gamma,\hat{\mathsf{f}}} |\hat{\mathsf{f}}| [\![\mathbf{u}_h]\!]_{\hat{\mathsf{f}}} [\![\mathbf{v}_h]\!]_{\hat{\mathsf{f}}}, \tag{3.32}$$

where M is the inner product matrix, which is constructed by assembling local contributions M_P given by equations (2.21)-(2.23). From (3.32) we can recover the

following characterization for the elements of the matrix M_c:

$$[\mathsf{M}_c]_{ij} = \begin{cases} \mathsf{M}_{ij} & \text{if } \mathsf{f}_i, \mathsf{f}_j \notin \mathscr{F}_h^\Gamma \\ \mathsf{M}_{ij} + \eta_{\Gamma,\hat{\mathsf{f}}} \dfrac{|\hat{\mathsf{f}}|}{4} + \eta_{\Gamma,\hat{\mathsf{f}}} \xi_0 |\hat{\mathsf{f}}| & \text{if } \mathsf{f}_i = \mathsf{f}_i^+ \in \mathscr{F}_{h,+}^\Gamma \text{ and } \mathsf{f}_j = \mathsf{f}_i \\ \mathsf{M}_{ij} + \eta_{\Gamma,\hat{\mathsf{f}}} \dfrac{|\hat{\mathsf{f}}|}{4} - \eta_{\Gamma,\hat{\mathsf{f}}} \xi_0 |\hat{\mathsf{f}}| & \text{if } \mathsf{f}_i = \mathsf{f}_i^+ \in \mathscr{F}_{h,+}^\Gamma \text{ and } \mathsf{f}_j = (\mathsf{f}_i)^- \\ \mathsf{M}_{ij} + \eta_{\Gamma,\hat{\mathsf{f}}} \dfrac{|\hat{\mathsf{f}}|}{4} + \eta_{\Gamma,\hat{\mathsf{f}}} \xi_0 |\hat{\mathsf{f}}| & \text{if } \mathsf{f}_i = \mathsf{f}_i^- \in \mathscr{F}_{h,-}^\Gamma \text{ and } \mathsf{f}_j = \mathsf{f}_i \\ \mathsf{M}_{ij} + \eta_{\Gamma,\hat{\mathsf{f}}} \dfrac{|\hat{\mathsf{f}}|}{4} - \eta_{\Gamma,\hat{\mathsf{f}}} \xi_0 |\hat{\mathsf{f}}| & \text{if } \mathsf{f}_i = \mathsf{f}_i^- \in \mathscr{F}_{h,-}^\Gamma \text{ and } \mathsf{f}_j = (\mathsf{f}_i)^+, \end{cases} \tag{3.33}$$

for $i = 1, \ldots, N_\mathsf{f}$ and $j = 1, \ldots, N_\mathsf{f}$. Note that $(\mathsf{f}_i)^-$ indicates the minus facet corresponding to $\mathsf{f}_i = \mathsf{f}_i^+$, $(\mathsf{f}_i)^+$ the plus facet corresponding to $\mathsf{f}_i = \mathsf{f}_i^-$ and we have $\hat{\mathsf{f}} = \Gamma_h(\mathsf{f}_i)$.

Next, we characterize the elements of the matrix B which represents the discrete bilinear form $-[\mathrm{div}_h \mathbf{u}_h, q_h]_{Q_h}$. In particular, from the discrete divergence defined in (2.3) and the inner product in Q_h defined in (2.4), we have

$$-[\mathrm{div}_h \mathbf{u}_h, q_h]_{Q_h} = \sum_{\mathsf{P} \in \Omega_h} |\mathsf{P}| q_\mathsf{P} \frac{1}{|\mathsf{P}|} \sum_{\mathsf{f} \in \mathscr{F}_h(\mathsf{P})} \alpha_{\mathsf{P},\mathsf{f}} |\mathsf{f}| u_\mathsf{P} = \sum_{\mathsf{P} \in \Omega_h} q_\mathsf{P} \sum_{\mathsf{f} \in \mathscr{F}_h(\mathsf{P})} \alpha_{\mathsf{P},\mathsf{f}} |\mathsf{f}| u_\mathsf{P} = \mathbf{q}^\mathsf{T} \mathsf{B} \mathbf{u},$$

where

$$\mathsf{B}_{ij} = \begin{cases} -\alpha_{\mathsf{P}_i,\mathsf{f}_j} |\mathsf{f}_j| & \text{if } \mathsf{f}_j \in \mathscr{F}_h(\mathsf{P}_i) \\ 0 & \text{otherwise,} \end{cases} \tag{3.34}$$

for $i = 1, \ldots, N_\mathsf{P}$ and $j = 1, \ldots, N_\mathsf{f}$. As M the matrix B can be constructed by assembling local contributions $\mathsf{B}_\mathsf{P} \in \mathbb{R}^{1 \times N_\mathsf{P}^\mathscr{F}}$ given by

$$\mathsf{B}_\mathsf{P} = -\begin{bmatrix} \alpha_{\mathsf{P},\mathsf{f}_1} |\mathsf{f}_1| & \ldots & \alpha_{\mathsf{P},\mathsf{f}_i} |\mathsf{f}_i| & \ldots & \alpha_{\mathsf{P},\mathsf{f}_{N_\mathsf{P}^\mathscr{F}}} |\mathsf{f}_{N_\mathsf{P}^\mathscr{F}}| \end{bmatrix}, \tag{3.35}$$

where $\mathscr{F}_h(\mathsf{P}) = \{\mathsf{f}_1, \ldots, \mathsf{f}_{N_\mathsf{P}^\mathscr{F}}\}$.

The matrix C represents the discrete bilinear form $\sum_{\hat{\mathsf{f}} \in \Gamma_h} |\hat{\mathsf{f}}| p_{\Gamma,\hat{\mathsf{f}}} [\![\mathbf{v}_h]\!]_{\hat{\mathsf{f}}}$ and its elements has the following form

$$\mathsf{C}_{ij} = \begin{cases} \alpha_{\mathsf{P},\mathsf{f}_j} |\hat{\mathsf{f}}_i| & \text{if } \mathsf{f}_j \in \mathscr{F}_h^\Gamma(\hat{\mathsf{f}}_i), \text{ where } \{\mathsf{P}\} = \Omega_h(\mathsf{f}_j) \\ 0 & \text{otherwise,} \end{cases} \tag{3.36}$$

for $i = 1, \ldots, N_\Gamma$ and $j = 1, \ldots, N_\mathsf{f}$.

Finally, recall that the linear system can be reformulated in a more standard saddle point form (see equations (2.36) and (2.37)) with the following matrix

$$\mathcal{A} = \begin{bmatrix} \mathsf{M}_c & \widetilde{\mathsf{B}}^\mathsf{T} \\ \widetilde{\mathsf{B}} & -\widetilde{\mathsf{T}} \end{bmatrix}. \tag{3.37}$$

3.5.2 Preliminary lemmas and mesh assumption

Here we describe some preliminary results and an assumption on the mesh that will be used in the spectral analysis of $\mathcal{A}$.

Extension of the stability condition

The mimetic inner product $[\cdot,\cdot]_{W_h}$ fulfils a consistency condition (see (2.10)), which ensures that it reproduces exactly the continuous counterpart under certain conditions, and a stability condition (see (2.9)) that is important for the well-posedness of the discrete problem. The stability condition can be expressed as: there exist two positive constants $\sigma_*, \sigma^* > 0$, independent from the mesh size h, such that

$$\sigma_* |||\mathbf{v}_h|||_{W_h} \leq [\mathbf{v}_h, \mathbf{v}_h]_{W_h} \leq \sigma^* |||\mathbf{v}_h|||_{W_h} \quad \forall\ \mathbf{v}_h \in W_h, \tag{3.38}$$

where

$$|||\mathbf{v}_h|||^2_{W_h} = \sum_{\mathsf{P}\in\Omega_h} |||\mathbf{v}_\mathsf{P}|||^2_\mathsf{P} = \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} v_\mathsf{f}^2.$$

We want to extend the stability condition given by (3.38) to the discrete bilinear form $a_h(\cdot,\cdot)$ defined in (3.31). We equip the space of the discrete velocities W_h with a different norm that takes into account the jump and the average of the velocity across the fracture:

$$||\mathbf{v}_h||^2_{W_h} = \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} v_\mathsf{f}^2 + \sum_{\hat{\mathsf{f}}\in\Gamma_h} |\hat{\mathsf{f}}|(\llbracket \mathbf{v}_h \rrbracket^2_{\hat{\mathsf{f}}} + \{\mathbf{v}_h\}^2_{\hat{\mathsf{f}}}).$$

Let us introduce the following quantities $\eta_* = \min_{\mathbf{x}\in\Gamma} \eta(\mathbf{x})$ and $\eta^* = \max_{\mathbf{x}\in\Gamma} \eta(\mathbf{x})$. The result for $a_h(\cdot,\cdot)$ is the following.

Lemma 3.2. *Provided that $\xi_0 > 0$, the discrete bilinear form $a_h(\cdot,\cdot)$ is stable with respect to $||\cdot||_{W_h}$ norm. More precisely,*

$$C_* ||\mathbf{v}_h||^2_{W_h} \leq a_h(\mathbf{v}_h, \mathbf{v}_h) \leq C^* ||\mathbf{v}_h||^2_{W_h} \qquad \forall\ \mathbf{v}_h \in W_h, \tag{3.39}$$

where $C_* = \min(\sigma_*, \min(1,\xi_0)\eta_*)$ and $C^* = \max(\sigma^*, \max(1,\xi_0)\eta^*)$.

Proof. It is sufficient to use the stability condition (3.38) for $[\mathbf{v}_h, \mathbf{v}_h]_{W_h}$ and noting that for the coupling term we have the following

$$\min(1,\xi_0)\eta_* \sum_{\hat{\mathsf{f}}\in\Gamma_h} |\hat{\mathsf{f}}|(\llbracket \mathbf{v}_h \rrbracket^2_{\hat{\mathsf{f}}} + \{\mathbf{v}_h\}^2_{\hat{\mathsf{f}}}) \leq [\mathbf{v}_h, \mathbf{v}_h]_\Gamma \leq \max(1,\xi_0)\eta^* \sum_{\hat{\mathsf{f}}\in\Gamma_h} |\hat{\mathsf{f}}|(\llbracket \mathbf{v}_h \rrbracket^2_{\hat{\mathsf{f}}} + \{\mathbf{v}_h\}^2_{\hat{\mathsf{f}}})$$

to recover the bound on $a_h(\cdot,\cdot)$. □

Next, we show an equivalence result for the norms $||\cdot||_{W_h}$ and $|||\cdot|||_{W_h}$.

Lemma 3.3. *The discrete norms $|||\cdot|||_{W_h}$ and $||\cdot||_{W_h}$ are equivalent for any mesh size h. More precisely,*

$$|||\mathbf{v}_h|||_{W_h} \leq ||\mathbf{v}_h||_{W_h} \leq \sqrt{1 + \frac{C}{h}} |||\mathbf{v}_h|||_{W_h} \qquad \forall\ \mathbf{v}_h \in W_h. \tag{3.40}$$

Proof. Clearly it holds that

$$|||\mathbf{v}_h|||_{W_h} \leq ||\mathbf{v}_h||_{W_h} \qquad \forall\ \mathbf{v}_h \in W_h.$$

For the upper bound:

$$\begin{aligned}
||\mathbf{v}_h||^2_{W_h} &= \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} v_\mathsf{f}^2 + \sum_{\hat{\mathsf{f}}\in\Gamma_h} |\hat{\mathsf{f}}|([\![\mathbf{v}_h]\!]^2_{\hat{\mathsf{f}}} + \{\mathbf{v}_h\}^2_{\hat{\mathsf{f}}}) \\
&\leq \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} v_\mathsf{f}^2 + \sum_{\hat{\mathsf{f}}\in\Gamma_h} 2|\hat{\mathsf{f}}|(v^2_{\mathsf{f}^+} + v^2_{\mathsf{f}^-}) \\
&\leq \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} v_\mathsf{f}^2 + \sum_{\hat{\mathsf{f}}} 2Ch^{-1}(|\mathsf{P}_{\mathsf{f}^+}|v^2_{\mathsf{f}^+} + |\mathsf{P}_{\mathsf{f}^-}|v^2_{\mathsf{f}^-}) \\
&\leq \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} v_\mathsf{f}^2 + \frac{2C}{h} \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} v_\mathsf{f}^2 \\
&= \left(1 + \frac{C}{h}\right) |||\mathbf{v}_h|||^2_{W_h},
\end{aligned}$$

where in the second inequality we have made use of the bounds

$$|\hat{\mathsf{f}}| \leq Ch^{-1}|\mathsf{P}_{\mathsf{f}^+}|, \quad |\hat{\mathsf{f}}| \leq Ch^{-1}|\mathsf{P}_{\mathsf{f}^-}|,$$

which follows from the mesh regularity assumptions of the mimetic framework (C is a positive constant). Thus we can conclude the stated equivalence result. □

Spectral lemma for saddle point matrices

Let us now recall from [41] a lemma for the spectrum of a generic saddle point matrix that will be the starting point for the spectral analysis of $\mathcal{A}$. The result is the following.

Lemma 3.4. *Let us consider the following block matrix*

$$\mathcal{A} = \begin{bmatrix} \mathsf{A} & \mathsf{B}^\mathsf{T} \\ \mathsf{B} & -\mathsf{C} \end{bmatrix},$$

where $\mathsf{A} \in \mathbb{R}^{n\times n}$ is a symmetric positive definite matrix, $\mathsf{B} \in \mathbb{R}^{m\times n}$ is a full rank matrix and $\mathsf{C} \in \mathbb{R}^{m\times m}$ is a symmetric positive semi-definite matrix. Let us denote with λ_1^A and λ_n^A the maximum and minimum eigenvalues of A, with λ_1^C the maximum eigenvalue of C and with μ_1^B and μ_m^B the maximum and minimum singular values of B. Then we have that the spectrum of $\mathcal{A}$ satisfies

$$\sigma(\mathcal{A}) \subseteq I^- \cup I^+,$$

where

$$\begin{aligned}
I^- &= \left[\frac{1}{2}\left(\lambda_n^\mathsf{A} - \lambda_1^\mathsf{C} - \sqrt{(\lambda_n^\mathsf{A} + \lambda_1^\mathsf{C})^2 + 4(\mu_1^\mathsf{B})^2}\right),\ \ \frac{1}{2}\left(\lambda_1^\mathsf{A} - \sqrt{(\lambda_1^\mathsf{A})^2 + 4(\mu_m^\mathsf{B})^2}\right)\right] \subset \mathbb{R}^-, \\
I^+ &= \left[\lambda_n^\mathsf{A},\ \ \frac{1}{2}\left(\lambda_1^\mathsf{A} + \sqrt{(\lambda_1^\mathsf{A})^2 + 4(\mu_1^\mathsf{B})^2}\right)\right] \subset \mathbb{R}^+.
\end{aligned}$$

An assumption on the mesh

Following [51], we introduce an assumption on the structure of the mesh besides the mesh-regularity requirements stated in (**AM**), see Section 2.2 in Chapter 2. First of all we define the total grid $\Lambda_h = \Omega_h \cup \Gamma_h$, which contains both the bulk polyhedral cells and the fracture polygonal cells. Sometimes we will refer to a generic cell $c \in \Lambda_h$ that may be a bulk cell or a fracture cell. For a better visualization of the fracture facets as "cells", it may be useful thinking the fracture facets as virtual cells obtained by extrusion along their normal direction by a thickness of half the aperture.

Let us consider the graph $\mathcal{G}$ with set of nodes Λ_h and set of edges $\mathcal{L}_h \subseteq \Lambda_h \times \Lambda_h$ defined by

- $(\mathsf{P}_1, \mathsf{P}_2) \in \mathcal{L}_h$ if and only if $\exists \mathsf{f} \in \mathscr{F}_h^0$ such that $\Omega_h(\mathsf{f}) = \{\mathsf{P}_1, \mathsf{P}_2\}$,
- $(\mathsf{P}, \hat{\mathsf{f}}) \in \mathcal{L}_h$ if and only if $\exists \mathsf{f} \in \mathscr{F}_h^\Gamma$ such that $\Omega_h(\mathsf{f}) = \{\mathsf{P}\}$ and $\Gamma_h(\mathsf{f}) = \{\hat{\mathsf{f}}\}$.

Condition (M1). The mesh Λ_h satisfies the following assumption: there exists a family of elementary paths $\Psi = \{\gamma_i\}_{i=1,\dots,N_\gamma}$ which are connected subgraphs of $\mathcal{G}$ with nodes of maximal degree 2, such that

- for every cell $c \in \Lambda_h$ there exists one and only one path $\gamma_i \in \Psi$ with $c \in \gamma_i$, i.e. the family of paths Ψ defines a partition of Λ_h;
- the first ad last node, $c_{s,i}$ and $c_{e,i}$, of any $\gamma_i \in \Psi$ have a facet in $\mathscr{F}_h^{\partial\Omega}$;
- there exists a constant L^* independent from h such that

$$L_i \leq L^* h^{-1}, \qquad i = 1, ..., N_\gamma,$$

  where L_i is the number of cells contained in the path γ_i.

Spectral properties of the transmissibility matrix

To study the spectrum of the system a characterization of the spectral properties of the TPFA version of the finite volume scheme is required. However, no results on the spectrum of T are available in literature. We assume that for TPFA a similar spectral bound of linear finite elements holds. More precisely, we assume

$$M_* h^n \leq \lambda(\mathsf{T}) \leq M^* h^{n-2}, \tag{3.41}$$

where n is the dimension of the problem discretized with TPFA and M_*, M^* are positive constants independent from h. In our case we have $n = 2$ because a reduced model is considered in the fracture. See [55] for the proof of the spectral bound (3.41) for linear finite elements. We stress that numerical experiments shown in Section 5.3 of Chapter 5 validate the assumption.

3.5.3 Estimate of the condition number

This estimate is a generalization of the work contained in [51], where a standard diffusion problem was considered.

Theorem 3.5. *Let $\mathcal{A}$ be the matrix given in (3.37), under assumption (**M1**) the spectrum of $\mathcal{A}$ satisfies*

$$\sigma(\mathcal{A}) \subseteq [-k_1 h^{d-3},\ -k_2 h^{d+1}] \cup [k_3 h^d,\ k_4 h^{d-1}], \tag{3.42}$$

where k_i, $i = 1, \ldots, 4$ are positive constants h-independent. The 2-norm condition number is characterized by the following bound:

$$K_2(\mathcal{A}) \leq \frac{k_1}{k_2} h^{-4}. \tag{3.43}$$

Proof. We want to derive appropriate bounds for the eigenvalues and singular values involved in the two intervals I^- and I^+ of Lemma 3.4 to recover a result for the spectrum $\sigma(\mathcal{A})$. From it we derive the estimate of the condition number $K(\mathcal{A})$.

For $\lambda_1(\widetilde{\mathsf{T}}) = \lambda_1(\mathsf{T})$ we have assumed (see (3.41)) the following bound

$$\lambda_1(\mathsf{T}) \leq M^* h^{d-3}. \tag{3.44}$$

where M^* is a positive constant and d is the bulk dimension ($d = 3$ in our case).

Now we enter into the details of matrix M_c. Let us start with the maximum eigenvalue that can be written in terms of Rayleigh coefficient as

$$\lambda_1(\mathsf{M}_c) = \max_{\mathbf{u}\neq\mathbf{0}} \frac{\mathbf{u}^\mathsf{T}\mathsf{M}_c\mathbf{u}}{\mathbf{u}^\mathsf{T}\mathbf{u}} = \max_{\mathbf{u}\neq\mathbf{0}} \frac{\mathbf{u}^\mathsf{T}\mathsf{M}_c\mathbf{u}}{\frac{1}{2}\left(\sum\limits_{\mathsf{P}\in\Omega_h} \mathbf{u}_\mathsf{P}^\mathsf{T}\mathbf{u}_\mathsf{P} + \sum\limits_{\mathsf{f}\in\mathscr{F}_h^\Gamma\cup\mathscr{F}_h^{\partial\Omega}} u_\mathsf{f}^2\right)}.$$

Using the stability condition (3.39) and the norms equivalence result (3.40), we have the following bound:

$$C_* \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \mathbf{u}_\mathsf{P}^\mathsf{T}\mathbf{u}_\mathsf{P} \leq \mathbf{u}^\mathsf{T}\mathsf{M}_c\mathbf{u} \leq C^* \left(1 + \frac{C}{h}\right) \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \mathbf{u}_\mathsf{P}^\mathsf{T}\mathbf{u}_\mathsf{P} \qquad \forall\ \mathbf{u} \in \mathbb{R}^{N_\mathsf{f}}. \tag{3.45}$$

Using the right inequality we can derive:

$$\begin{aligned} \lambda_1(\mathsf{M}_c) &\leq 2C^* \left(1 + \frac{C}{h}\right) \max_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \max_{\mathbf{u}\neq\mathbf{0}} \frac{\sum\limits_{\mathsf{P}\in\Omega_h} \mathbf{u}_\mathsf{P}^\mathsf{T}\mathbf{u}_\mathsf{P}}{\left(\sum\limits_{\mathsf{P}\in\Omega_h} \mathbf{u}_\mathsf{P}^\mathsf{T}\mathbf{u}_\mathsf{P} + \sum\limits_{\mathsf{f}\in\mathscr{F}_h^\Gamma\cup\mathscr{F}_h^{\partial\Omega}} u_\mathsf{f}^2\right)} \\ &\leq 2C^* \left(1 + \frac{C}{h}\right) \max_{\mathsf{P}\in\Omega_h} |\mathsf{P}| \leq 2C^* \left(1 + \frac{C}{h}\right) a^* h^d, \end{aligned} \tag{3.46}$$

having used in the last inequality the fact that the volume of a cell scales in a uniform way thanks to the mesh regularity assumptions of the mimetic framework (a^* is a positive constant). Asymptotically we get

$$\lambda_1(\mathsf{M}_c) \lesssim 2C^* C a^* h^{d-1}. \tag{3.47}$$

For the minimum eigenvalue the procedure is similar:

$$\begin{aligned}\lambda_{N_\mathsf{f}}(\mathsf{M}_c) = \min_{\mathbf{u}\neq\mathbf{0}} \frac{\mathbf{u}^\mathsf{T}\mathsf{M}_c\mathbf{u}}{\mathbf{u}^\mathsf{T}\mathbf{u}} = \min_{\mathbf{u}\neq\mathbf{0}} \frac{\mathbf{u}^\mathsf{T}\mathsf{M}_c\mathbf{u}}{\left(\sum\limits_{\mathsf{P}\in\Omega_h}\mathbf{u}_\mathsf{P}^\mathsf{T}\mathbf{u}_\mathsf{P} - \sum\limits_{\mathsf{f}\in\mathscr{F}_h^0} u_\mathsf{f}^2\right)} \\ \geq C_* \min_{\mathsf{P}\in\Omega_h}|\mathsf{P}| \min_{\mathbf{u}\neq\mathbf{0}} \frac{\sum\limits_{\mathsf{P}\in\Omega_h}\mathbf{u}_\mathsf{P}^\mathsf{T}\mathbf{u}_\mathsf{P}}{\left(\sum\limits_{\mathsf{P}\in\Omega_h}\mathbf{u}_\mathsf{P}^\mathsf{T}\mathbf{u}_\mathsf{P} - \sum\limits_{\mathsf{f}\in\mathscr{F}_h^0} u_\mathsf{f}^2\right)} \geq C_* a_* h^d.\end{aligned} \tag{3.48}$$

We focus now on the singular values of the matrix $\widetilde{\mathsf{B}}$, defined as

$$(\mu_1(\widetilde{\mathsf{B}}))^2 = \lambda_1(\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T}), \quad (\mu_{N_\mathsf{P}+N_\Gamma}(\widetilde{\mathsf{B}}))^2 = \lambda_{N_\mathsf{P}+N_\Gamma}(\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T}).$$

We have then to study the spectrum of the matrix $\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T} \in \mathbb{R}^{(N_\mathsf{P}+N_\Gamma)\times(N_\mathsf{P}+N_\Gamma)}$, which is symmetric and has the following block structure:

$$\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T} = \begin{bmatrix}\mathsf{B}\\ \mathsf{C}\end{bmatrix}\begin{bmatrix}\mathsf{B}^\mathsf{T} & \mathsf{C}^\mathsf{T}\end{bmatrix} = \begin{bmatrix}\mathsf{B}\mathsf{B}^\mathsf{T} & \mathsf{B}\mathsf{C}^\mathsf{T}\\ \mathsf{C}\mathsf{B}^\mathsf{T} & \mathsf{C}\mathsf{C}^\mathsf{T}\end{bmatrix}.$$

We study each block to characterize the elements of the matrix in detail. The block (1,1) is $\mathsf{B}\mathsf{B}^\mathsf{T} \in \mathbb{R}^{N_\mathsf{P}\times N_\mathsf{P}}$ and starting from the formula (3.34) for the elements of B, we have:

$$[\mathsf{B}\mathsf{B}^\mathsf{T}]_{ij} = \begin{cases}\sum\limits_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P}_i)}|\mathsf{f}|^2 & \text{if } i=j\\ -\sum\limits_{l=1}^{k}|\mathsf{f}_l|^2 & \text{if } i\neq j, \text{ where } \{\mathsf{f}_1,...,\mathsf{f}_k\} = \mathscr{F}_h(\mathsf{P}_i)\cap\mathscr{F}_h(\mathsf{P}_j)\\ 0 & \text{otherwise.}\end{cases} \tag{3.49}$$

Note that in the summation for $i \neq j$ the boundary and fracture facets can never appear, because two elements of Ω_h cannot share neither a boundary nor a (decoupled) fracture facet. Now we focus on the block (1,2) that is the matrix $\mathsf{B}\mathsf{C}^\mathsf{T} \in \mathbb{R}^{N_\mathsf{P}\times N_\Gamma}$. From formulas (3.34) and (3.36) for the elements of B and C, we derive:

$$[\mathsf{B}\mathsf{C}^\mathsf{T}]_{ij} = \begin{cases}-|\hat{\mathsf{f}}_j|^2 & \text{if } \mathscr{F}_h(\mathsf{P}_i)\cap\mathscr{F}_h(\hat{\mathsf{f}}_j)\neq\emptyset\\ 0 & \text{otherwise.}\end{cases} \tag{3.50}$$

Note that if $\mathscr{F}_h(\mathsf{P}_i)\cap\mathscr{F}_h(\hat{\mathsf{f}}_j)$ is not void it necessarily contains one and only one (decoupled) fracture facet. The block (2,1) is simply the transpose of the block (1,2); so up to now only the block (2,2) is left to be described. It represents the matrix $\mathsf{C}\mathsf{C}^\mathsf{T} \in \mathbb{R}^{N_\Gamma\times N_\Gamma}$, which has the following characterization, obtained from (3.36):

$$[\mathsf{C}\mathsf{C}^\mathsf{T}]_{ij} = \begin{cases}2|\hat{\mathsf{f}}_i|^2 & \text{if } i=j\\ 0 & \text{if } i\neq j.\end{cases} \tag{3.51}$$

Note that $\mathsf{C}\mathsf{C}^\mathsf{T}$ is a diagonal matrix.

We start from the upper bound for $\lambda_N(\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T})$. We apply the Gershgorin Circle Theorem to the matrix $\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T}$. The matrix is real and symmetric, so its eigenvalues

are real and the circles are intervals; moreover the row circles and column circles coincide. So we have that

$$\sigma(\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^{\mathsf{T}}) \subseteq \bigcup_{i=1}^{N_{\mathsf{P}}+N_{\Gamma}} R_i,$$

where R_i are the row circles. Taking into account the characterization of $\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^{\mathsf{T}}$ described by (3.49)-(3.51) relationships, we have two types of row circles:

$$\begin{aligned} &i = 1, \dots, N_{\mathsf{P}} \qquad x_c^i = \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P}_i)} |\mathsf{f}|^2, \quad r^i = \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P}_i)\setminus\mathscr{F}_h^{\partial\Omega}} |\mathsf{f}|^2 \\ &i = N_{\mathsf{P}}, \dots, N_{\mathsf{P}}+N_{\Gamma} \qquad x_c^i = 2|\hat{\mathsf{f}}_{i-N_{\mathsf{P}}}|^2, \quad r^i = 2|\hat{\mathsf{f}}_{i-N_{\mathsf{P}}}|^2, \end{aligned}$$

where x_c^i and r^i represent the center and the radius of the i-th circle R_i, respectively. From these considerations we can derive the upper bound:

$$\begin{aligned} \lambda_1(\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^{\mathsf{T}}) &\le \max\left\{ \max_{\mathsf{P}\in\Omega_h}\left(\sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})} |\mathsf{f}|^2 + \sum_{\mathsf{f}\in\mathscr{F}_h(\mathsf{P})\setminus\mathscr{F}_h^{\partial\Omega}} |\mathsf{f}|^2 \right),\ 4\max_{\hat{\mathsf{f}}\in\Gamma_h} |\hat{\mathsf{f}}|^2 \right\} \\ &\le \max\left\{ 2N^* \max_{\mathsf{f}\in\mathscr{F}_h} |\mathsf{f}|^2,\ 4\max_{\hat{\mathsf{f}}\in\Gamma_h} |\hat{\mathsf{f}}|^2 \right\} = 2N^* \max_{\mathsf{f}\in\mathscr{F}_h} |\mathsf{f}|^2 \le 2N^* b^* h^{2(d-1)}, \end{aligned} \tag{3.52}$$

where we have made use of the uniform scaling of the area of a polygonal facet.

The estimation of the smallest eigenvalue $\lambda_{N_{\mathsf{P}}+N_{\Gamma}}$ requires the additional mesh assumption (**M1**). Let us start noting that

$$[\widetilde{\mathsf{B}}^{\mathsf{T}}\widetilde{\mathbf{p}}]_{\mathsf{f}} = \begin{cases} -\sum_{\mathsf{P}\in\Omega_h(\mathsf{f})} \alpha_{\mathsf{P},\mathsf{f}}|\mathsf{f}|p_{\mathsf{P}}, & \text{if } \mathsf{f}\in\mathscr{F}_h\setminus\mathscr{F}_h^{\Gamma} \\ -\alpha_{\mathsf{P},\mathsf{f}}|\mathsf{f}|p_{\mathsf{P}} + \alpha_{\mathsf{P},\mathsf{f}}|\mathsf{f}|p_{\Gamma,\hat{\mathsf{f}}}, & \text{if } \mathsf{f}\in\mathscr{F}_h^{\Gamma},\ \text{where } \{\mathsf{P}\}=\Omega_h(\mathsf{f}),\ \{\hat{\mathsf{f}}\}=\Gamma_h(\mathsf{f}). \end{cases}$$

More in detail, we have three cases:

$$[\widetilde{\mathsf{B}}^{\mathsf{T}}\widetilde{\mathbf{p}}]_{\mathsf{f}} = \begin{cases} -\alpha_{\mathsf{P},\mathsf{f}}|\mathsf{f}|p_{\mathsf{P}}, & \text{if } \mathsf{f}\in\mathscr{F}_h^{\partial\Omega},\ \text{where } \{\mathsf{P}\}=\Omega_h(\mathsf{f}) \\ \alpha_{\mathsf{P}_1,\mathsf{f}}|\mathsf{f}|(p_{\mathsf{P}_2}-p_{\mathsf{P}_1}), & \text{if } \mathsf{f}\in\mathscr{F}_h^{0},\ \text{where } \{\mathsf{P}_1,\mathsf{P}_2\}=\Omega_h(\mathsf{f}) \\ \alpha_{\mathsf{P},\mathsf{f}}|\mathsf{f}|(p_{\Gamma,\hat{\mathsf{f}}}-p_{\mathsf{P}}), & \text{if } \mathsf{f}\in\mathscr{F}_h^{\Gamma},\ \text{where } \{\mathsf{P}\}=\Omega_h(\mathsf{f}),\ \{\hat{\mathsf{f}}\}=\Gamma_h(\mathsf{f}). \end{cases}$$

Now we write the Rayleigh coefficient in terms of $\widetilde{\mathsf{B}}^{\mathsf{T}}\widetilde{p}$ and we use the previous characterization.

$$\begin{aligned} \lambda_{N_{\mathsf{P}}+N_{\Gamma}}(\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^{\mathsf{T}}) &= \min_{\widetilde{\mathbf{p}}\neq\mathbf{0}} \frac{(\widetilde{\mathsf{B}}^{\mathsf{T}}\widetilde{\mathbf{p}})^{\mathsf{T}}(\widetilde{\mathsf{B}}^{\mathsf{T}}\widetilde{\mathbf{p}})}{\widetilde{\mathbf{p}}^{\mathsf{T}}\widetilde{\mathbf{p}}} \\ &= \min_{\widetilde{\mathbf{p}}\neq\mathbf{0}} \frac{\sum_{\mathsf{f}\in\mathscr{F}_h^{\partial\Omega}} |\mathsf{f}|^2 p_{\mathsf{P}}^2 + \sum_{\mathsf{f}\in\mathscr{F}_h^{0}} |\mathsf{f}|^2 (p_{\mathsf{P}_2}-p_{\mathsf{P}_1})^2 + \sum_{\mathsf{f}\in\mathscr{F}_h^{\Gamma}} |\mathsf{f}|^2 (p_{\Gamma,\hat{\mathsf{f}}}-p_{\mathsf{P}})^2}{\sum_{\mathsf{P}\in\Omega_h} p_{\mathsf{P}}^2 + \sum_{\hat{\mathsf{f}}\in\Gamma_h} p_{\Gamma,\hat{\mathsf{f}}}^2} \\ &\ge \min_{\mathsf{f}\in\mathscr{F}_h} |\mathsf{f}|^2 \min_{\widetilde{\mathbf{p}}\neq\mathbf{0}} \frac{\sum_{\mathsf{f}\in\mathscr{F}_h^{\partial\Omega}} p_{\mathsf{P}}^2 + \sum_{\mathsf{f}\in\mathscr{F}_h^{0}} (p_{\mathsf{P}_2}-p_{\mathsf{P}_1})^2 + \sum_{\mathsf{f}\in\mathscr{F}_h^{\Gamma}} (p_{\Gamma,\hat{\mathsf{f}}}-p_{\mathsf{P}})^2}{\sum_{\mathsf{P}\in\Omega_h} p_{\mathsf{P}}^2 + \sum_{\hat{\mathsf{f}}\in\Gamma_h} p_{\Gamma,\hat{\mathsf{f}}}^2} \end{aligned}$$

Without loss of generality it's possible to suppose that the cells of Λ_h, i.e. the bulk and fracture cells, are ordered path by path. Therefore the vector $\widetilde{\mathbf{p}}$ may be partitioned in $[\widetilde{\mathbf{p}}_1, \ldots, \widetilde{\mathbf{p}}_{N_\gamma}]^\mathsf{T}$, where $\widetilde{\mathbf{p}}_i = [\widetilde{p}_{c_{s,i}}, \ldots, \widetilde{p}_{c_{e,i}}]^\mathsf{T}$ are associated to the nodes (cells) in path γ_i. For each edge e of γ_i we indicate with $\widetilde{p}_{e_{1,i}}$ and $\widetilde{p}_{e_{2,i}}$ the elements of $\widetilde{\mathbf{p}}_i$ associated to the cells at end of the edge. So we can write

$$\lambda_{N_\mathsf{P}+N_\Gamma}(\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T}) \geq b_* h^{2(d-1)} \min_{\widetilde{\mathbf{p}}\neq\mathbf{0}} \frac{\sum_{i=1}^{N_\gamma}\left(\widetilde{p}^2_{c_{s,i}} + \sum_{e\in\gamma_i}(\widetilde{p}_{e_{2,i}} - \widetilde{p}_{e_{1,i}})^2 + \widetilde{p}^2_{c_{e,i}}\right)}{\sum_{c\in\Lambda_h}\widetilde{p}^2_c}.$$

This expression is equivalent to the Rayleigh coefficient for calculating the smallest eigenvalue of the block diagonal matrix

$$\Sigma = b_* h^{2(d-1)} \mathsf{diag}(\mathsf{E}_1, \mathsf{E}_2, \ldots, \mathsf{E}_{N_{\gamma-1}}, \mathsf{E}_{N_\gamma}) \in \mathbb{R}^{(N_\mathsf{P}+N_\Gamma)\times(\mathsf{N}_\mathsf{P}+N_\Gamma)},$$

where $\mathsf{E}_i = \mathsf{tridiag}(-1, 2, -1) \in \mathbb{R}^{L_i \times Li}$. The eigenvalues of E_i can be computed explicitly as

$$\lambda_j(\mathsf{E}_i) = 2\left(1 - \cos\frac{L_i + 1 - j}{L_i + 1}\right), \qquad j = 1, \ldots, L_i$$

and for the minimum eigenvalue we have $\lambda_{L_i}(\mathsf{E}_i) = 2\left(1 - \cos\frac{1}{L_i+1}\right)$. Therefore we have

$$\lambda_{N_\mathsf{P}+N_\Gamma}(\Sigma) = b_* h^{2(d-1)} \min_{i=1,\ldots,N_\gamma} \lambda_{L_i}(\mathsf{E}_i) = 2b_* h^{2(d-1)} \min_{i=1,\ldots,N_\gamma}\left(1 - \cos\frac{1}{L_i+1}\right).$$

From assumption (**M1**) on the maximum length of the paths we have that

$$\lambda_{\mathsf{N}_\mathsf{P}+N_\Gamma}(\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T}) \geq \lambda_{N_\mathsf{P}+N_\Gamma}(\Sigma) \geq 2b_* h^{2(d-1)}\left(1 - \cos\frac{1}{L_* h^{-1} + 1}\right).$$

We can conclude that, asymptotically,

$$\lambda_{N_\mathsf{P}+N_\Gamma}(\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T}) \gtrsim 2b_* h^{2(d-1)} \frac{h^2}{2L_*^2} = \frac{b_*}{L_*^2} h^{2d}. \tag{3.53}$$

The bounds to characterize the intervals I^- and I^+ of Lemma 3.4 are given by estimates (3.44), (3.47), (3.48), (3.52) and (3.53). Noting that the function $(\lambda, \mu) \mapsto \frac{1}{2}(\lambda - \sqrt{\lambda^2 + 4\mu^2})$ is monotonically increasing in the first argument and monotonically decreasing in the second argument, we have that

$$\sigma(\mathcal{A}) \subseteq I^- \cup I^+,$$

where

$$\begin{aligned} I^- \subseteq \Bigg[&\frac{1}{2}\left(C_* a_* h^d - M^* h^{d-3} - \sqrt{(C_* a_* h^d - M^* h^{d-3})^2 + 8N^* b^* h^{2(d-1)}}\right), \\ &\frac{1}{2}\left(2C^* C a^* h^{d-1} - \sqrt{4(C^* C a^* h^{d-1})^2 + 4b_* L_*^{-2} h^{2d}}\right)\Bigg], \end{aligned} \tag{3.54}$$

$$I^+ \subseteq \left[C_* a_* h^d, \quad \frac{1}{2}\left(2C^*Ca^*h^{d-1} + \sqrt{4(C^*Ca^*h^{d-1})^2 + 8N^*b^*h^{2(d-1)}}\right)\right]. \quad (3.55)$$

Now we can recover the asymptotic spectrum estimate (3.42). For the lower extreme of the first interval we have the estimate $-M^*h^{d-3}$. For the upper extreme it's a bit less immediate. Let us define the constants $\nu = 2C^*Ca^*$ and $\rho = 4b_*L_*^{-2}$ and the function of the mesh size $f(h) = \frac{1}{2}\left(\nu h^{d-1} - \sqrt{\nu^2 h^{2(d-1)} + \rho h^{2d}}\right)$. We have:

$$\begin{aligned} f(h) &= \frac{1}{2}\left(\nu h^{d-1} - \nu h^{d-1}\sqrt{1 + \frac{\rho}{\nu^2}h^2}\right) = \frac{\nu}{2}h^{d-1}\left(1 - \sqrt{1 + \frac{\rho}{\nu^2}h^2}\right) \\ &\sim \frac{\nu}{2}h^{d-1}\left(1 - \left(1 + \frac{\rho}{2\nu^2}h^2\right)\right) = -\frac{\rho}{4\nu}h^{d+1}. \end{aligned}$$

So for the upper extreme we get $-\dfrac{b_*}{2L_*^2 C^*Ca^*}h^{d+1}$.

Whereas, for the second interval we have $C_* a_* h^d$ for the lower extreme and an estimate of $(C^*Ca^* + \sqrt{(C^*Ca^*)^2 + 2N^*b^*})h^{d-1}$ for the upper one. Defining the following constants

$$\begin{aligned} k_1 &= M^* \\ k_2 &= \frac{b_*}{2L_*^2 C^*Ca^*} \\ k_3 &= C_* a_* \\ k_4 &= (C^*Ca^* + \sqrt{(C^*Ca^*)^2 + 2N^*b^*}), \end{aligned}$$

we recover the asymptotic spectrum estimate (3.42).

From the spectral estimate we can conclude that

$$||\mathcal{A}||_2 \leq k_1 h^{d-3}, \qquad ||\mathcal{A}^{-1}||_2 \leq \frac{1}{k_2}h^{-d-1}$$

and from these relationships we derive the estimate of the condition number (3.43):

$$K_2(\mathcal{A}) = ||\mathcal{A}||_2 ||\mathcal{A}^{-1}||_2 \leq \frac{k_1}{k_2}h^{-4}.$$

□

Remark 3.4. *Numerical experiments have shown that probably the bound on the condition number is not sharp and may provide an overestimation. Further developments are surely required in the spectral analysis of the system. See numerical experiments in Section 5.3 of Chapter 5 for a deeper discussion on this topic.*

Chapter 4

The C++ code

In this chapter we describe the implementation details of the code which implements the numerical model and the iterative preconditioned solver described in the previous chapters. The work starts from an existing code implementing the TPFA version of the finite volume scheme in a 3D fractured porous medium discretized via a polyhedral mesh (the numerical model was the 3D extension of the work developed in [61]). The original program solved, via the FV method, both the steady and pseudo-steady problem with direct solvers; in this thesis all the work has been carried out on the steady state problem. In the code there was also a first simple implementation of MFD in the bulk, but the scheme was in primal form and the coupling conditions were implemented only in a particular case ($\xi = 1$). With this thesis work we have converted the implementation of MFD in mixed form 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; to have a comprehensive guide on C++ see [50]. The development of a proper iterative solver and the implementation of the preconditioners proposed in Chapter 3 have been the other main task of the programming work. 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, the `IML++` library has been adapted to the `Eigen` framework.

A remarkable coding feature is the capability of the solver to choose at run time the storage format of the matrix of the system. Indeed, 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 is stored, because the LU factorization requires the matrix in its monolithic form. The fact that the storage format of the matrix is chosen 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. Another interesting aspect is that, with respect to the codes developed in [61] and [60], where 2D problems were considered, not only the part in term of iterative solver is new, but also the assembly of the matrices has been developed in a different way in order to increase the efficiency of the code.

In the following sections we describe the implementation details regarding the

assembly of the system, the classes that handle the preconditioners and the iterative solvers and the top level class `Problem`, which makes uses of all the utilities of the code. Finally we describe how to use the code, showing the setting of the input parameters of the software and presenting a practical tutorial. Regarding the mesh, we do not enter into the implementation details, we just make only some considerations in the following paragraph.

The mesh

A first important aspect is clearly the mesh. The mesh is read from a `.fvg` file (see Section 4.6 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 [63] 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 assembly process. The reason is simply that the `Mesh3D` class implements a mesh flexible with respect to refinement and modification, indeed a lot of data in `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, such as 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 [61] and [60]. One difference is that in our case `Rigid_Mesh` is no more a template class with respect to the problem dimension, because we focus only on the 3D case. Moreover, since in a 3D framework facets and edges are two different things, in addition to the classes for cells and facet a proper class `edge` has been developed, together 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 [61] and [60] and, since it is not the main focus of this thesis, we omit its presentation.

4.1 The numerical operators

In [61] and [60] 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 [61] and [60] 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 is necessary to define a proper object for every discrete operator (the mimetic inner product, the weak divergence, the coupling conditions...) and to build the corresponding 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 local matrices M_P and B_P given by (2.21)-(2.23) and (3.35), respectively. The best way of proceeding to assemble the bulk problem is:

- Do a loop over the cells of the mesh.

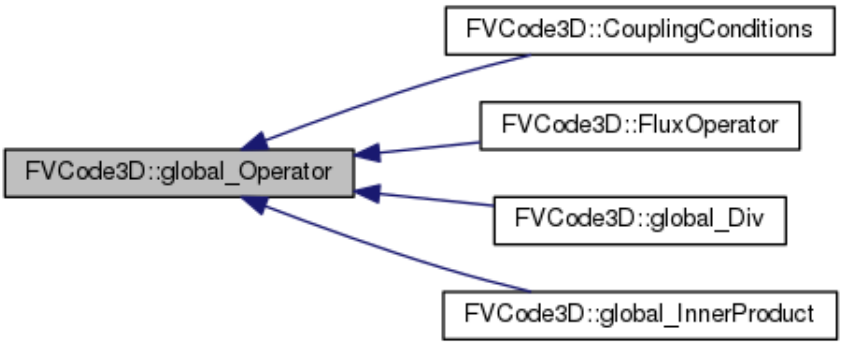


Figure 4.1: Inheritance graph for `global_Operator`.

(1) Do a loop over the facets of the cell to build up $\mathsf{M_P}$ and $\mathsf{B_P}$.

(2) Do a loop over the facets of the cell to assemble $\mathsf{M_P}$ and $\mathsf{B_P}$ in M and B.

The assembly step (2) is equivalent to what is done normally in FEM schemes (so the global IDs of the degrees of freedom of the element is recovered and the local matrix is assembled 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 routine pointed out before must be repeated two times, one for M and the other for B, and so we are cycling more times both on cells and facets. Taking into account that assembling a matrix like B requires only simple memory accesses, a single loop would be better. 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 this way a more efficient routine has been derived. This has been implemented through specific builder classes, and this is a notable improvement with respect to [61] and [60].

The class `global_Operator`

Four classes have been implemented and every class 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 4.1. The term "global" reflects the fact that these classes assemble their matrices on the whole mesh (the mimetic operators have also a local version). The class `global_Operator` is briefly 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, which indicate the types of degrees of freedom 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 setting of `Ncol` is clearly the same. `M_mesh` is a constant reference to the `Rigid_Mesh` and plays an essential role in the assembly; it is also fundamental to size properly the matrix as seen above. It is important to point out that `dFacet` takes into account the decoupling of fracture facets, which, in other words, is the decoupling of the velocity degrees of freedom of the fracture facets. To the degrees of freedom of velocity corresponding to the facets of the mesh, we add N_Γ additional degrees, one for every fracture facet. Given a mesh facet f of a polyhedron P that is matching a fracture cell, we state the following convention:

- 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 N_{f} is the number of facets of the mesh and ID_Γ is the identifier of the facet 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 matrix of the system that, as always in numerical methods for PDEs, is a sparse matrix. It is stored as an `SpMat` that is a proper `typedef` for an `Eigen` sparse matrix:

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

where `Real` is simply 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 [64]). 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, since it allows to use some efficient direct solvers, like the UmfPack, and also for the simple fact that all the `Eigen` algorithms have been tested more extensively on column-major matrices. There are basically two ways of inserting elements in an `SpMat`. The first is to use a vector of `Eigen::Triplet` that stores for every non zero three data: 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 to 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 those of the fractures are assembled jointly with fewer 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 is 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 is 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, which employ the utilities of the single operators to build the matrices of the problem directly in the system in a more efficient way.

The class `global_InnerProduct`

Let us start 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 analyse 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 methods `assembleFace`. Let us point out the assembly algorithm:

- Do a loop over the mesh cells.
  (1) Build the local matrix $\mathsf{M_P}$.
  (2) Assemble $\mathsf{M_P}$ 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 $M(1, \texttt{iloc:end})$ and $M(\texttt{iloc:end}, 1)$ 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 taken as input, and the assembly is carried out with the row offset `rowoff` and column offset `coloff`. In practice this `assembleFace` version allows 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 $\widetilde{B}$ and $\widetilde{T}$, 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, indeed 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 N_P and R_P matrices, which are built through some geometrical quantities of the cell (see (2.13) and (2.17)), the consistency matrix M_P^0 (see (2.22)), the stability matrix M_P^1 (see (2.23)) and the final local inner product matrix M_P, which arises 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 M_P because one may be interested in studying them (the theory behind the construction of M_P, e.g. its spectral properties, are a challenging topic: see [14]). In principle the storage as attributes of these matrices is not required, so after

the creation of 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), which is the 2 factor in the formula. The 2 value has been estimated as the best one in [14]. It is `static` because it can be considered a constant of the scheme. Finally 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);
    }
}
```

The class `global_Div`

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

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 is 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`, which 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 a matrix S using

the offsets `rowoff` and `coloff`. In the case of `assembleFrFace` there is 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);
    }
}
```

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 shown here.

```
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();
```

First, let us point out 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; and 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 of the corresponding bulk boundary is imposed, while on an internal tip edge an homogeneous 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 vertexes 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 separated 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 method `findFracturesAlpha` 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, which 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`, which is a base class of a complex inheritance tree 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 is very easy implementing the assembly:

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

4.2 The builders of the system

In the code the builders are those classes 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.

```
 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, 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, so it is deleted together with the default one, which could not initialize the

reference; only one constructor that initializes the reference is kept. The destructor is defaulted and `virtual`.

We have two pure virtual methods: `reserve_space` and `build`. The first one computes the sparsity patterns of the interested matrices and stores the proper space in `S`, the second actually builds the interested matrices in `S`, which 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.

4.2.1 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.

```
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 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 overrides the methods `reserve_space` and `build` in the case of only one argument `S`, which represents the monolithic matrix of the system. The `reserve_space` method computes the sparsity pattern of M, B and B^T and stores the proper space for these matrices in `S`; while the method `build` assembles the three matrices in `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 $\widetilde{\mathsf{B}}$. The arguments `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, which is more interesting. For the direct case the method is similar, only the row offset and the `bool transpose` in the `assembleFace` of the divergence operator change, to accommodate 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 note that, as already pointed out, the assembly is carried out with only one cell loop. Moreover, also M_P and B_P are built together with only one loop over the facets of the cell. This is performed through an object `localBUILDER` of class `local_builder`. We omit 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 `local_InnerProduct` and `local_Div` to build M_P and 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.

4.2.2 The class `FractureBuilder`

The class `FractureBuilder` inherits from `global_Builder`. 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 separated 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 fractures.

Now we show the methods, which basically are equivalent to those 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 and the matrices C, C^T and T directly in the `S` argument, which is the monolithic matrix of the system. While the second versions of the methods reserve the proper space and assemble the coupling modification in M (to obtain the M_c block), the matrix C in the block $\widetilde{B}$, and the matrix T in the block $\widetilde{T}$. The blocks $\widetilde{B}$ and $\widetilde{T}$ 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-contributions 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 steps (1)-(2). 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 $\widetilde{B}$; the `bool false` indicates to avoid the assembly of C^T. The `assembleFrFace` method of `FluxOp` takes care of the steps (3)-(5) and takes in input two offsets equal to N_P because of the position of T in $\widetilde{T}$.

4.2.3 The class SaddlePoint_StiffMatHandler

Now we introduce the class that uses all the utilities described up to here to build the whole matrix of the system. Actually we have two classes that have this goal: the class StiffMatHandlerMFD, which builds the matrix in the case of a direct solver, and the class SaddlePoint_StiffMatHandler, which does the same thing in the case of an iterative solver. The class StiffMatHandlerMFD assembles the matrix of the system in monolithic form as a SpMat. The class SaddlePoint_StiffMatHandler assembles a matrix stored in blocks form, which is implemented with a specific class called SaddlePointMat that we describe in this section. The class StiffMatHandlerMFD belongs to an inheritance tree in which also the case of a full FV scheme (both in bulk and fractures) is implemented through a specific derived class. This is because, until a direct solver is used, the storage of the matrix is always a SpMat and the full FV code supports only direct solvers up to now. We omit 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 the matrix 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 does not 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, which follows the same principles of the class BCcond in [60]. The need for 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 in order to impose the boundary conditions and this is performed through the class `BCimposition`. It requires for its construction a reference to `Rigid_Mesh` and a reference to `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. More precisely, the Dirichlet condition on pressure affects only `M_b`, while the Neumann condition, imposed through the Nitsche method, affects the two matrix blocks and `M_b`. For the boundary conditions on 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 matrix and the vector. We do not show here the class `BCimposition`. Finally 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 the following attributes:

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

`M`, `Bt` and `Tt` are the three blocks M_c, $\widetilde{\mathsf{B}}$ and $\widetilde{\mathsf{T}}$, stored as `SpMat`, while `isSymUndef` is a coefficient that is 1 if the matrix is symmetric but indefinite, -1 if the matrix is unsymmetric but positive definite. This distinction is necessary because the HSS preconditioner is designed only for unsimmetryc positive definite saddle point matrices, so if the user select the HSS preconditioner the `isSymUndef` will be set to -1, otherwise it will be set to 1 that is the default value.

This class represents a saddle point matrix and so, to make it flexible, we have implemented different constructors; clearly it must be also copyable, so there is a copy constructor. We have a defaulted constructor, a constructor 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 $\widetilde{\mathsf{B}}$. 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 is 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.

```
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()) = isSymUndef*Bt*x.head(M.cols()) +
        isSymUndef*T*x.tail(Bt.rows());
    return result;
}
```

Note that `isSymUndef` allows to handle both the symmetric indefinite case (3.11), for the block triangular and ILU preconditioners, and the unsymmetric positive definite case (3.12), for the HSS preconditioner. The `Eigen` methods `head` and `tail` allow to extract a top or bottom sub-vector, while the $\widetilde{B}^T$ is obtained through the `Eigen` method `transpose`. The overload of the matrix-vector product is very important because it is one of the two ingredients necessary for the preconditioning of the system (see Section 3.3 in Chapter 3).

4.3 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`, which is the first operation needed to solve a linear system with a preconditioned iterative method. The second operation is solving the linear system $P\mathbf{z} = \mathbf{r}$, so a preconditioner class has to provide a method `solve` that, given a vector $\mathbf{r}$, solves this linear system and returns the vector $\mathbf{z}$. The abstract base class `preconditioner` of the inheritance tree in Figure 4.2 collects just the pure virtual method that performs this operation:

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

The classes `identity_preconditioner` and `diagonal_preconditioner` 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 huge amount of iterations is needed to obtain convergence with a suitable 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 other three preconditioners implemented in the code are the block triangular and the block ILU preconditioners, both based on the

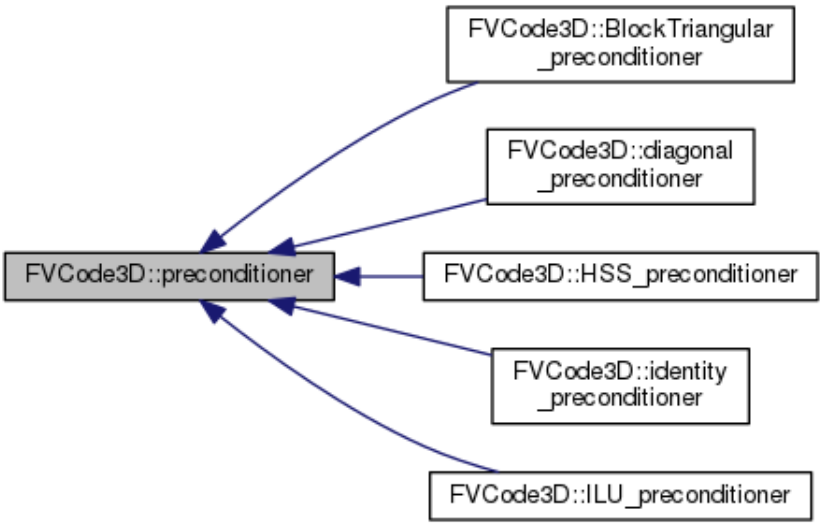


Figure 4.2: Inheritance graph for `preconditioner`.

diagonal part of M_c, and the HSS preconditioner; all of them have been introduced theoretically in Section 3.4 of Chapter 2. The first is implemented in the class `BlockTriangular_preconditioner`, the second in `ILU_preconditioner` and the third in `HSS_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 in a class `preconHandler` 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 fill the map and it is implemented in the class `preconProxy<T>`, which is a template class on the type of preconditioner. This is a classical factory pattern about which we do not enter into the details, see [3] for an in-depth survey on singletons and factory patterns. 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, which implements the block triangular preconditioner described in Section 3.4 of Chapter 3.

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

    void set(const SaddlePointMat & SP);
    Vector solve(const Vector & r) const;

private:
    const SpMat *     Btptr;
    DiagMat           Md_inv;
    CholDec           chol;
```

```
};
```

Let us note that the operations reported in (3.18) for the inversion of the preconditioner involve three matrices: the block $\widetilde{\mathsf{B}}$, the diagonal part of the inner product matrix $\widehat{\mathsf{M}}_c$ (or better its inverse) and the inexact Schur Complement $\widehat{\mathsf{S}} = -(\widetilde{\mathsf{T}} + \widetilde{\mathsf{B}}\widehat{\mathsf{M}}_c^{-1}\widetilde{\mathsf{B}}^{\mathsf{T}})$. The constant pointer `Btptr` provides access to the block matrix $\widetilde{\mathsf{B}}$, whereas the attribute `Md_inv` represents $\widehat{\mathsf{M}}_c^{-1}$. 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 employ 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 initialize a reference in the default constructor.

The other matrix involved in (3.18) is $\widehat{\mathsf{S}}$. The linear system associated with this matrix is solved in the code through a sparse Cholesky factorization. More precisely, the matrix $\widehat{\mathsf{S}}$ is pre-factorized and, for every iteration of GMRES, a couple of triangular linear systems is solved. The matrix $\widehat{\mathsf{S}}$ is assembled, but there is no need to store it as an attribute; indeed in the `solve` method only the factor matrix of the Cholesky decomposition is needed. Therefore, the class has an attribute `chol` that allows to perform the factorization and to solve the triangular systems. The type `CholDec` is a typedef for the following `Eigen` class

```
typedef Eigen::SimplicialCholesky<SpMat, Eigen::Upper> CholDec;
```

This class implements the Cholesky factorization $\widehat{\mathsf{S}} = \mathsf{L}\mathsf{L}^{\mathsf{T}}$ for sparse, symmetric and positive definite matrices. The first template parameter indicates the matrix type and the second what triangular factor has to be constructed with the decomposition (upper or lower). We mention that, in order to reduce the fill-in, a symmetric permutation P is applied prior to the factorization such that the factorized matrix is $\mathsf{P}\widehat{\mathsf{S}}\mathsf{P}^{-1}$. For details on this technique we refer to [29] and for its implementation to [64].

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

To set up the preconditioner the following method is used:

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

It takes the `SaddlePointMat SP`, which is the matrix of the system, and it sets `Btptr`, extracts the inverse of the diagonal part of M_c and assembles $\widehat{\mathsf{S}}$; finally it performs the factorization. Note the `Eigen` methods used to compute $\widetilde{\mathsf{M}}_c^{-1}$: `diagonal` extracts the diagonal part of M_c as a vector, `asDiagonal` converts it to

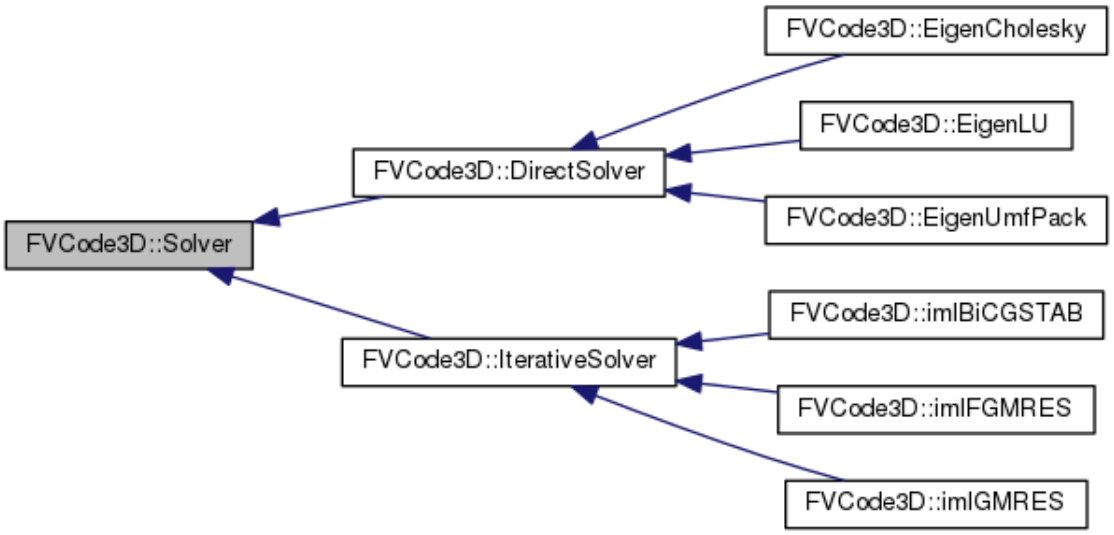


Figure 4.3: Inheritance graph for `Solver`.

a diagonal matrix and `inverse` gives the inverse diagonal matrix. The matrix $\widehat{\mathsf{S}}$ is factorized with the `Eigen` method `compute`. This method assembles the factor L that is stored inside the object `chol`.

We report the method `solve` that implements the steps of formula (3.18).

```
Vector BlockTriangular_preconditioner::solve(const Vector & r) const
{
    auto & B = *Bptr;
    // First step: solve Inexact Schur Complement linear system
    Vector y2 = chol.solve(r.tail(B.rows()));
    // Second step: solve the diagonal linear system
    Vector z(Md_inv.rows()+B.rows());
    z.head(Md_inv.rows()) = Md_inv*(r.head(Md_inv.rows())+B.transpose
        ()*y2);
    z.tail(B.rows()) = -y2;
    return z;
}
```

The first step involves the resolution of two triangular systems and it is performed with the method `solve` that takes in input the right-hand side vector. The other step involves only matrix-vector products.

We do not show the class `ILU_preconditioner`, because it is exactly equal to `BlockTriangular_preconditioner`, only the method `solve` changes in order to implement the formula (3.23). We avoid also the presentation of the class `HSS_preconditioner` that follows the same principles and implements the steps in (3.29).

4.4 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 abstract interface of all the possible solvers; its inheritance diagram is shown in Figure 4.3. `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 positive definite; it is available in the code for the case of fully FV scheme that produces a symmetric and positive definite matrix. The `IterativeSolver` has three children: `imlBiCGStab` implements the BiCGStab method through the version implemented in the `IML++` library, `imlGMRES` and `imlFGMRES` are the same with the GMRES and FGMRES methods.

4.4.1 The class `Solver`

Let us describe the base class `Solver`, which 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 which are both stored as `Vector`. Note that this class does not 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. If no argument is passed to the constructor, the empty one is called and it creates two vectors of null length. 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 have to resize also the matrix. Finally the method `solve` is pure virtual and it will be overridden to actually solve the linear system.

Remark 4.1. *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 polymorphism. We stress that the polymorphism 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.*

4.4.2 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 = 20000;
};
```

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` and, 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 20000 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.

The class `IterativeSolver` makes use of a preconditioner through the polymorphic object `preconPtr`, which 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`, employs the preconditioner factory to 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 $\widetilde{\mathsf{B}}$; 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 do not 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 the attributes `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.

4.4.3 The class `imlGMRES`

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

The `imlFGMRES` 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. This is

important to avoid an excessive memory consumption. See Section 3.1 of Chapter 3 for more details on this topic.

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 implementation see [66]. The library that we use has been fully updated for the Eigen framework, in order to avoid the overload of some matrix-vector operations otherwise required. 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 the vectors of the system (i.e. the right-hand side and the solution), and the `Preconditioner`, which is the class implementing the preconditioner. The `Matrix` class must supply the matrix-vector product and the preconditioner must supply a method solve for the solution of the system $\mathsf{P}\mathbf{z} = \mathbf{r}$. The arguments taken as input by the function 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 required tolerance `tol`. The GMRES returns 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` contains the number of iterations and `tol` the current residual error.

Let us note that in the `IML++` library the classical GMRES is implemented. The class `imlFGMRES` makes use of a template method that we have implemented following the same principles of the classical GMRES of the library. The algorithm of the FGMRES requires only a simple modification of the GMRES, in particular it stores the double of vectors. For more details on the FGMRES see [57].

For 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.

4.5 The class `Problem`

The upper level classes seen up to now are: the `SaddlePoint_StiffMatHandler`, which assembles the matrix of the system, and the `Solver` class, which stores the algebraic system and solves it. However these classes are not handled directly in

the main program of the software. We have implemented a class `Problem`, which makes use of the `SaddlePoint_StiffMatHandler` to assemble the system, handles the imposition of the source term modifying the right-hand side of the system and contains the solver, through which the linear system is actually solved. It is an abstract 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 `QRMatrix` and `QRFracture` are quadrature rules for the forcing term over the bulk and 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 we have two derived classes: `CentroidQuadrature`, which applies the midpoint rule in the polyhedron, and `Tetrahedralization`, which subdivides the polyhedron in tetrahedra and, in every tetrahedron, applies the mid point rule; this latter quadrature rule can be viewed as a composite mid point rule on the polyhedron. We stress the fact that, to have more accurate quadrature rules on a polyhedron, a tetrahedral partition, which allows to apply standard quadrature rules for tetrahedra, is an interesting approach.

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 `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, which 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. Regarding function, it is a class of the standard library that can wrap any kind of callable element (such as functions and function objects) into a copyable object 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;
    const int                         M_isSymUndef
    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 four attributes to indicate the numerical method, the solver policy, where the source term is applied and the nature of the saddle point matrix; this latter attribute is useful only if an iterative solver is used. The first three of these attributes are of particular types that are defined as 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 and 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 do not make sense and so they are deleted, there is 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`, which in short is a huge collector of all the data of the problem. Finally the destructor is virtual and defaulted.

Let us describe the methods `assemble` and `solve`. The `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. For this purpose it defines and uses the proper builder (for instance `SaddlePoint_StiffMat` in the case of an iterative solver for the MFD scheme). The method `assembleVector` has to set up the `Quadrature` object and to 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 is possible to get the matrix and the right-hand side through `M_solver`. Regarding this aspect, we stress that is not easy to get the matrix, because the storage format of the matrix is not known at compile time (indeed it depends on the type of solver). The problem is that `M_solver` is a pointer to the base `Solver` class, which stores the vector and returns it through an appropriate method, but it cannot return the matrix of the system because it does not 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 and 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 problem. 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 and in the following sections we give some practical information on the data input and the usage of the code.

4.6 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 is not smart to generate another mesh every time a change in the properties (for instance in the permeability) 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 file name;
- **output**: the output directory and file name;
- **problem parameter**: the parameters 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,HSS
```

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 centred 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.;};
```

4.7 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 is necessary to handle some classes that have not been described. For example, to import the mesh an importer object is needed, while to export the results of the computation an exporter object must be defined. We do not focus on these classes, but we show a practical example of usage.

The classes required to use 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.

Preliminarily we define two `typedef` 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 [65].

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 and with polymorphism it is easy to select the correct importer. In any case in the example we use the importer for the mesh file of `.fvg` extension.

After reading the mesh, it is 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 permeability 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);
myrmesh.BuildEdges();
myrmesh.BuildOrientationFacets();
```

`BuildEdges` and `BuildOrientationFacets` perform some operations to build the data structures of the edges and to compute the orientations of the facets.

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 elements corresponding to the total number of cells (bulk plus fractures).

Finally we delete the instances:

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

Chapter 5

Numerical results

In this chapter we present several numerical experiments to test the discrete model and the selected preconditioners. In particular, in the first test case, a single fracture configuration designed to have an analytical solution, we verify numerically the theoretical convergence order of the MFD for the bulk pressure and velocity and of FV for the fracture pressure. Then, we present three tests to analyse the preconditioners presented in Section 3.4 of Chapter 3. We study their dependence on the mesh size, the fracture model parameters and the coupling coefficient in a single fracture case, and, in two more realistic cases of network of fractures with several intersections, we study them with respect to the fracture model parameters. Some other tests are devoted to the numerical study of the spectral properties of the system, always in the case of a single fracture. In particular we describe a preliminary test to study the spectral properties of the transmissibility matrix of FV-TPFA, then we verify numerically the condition number estimate for the matrix of the system presented in Section 3.5 of Chapter 3. For the 3D visualization of pressure in bulk and fractures and for the construction of the graphics `Paraview` [69] and `Matlab` [67] have been used, respectively.

Regarding the mesh, throughout the chapter, different kinds of grids, always conforming to the fractures, have been used: hexahedral Cartesian meshes (only in very simplified cases), tetrahedral and generic polyhedral meshes, which are not Voronoi diagrams and may contain also non-convex elements. The tetrahedral grids have been generated with `TetGen` [70], through which a tetrahedralization constrained to the fractures can be performed. Whereas polyhedral grids have been generated by converting constrained tetrahedral meshes in their dual meshes of polyhedra. This step has been carried out with `OpenFOAM` [68], after having applied a patch to the library to take into account the conformity to the fractures in the dualization procedure. We stress that about the mesh generation no work has been carried out in this thesis, which focuses on the mimetic implementation and especially on the development of suitable preconditioners; for further details regarding the mesh generation see [62].

5.1 Verification of the order of convergence

This test is a three-dimensional generalization of the one considered in [7] 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 at $y = 0$. 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-fracture permeability contrast that the code has to reproduce. We take as forcing term the following function [7]:

$$f(x,y,z) = \begin{cases} (1-k_f) cosh\Big(\frac{l_\Gamma}{2}\Big) cos(x) cos(z) & in\ \Omega \\ k_f^2 cos(x) + k_f(1-k_f) cosh\Big(\frac{l_\Gamma}{2}\Big) cos(x) cos(z) & in\ \Gamma, \end{cases} \tag{5.1}$$

so that the exact solution is given by

$$p(x,y,z) = \begin{cases} k_f cos(x) cosh(y) + \dfrac{1-k_f}{2} cosh\Big(\frac{l_\Gamma}{2}\Big) cos(x) cos(z) & in\ \Omega \\ k_f cos(x) + \dfrac{1-k_f}{2} cosh\Big(\frac{l_\Gamma}{2}\Big) cos(x) cos(z) & in\ \Gamma. \end{cases} \tag{5.2}$$

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 analogously.

Let us note that, in spite of the permeability discontinuity across Γ, the test case is built to preserve the continuity of the pressure p and of the normal component of velocity $\mathbf{u} \cdot \mathbf{n}_\Gamma$ across Γ. More precisely, if we compute the normal component of the velocity $\mathbf{u} = -\nabla p$ across the fracture, we get:

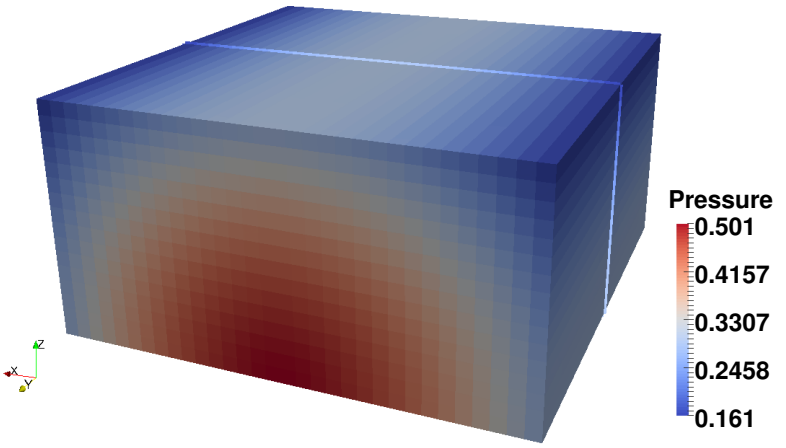
$$\mathbf{u} \cdot \mathbf{n}_\Gamma|_\Gamma = -\nabla p \cdot \mathbf{n}|_\Gamma = -\left.\frac{\partial p}{\partial y}\right|_\Gamma = -\, k_f cos(x) sinh(y)|_{\{y=0^\pm\}} = 0.$$

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

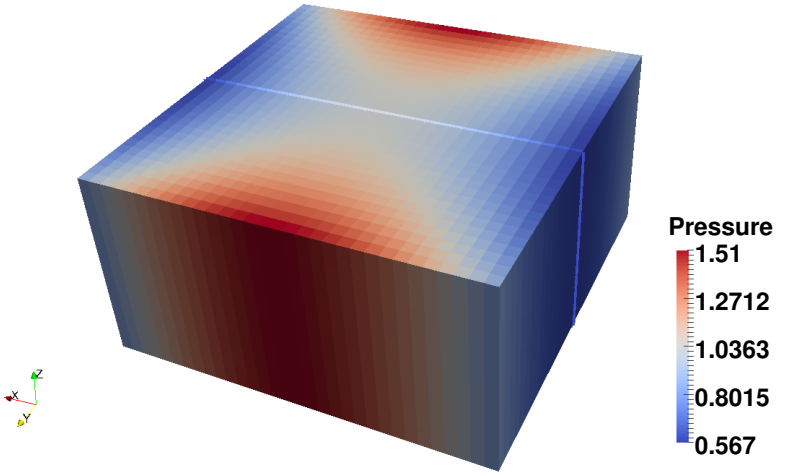
We consider different kinds of grids with different refinements to study the convergence order of the numerical method. The Table 5.1 reports the diameters of the grids used in the convergence test.

In terms of fracture permeability we consider $k_f = 10^{-3}, 1, 10^3$, while the aperture is kept fixed as $l_\Gamma = 10^{-2}$. The numerical solution over the hexahedral mesh with $h = 0.0625$ is shown in Figure 5.1 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 independent from the z coordinate.

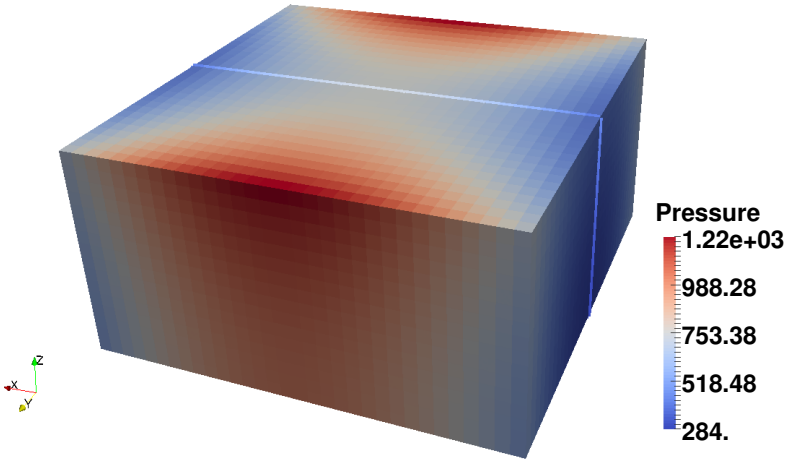
Let us now focus on the study of the order of convergence. In [14] the order of convergence of the MFD applied to a diffusion problem in mixed form has been widely studied: a first order has been proven for both pressure and velocity; moreover



(a) $k_f = 10^{-3}$



(b) $k_f = 1$



(c) $k_f = 10^3$

Figure 5.1: Discrete pressure in bulk by varying the fracture permeability k_f.

	Hexahedral	Tetrahedral	Polyhedral
h_1	0.25	1.118	0.559
h_2	0.125	0.563	0.335
h_3	0.0625	0.314	0.244
h_4	0.03125	0.144	0.146

Table 5.1: Diameters of the grids used in the convergence test.

k_f	Conv. rate p_h	Conv. rate $p_{\Gamma,h}$	Conv. rate $\mathbf{u}_h$
10^{-3}	1.99	1.96	1.81
1	1.94	1.97	1.85
10^3	1.96	1.93	1.82

(a) Hexahedral (Cartesian)

k_f	Conv. rate p_h	Conv. rate $p_{\Gamma,h}$	Conv. rate $\mathbf{u}_h$
10^{-3}	1.96	1.98	1.02
1	1.99	1.97	1.15
10^3	1.74	0.97	1.01

(b) Tetrahedral

k_f	Conv. rate p_h	Conv. rate $p_{\Gamma,h}$	Conv. rate $\mathbf{u}_h$
10^{-3}	1.94	1.93	0.91
1	1.87	1.99	0.95
10^3	1.77	1.40	0.94

(c) Polyhedral

Table 5.2: Mean convergence rate by varying the fracture permeability k_f for different mesh types.

for the pressure a result of super-convergence, of quadratic order in particular, has been proven too. Also the TPFA version of the finite volume scheme would show a quadratic order of convergence [31], provided that the K-orthogonality condition is fulfilled, because otherwise the convergence of the scheme is not guaranteed. So we expect quadratic order of convergence for the pressure in bulk and fracture and first order for velocity.

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 2.2 in Chapter 2). The formula is

$$||q_h||^2_{Q_h} = \sum_{\mathsf{P}\in\Omega_h} |\mathsf{P}| q^2_{\mathsf{P}} \qquad \forall\ q_h \in Q_h. \tag{5.3}$$

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

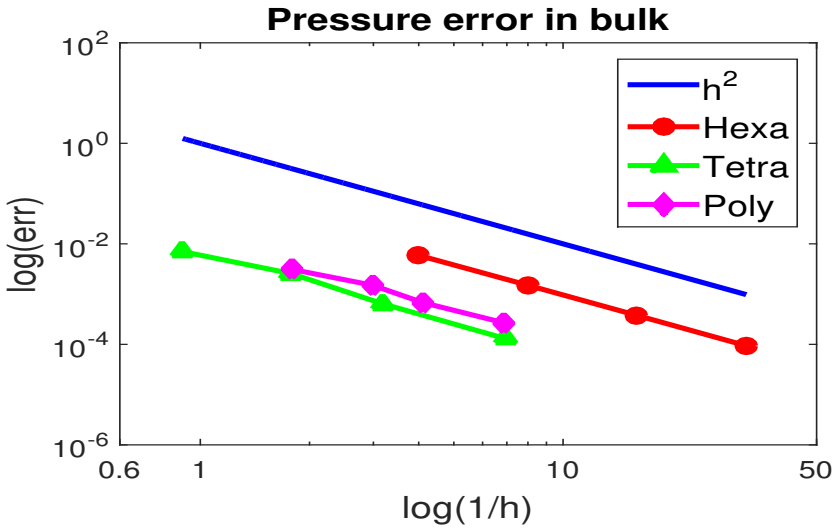
$$||\mathbf{v}_h||^2_{W_h} = \mathbf{v}_h^{\mathsf{T}} \mathsf{M} \mathbf{v}_h \qquad \forall\ \mathbf{v}_h \in W_h, \tag{5.4}$$

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

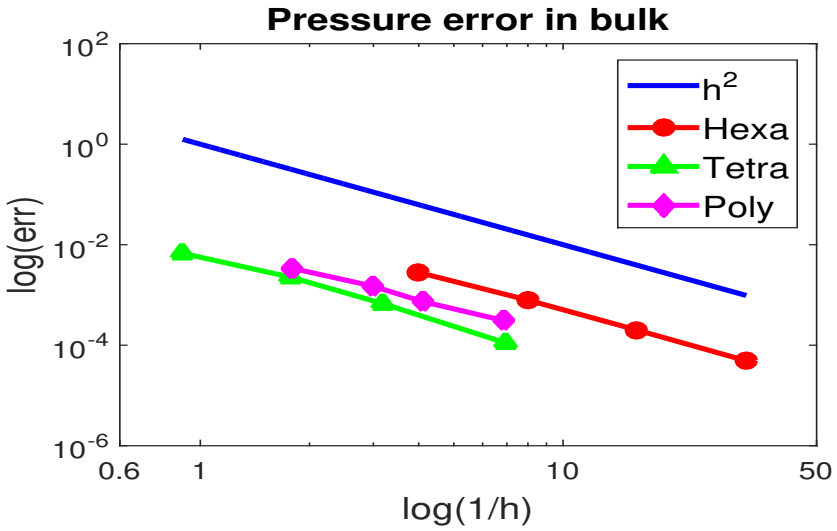
$$err_p = \frac{||p^I - p_h||_{Q_h}}{||p^I||_{Q_h}}, \qquad err_u = \frac{||\mathbf{u}^I - \mathbf{u}_h||_{W_h}}{||\mathbf{u}^I||_{W_h}}, \tag{5.5}$$

where the apex I indicates the projected values of the exact solution on Q_h and W_h (see Section 2.2 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 we add N_Γ elements at the tail of $\mathbf{u}^I$.

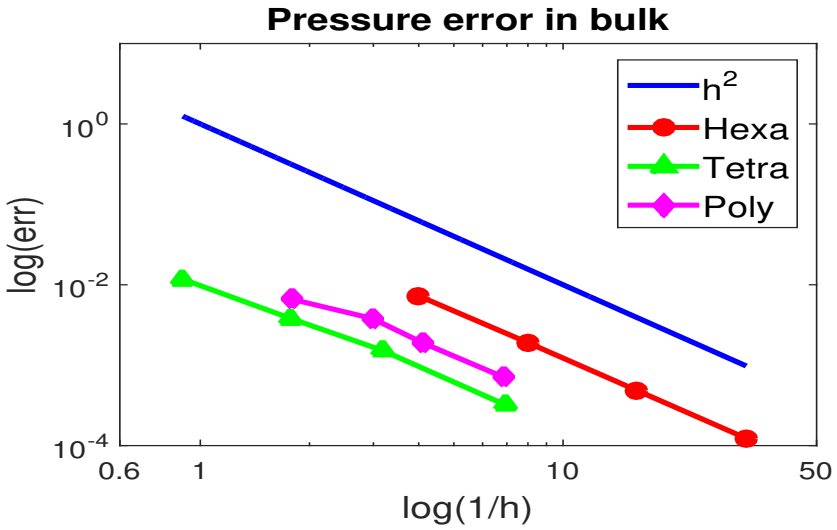
In Table 5.2 the mean convergence order for the three types of mesh and for various fracture permeabilities k_f are reported. Regarding the hexahedral case, a second order of convergence is clearly achieved for both the pressure in the porous medium and the pressure in the fracture; so the expected orders of convergence are verified. The estimated velocity order in the three permeability cases is about 1.8, so a super-convergence effect is observed also for the velocity in the case of hexahedral grids. Also with tetrahedral and polyhedral grids the results are in line with the theory of the MFD, because a second order is achieved for the pressure in bulk and a first order for the velocity. Regarding the FV in the fracture, a second order of fracture pressure 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 $\mathsf{K}_\Gamma = k_f\mathsf{I}$ is in our case, this condition is surely fulfilled in a case of a Cartesian grid and not in a case of an unstructured triangular/polygonal grid (indeed a tetrahedral/polyhedral mesh over the bulk induces a triangular/polygonal mesh over the fracture). To have a graphical verification of the convergence orders see Figure 5.2, 5.3 and 5.4, where the relative errors as a function of the mesh size by varying the fracture permeability k_f is reported. A slight underestimate of the convergence rate for the fracture pressure in the high permeable fracture case for tetrahedral and polyhedral meshes as well as the super-convergence of velocity in the hexahedral case are quite evident.



(a) $k_f = 10^{-3}$

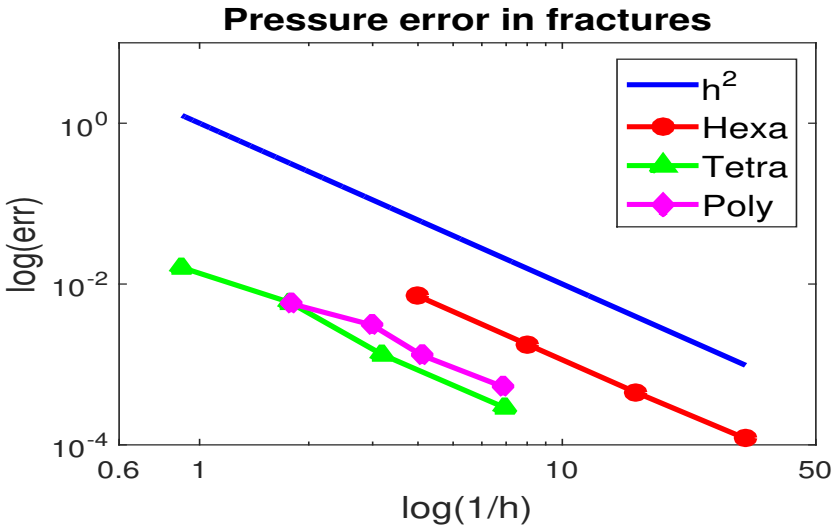


(b) $k_f = 1$

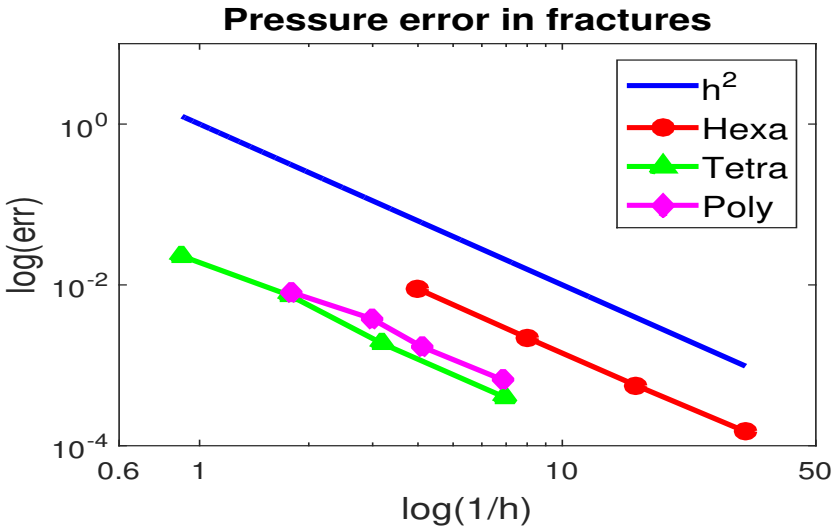


(c) $k_f = 10^3$

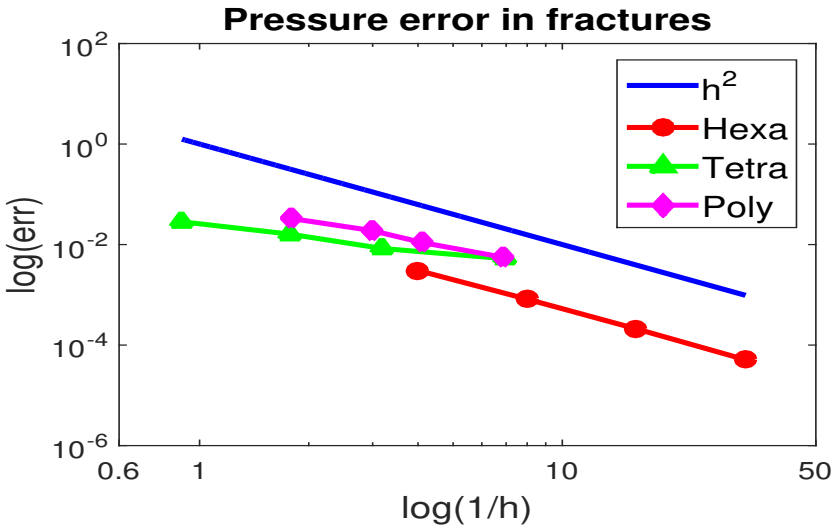
Figure 5.2: Relative errors of bulk pressure as a function of the mesh size (loglog scale) by varying the fracture permeability $k_f = 10^{-3}, 1, 10^3$. Hexa, Tetra, Poly stand for hexahedral (Cartesian), tetrahedral, polyhedral grids, respectively.



(a) $k_f = 10^{-3}$

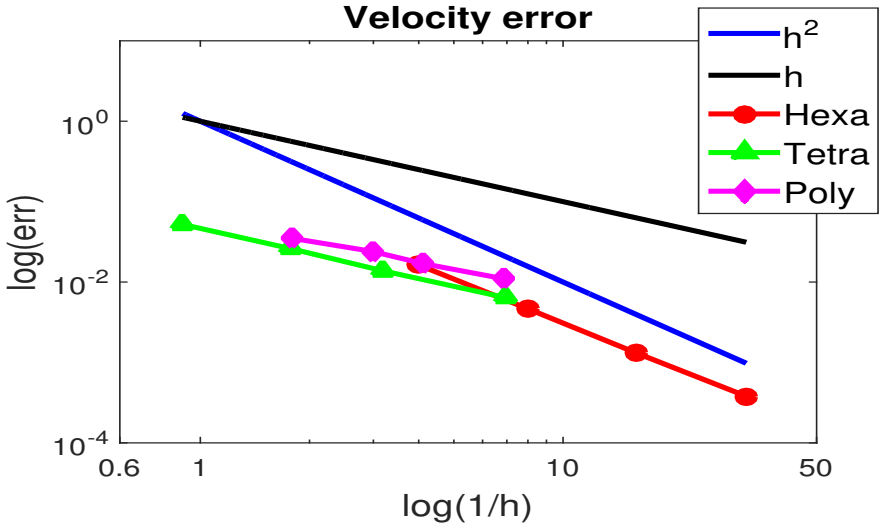


(b) $k_f = 1$

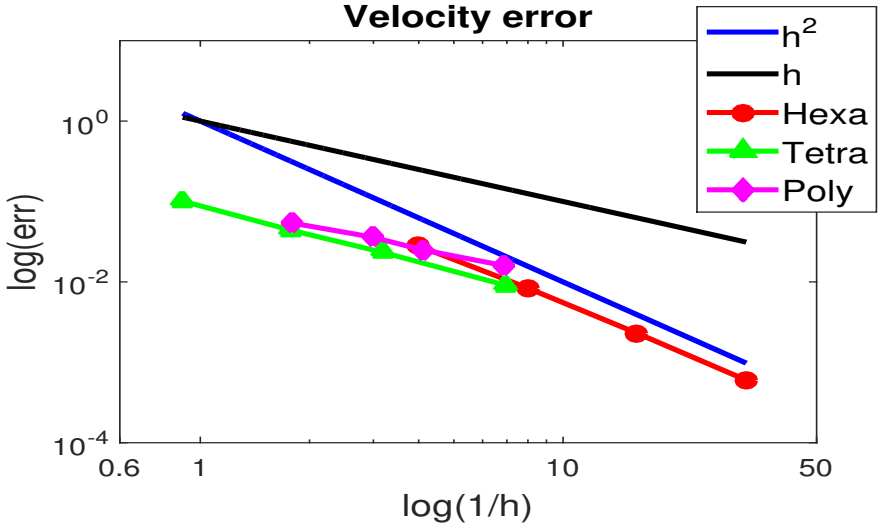


(c) $k_f = 10^3$

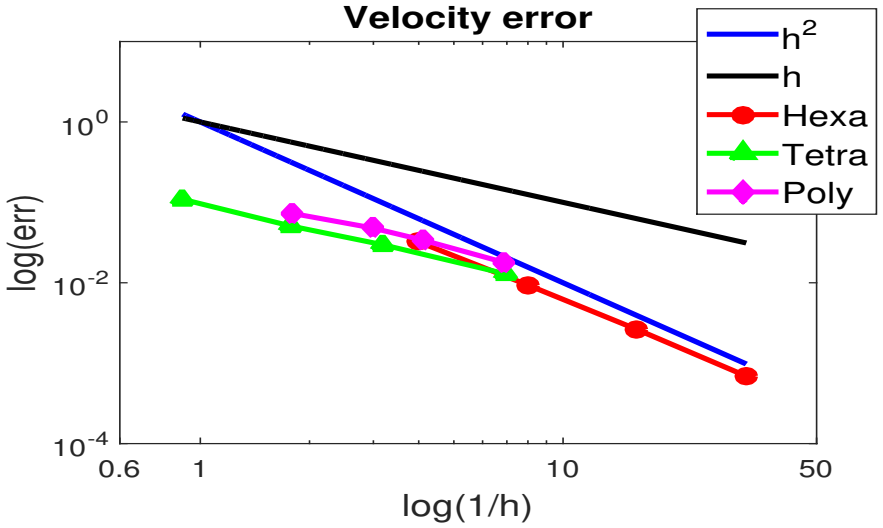
Figure 5.3: Relative errors of fracture pressure as a function of the mesh size (loglog scale) by varying the fracture permeability $k_f = 10^{-3}, 1, 10^3$. Hexa, Tetra, Poly stand for hexahedral (Cartesian), tetrahedral, polyhedral grids, respectively.



(a) $k_f = 10^{-3}$



(b) $k_f = 1$



(c) $k_f = 10^3$

Figure 5.4: Relative errors of velocity as a function of the mesh size (loglog scale) by varying the fracture permeability $k_f = 10^{-3}, 1, 10^3$. Hexa, Tetra, Poly stand for hexahedral (Cartesian), tetrahedral, polyhedral grids, respectively.

5.2 Testing the preconditioners

In this section we present three numerical tests in which we study the preconditioners introduced in Section 3.4 of Chapter 3. The first example deals with a case of a single fracture that cuts the whole domain, whereas the second and the third deal with cases of networks of fractures with several intersections and are more realistic and challenging. The linear system is solved with the restarted GMRES [58] for all the preconditioners except the HSS, for which the restarted FGMRES [57] is used to take into account the preconditioner variations due to the CG inner cycle. For the GMRES the maximum number of iterations is set to 20000, the tolerance is fixed to 10^{-6} and the restart level is set to 300 in all simulations. We report the number of iterations to reach convergence and the computed residual; also the computing time is reported to make a comparison between iterative preconditioned solvers and the UmfPack direct method [29]. This is important to have an idea of the efficiency of a preconditioner, because the goal of preconditioning is to make the software capable of solving iteratively large linear systems, which arise from 3D problems, avoiding the usage of direct solvers, which becomes unfeasible in 3D mainly because of memory limitations.

The fundamental goal of the analyses is the study of the sensitivity of the preconditioners with respect to the parameters of the fracture model. The bulk permeability is assumed to be $\mathsf{K} = \mathsf{I}$, while for the fractures we assume a permeability tensor given by the equation (1.3), with $\mathsf{K}_\Gamma^\tau = k_\Gamma^\tau \mathsf{I}$, so that the permeability in the fractures is completely determined by k_Γ^τ and k_Γ^n. The fracture parameters that govern the reduced model are $k_\Gamma^\tau l_\Gamma$ and k_Γ^n / l_Γ; they determine the interaction between the bulk and the fractures (see Remark 1.1 in Chapter 1) and we let vary them instead of the permeability and the aperture of the fractures singularly. We adopt the following notations for these two equivalent permeabilities: $k_\tau^* = k_\Gamma^\tau l_\Gamma$ and $k_n^* = k_\Gamma^n / l_\Gamma$. In all the three cases we make a study with respect to k_τ^* and k_n^*, whereas only in the single fracture case, which is much simpler, we present also a study by varying the mesh size and an analysis on the coupling parameter ξ. In all the cases where the coupling coefficient is considered fixed it is prescribed to the optimal value $\xi = 0.75$. For the meshes we only point out that all the tests in this section are performed with generic polyhedral grids.

We remind that the implementation of the block triangular and block ILU preconditioners requires the resolution of a linear system with matrix $-\widehat{\mathsf{S}}$ (see steps (3.18) and (3.23)). This matrix is pre-factorized with a sparse Cholesky factorization and a couple of upper and lower triangular systems is solved for every iteration of GMRES. Whereas the implementation of the HSS preconditioner requires the resolution of three linear systems (see steps (3.29)). The system with matrix $\mathsf{M}_c + \alpha\mathsf{I}$ is solved with a CG inner cycle. Regarding the CG, we have fixed a maximum number of iterations of 300 and a tolerance of 10^{-2}. The other two systems with matrices $\mathsf{T} + \alpha\mathsf{I}$ and $\widetilde{\mathsf{B}}\widetilde{\mathsf{B}}^\mathsf{T} + \alpha^2\mathsf{I}$ are solved with a sparse Cholesky pre-factorization.

5.2.1 The single fracture test case

Here we present a case of a single fracture that cuts the whole porous medium. The domain configuration is the same of the test shown in the Section 5.1 of this

	h=0.559		h=0.282		h=0.157	
	It	Res	It	Res	It	Res
Diag	575	8.98e-07	4086	9.95e-07	8074	9.98e-07
BlockTr_D	21	8.09e-07	32	7.47e-07	33	8.30e-07
ILU_D	14	6.24e-07	24	9.89e-07	27	9.77e-07
BlockTr_L	141	8.93e-07	X	X	X	X
ILU_L	39	9.09e-07	149	8.17e-07	1684	9.98e-07
HSS	37	8.08e-07	66	9.22e-07	108	9.04e-07

(a) Number of iterations for convergence and computed residual.

DOF	4100	17220	87500
UmfPack	0.06	2.75	204.19
Diag	0.83	31.07	395.28
BlockTr_D	0.019	0.14	1.15
ILU_D	0.017	0.12	1.08
BlockTr_L	0.33	X	X
ILU_L	0.23	5.83	255.34
HSS	0.056	0.83	6.92

(b) Computing time [sec].

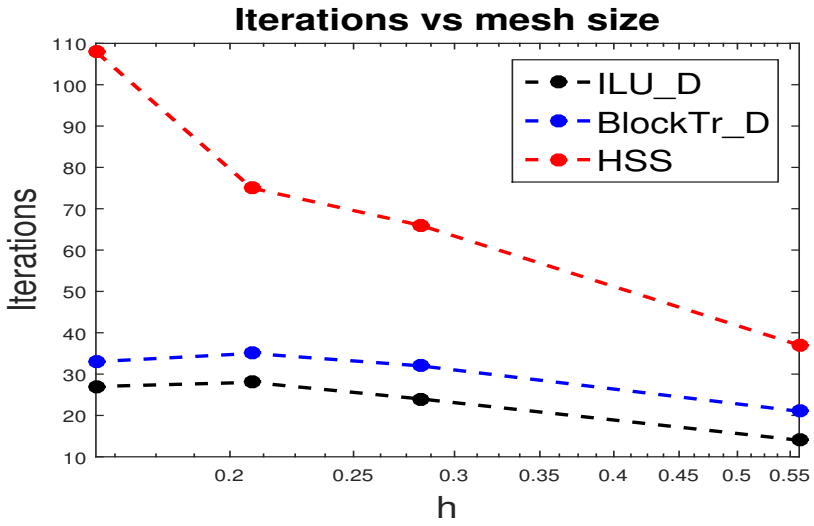
Table 5.3: (a) Number of iterations to reach convergence of GMRES and computed residual for different preconditioners and by varying the mesh size. (b) Computing time to solve the system with UmfPack and GMRES with different preconditioners as a function of the total number of degrees of freedom. Diag means the diagonal part of the whole system; BlockTr_D and ILU_D refer to the block triangular and the block ILU preconditioners based on the diagonal part of M_c; BlockTr_L and ILU_L are the block triangular and the block ILU preconditioners based on the lumping of M_c; HSS refers to the Hermitian and skew-Hermitian splitting preconditioner. X means that the GMRES has not reached convergence in the prescribed number of iterations. We have set: MaxIt = 20000, tol = 10^{-6}, restart = 300. Remind that for HSS the FGMRES has been used.

chapter. More precisely the domain is $\Omega = (-1,1)\times(-1,1)\times(0,1)$ and the fracture is the vertical plane $\Gamma = (-1,1)\times\{0\}\times(0,1)$. On the left and right boundary sides we consider Dirichlet conditions, in particular we fix $p = 1$ on $\{y = -1\}$ and $p = 0$ on $\{y = 1\}$, while on the top, bottom, front and back sides we impose homogeneous Neumann boundary conditions. We consider different boundary conditions on the top and bottom boundary sides of the fracture: we fix the value of pressure to $p = 1$ on the top and to $p = 0$ on the bottom side. In this way we are simulating two orthogonal flows, one in the bulk and one in the fracture, and, by varying the fracture parameters, we study their interaction.

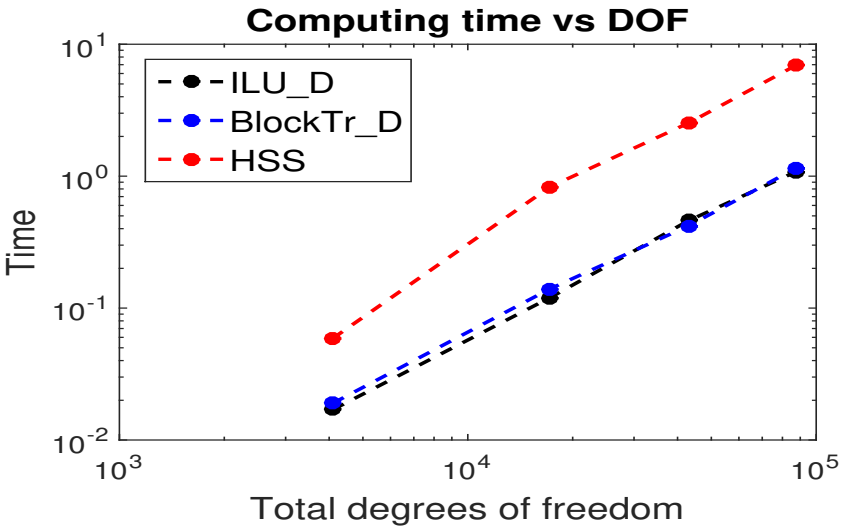
A study on the mesh size

First of all we present a test for all the preconditioners introduced in Section 3.4 of Chapter 3. In particular we perform a study with respect to the mesh size h. We fix the parameters to have the simplest case, i.e. $k_\tau^* = 10^{-2}$ and $k_n^* = 10^2$, which means no pressure jump and no velocity jump, in particular a situation in which the presence of the fracture does not have a significant effect on the flow in

the porous medium. More complex situations with sealed or conductive fracture will be considered later.



(a) Iterations as a function of the mesh size (logarithmic x-axis).



(b) Computing time as a function of the total number of degrees of freedom (loglog).

Figure 5.5: (a) Number of iterations to reach convergence of GMRES as a function of the mesh size for different preconditioners. (b) Computing time of GMRES as a function of the total number of degrees of freedom for different preconditioners.

The number of iterations to reach convergence and the computed residual are reported in (a) of Table 5.3, while the computing time is shown in (b) of Table 5.3. Note that solving the system with a trivial preconditioner, like the diagonal part of the matrix of the system (Diag in the table), is very inefficient and the computing times are always greater than solving the problem by the direct solver UmfPack. Whereas BlockTr_D, ILU_D and HSS provide a significant improvement of performances with respect to UmfPack. Observe that in the most refined case we pass from 204.19 seconds with UmfPack to approximately 1 second with GMRES and BlockTr_D and ILU_D and to 6.92 seconds with the HSS preconditioner. We

stress the fact that, if a block triangular or an ILU preconditioner is employed, only the diagonal part of M_c provides good results: using the lumping of M_c is practically unfeasible for the block triangular case and clearly inefficient for the ILU case. Note that the number of iterations for BlockTr_D and ILU_D is affected by mesh refinement but the dependence is moderate. Indeed we have a significant increase of iterations only for the first halving of h. Regarding the HSS, numerical experiments have shown that the optimal value of the α coefficient that minimizes the number of iterations of FGMRES depends on the mesh size. In Table 5.4 the values calibrated for α and used in this test are reported. Even with the optimal α, solving more refined problems requires more iterations (we pass from 37 iterations for $h = 0.559$ to 108 iterations for $h = 0.157$), so that the HSS preconditioner is significantly dependent on the mesh size.

In Figure 5.5 we have a graphical visualization of iterations and computing times as a function of h and of the total number of degrees of freedom, respectively. In both the graphs it's evident that BlocTr_D and ILU_D converge faster providing better performances than the HSS preconditioner. Note in (a) the slight dependence of BlockTr_D and ILU_D on the mesh size, indeed a stabilization of the iterations around 30 can be observed for both the preconditioners. In (b) a power dependence (estimated around 1.4) of the computing time on the total number of degrees of freedom can be observed for all the preconditioners; moreover we see that BlockTr_D and ILU_D provide about the same computing times.

Finally it is interesting to note that with BlockTr_D, ILU_D and HSS we have no restart of the GMRES, because it converges always in less than 300 iterations; this is important because the restart slows down much the convergence of the iterative method. In particular we can expect that with BlockTr_D and ILU_D the restart never happens because of their good scaling with respect to the mesh size. To have an idea of the pressure field in the bulk domain see (b) of Figure 5.7.

h	0.559	0.282	0.157
α	1e-2	1e-3	1e-3

Table 5.4: Optimal α coefficient of HSS depending on the mesh size h.

A study on the fracture model parameters

Now we present a study on the robustness of the preconditioner with respect to the parameters of the fracture model k_τ^* and k_n^*. We consider three cases: $k_\tau^* = 10^{-2}$, $k_n^* = 1$, which means a sealed fracture and a consequent non zero pressure jump; $k_\tau^* = 10^{-2}$, $k_n^* = 10^2$, which is simply the previous test where the presence of the fracture does not influence the flow in the porous medium; $k_\tau^* = 1$, $k_n^* = 10^2$, which is a case of a conductive fracture, i.e. the fracture represents a preferential path for the flow and we expect a non zero velocity jump. Hereafter we do not consider the two preconditioners based on the lumping of M_c, we keep only BlockTr_D, ILU_D and HSS, which provided a good performance in the previous test, and the Diag preconditioner to have a trivial preconditioner as a reference. For this test a polyhedral mesh of size $h = 0.208$ has been employed; the corresponding dimension of the linear system is 43031.

	k_τ^*=1e-2, k_n^*=1		k_τ^*=1e-2, k_n^*=1e2		k_τ^*=1, k_n^*=1e2	
	It	Res	It	Res	It	Res
Diag	8239	9.98e-07	8301	9.99e-07	2263	9.95e-07
BlockTr_D	34	8.89e-07	35	7.99e-07	36	7.81e-07
ILU_D	27	8.40e-07	28	6.72e-07	28	7.94e-07
HSS(1e-3)	44	7.71e-07	75	9.94e-07	125	8.67e-07

(a) Number of iterations for convergence and computed residual.

	k_τ^*=1e-2, k_n^*=1	k_τ^*=1e-2, k_n^*=1e2	k_τ^*=1, k_n^*=1e2
UmfPack	27.67	27.85	26.75
Diag	179.46	180.26	48.21
BlockTr_D	0.45	0.46	0.47
ILU_D	0.41	0.42	0.42
HSS(1e-3)	1.48	2.51	4.29

(b) Computing time [sec].

Table 5.5: (a) Number of iterations to reach convergence and computed residual of GMRES for different preconditioners and by varying the model parameters. (b) Computing time to solve the system with UmfPack and GMRES for different preconditioners and by varying the model parameters. Diag means the diagonal part of the whole system; BlockTr_D and ILU_D refer to the block triangular and the block ILU preconditioners based on the diagonal part of M_c; HSS(α) refers to the Hermitian and skew-Hermitian splitting preconditioner with α coefficient. X means that the GMRES has not reached convergence in the prescribed number of iterations. For the GMRES we have set: MaxIt = 20000, tol = 10^{-6}, restart = 300. Remind that for HSS the FGMRES has been used. The model parameters are $k_\tau^* = k_\Gamma^\tau l_\Gamma$ and $k_n^* = k_\Gamma^n / l_\Gamma$.

The number of iterations to reach convergence and the computed residual are reported in (a) of Table 5.5, while the computing time is shown in (b) of Table 5.5. Note that the preconditioners BlockTr_D and ILU_D show a good behaviour with respect to the parameters of the fracture model. Indeed we have an increase of only one/two iterations passing from the case of a sealed fracture to the case of a conductive fracture. So we can talk about optimality of these two preconditioners with respect to the parameters of the fracture model. This does not hold for the HSS which, although it represents a valid alternative to a direct solver (4.29 against 26.75 seconds in the worst case), shows a strong dependency on the parameters of the problem. Numerical experiments have shown that the optimal α of HSS that minimizes the number of iterations of FGMRES does not depend on the fracture parameters, in this test a value of 10^{-3} has been calibrated. It is interesting to note that different preconditioners may have a different sensitivity on the parameters. The Diag preconditioner, even if in general not effective, appears to behave better with the highly conductive case (the third case in Table 5.5). While the HSS preconditioner shows the opposite behaviour. Clearly solving the linear system with a direct solver is not influenced anyway from the parameters and the computing times are always approximately the same.

The pressure field in the bulk domain for the three cases that we have analysed is reported in Figure 5.7. Note the jump of pressure across the fracture in the case of a sealed fracture (the (a) image) and the linear pressure profile in the case

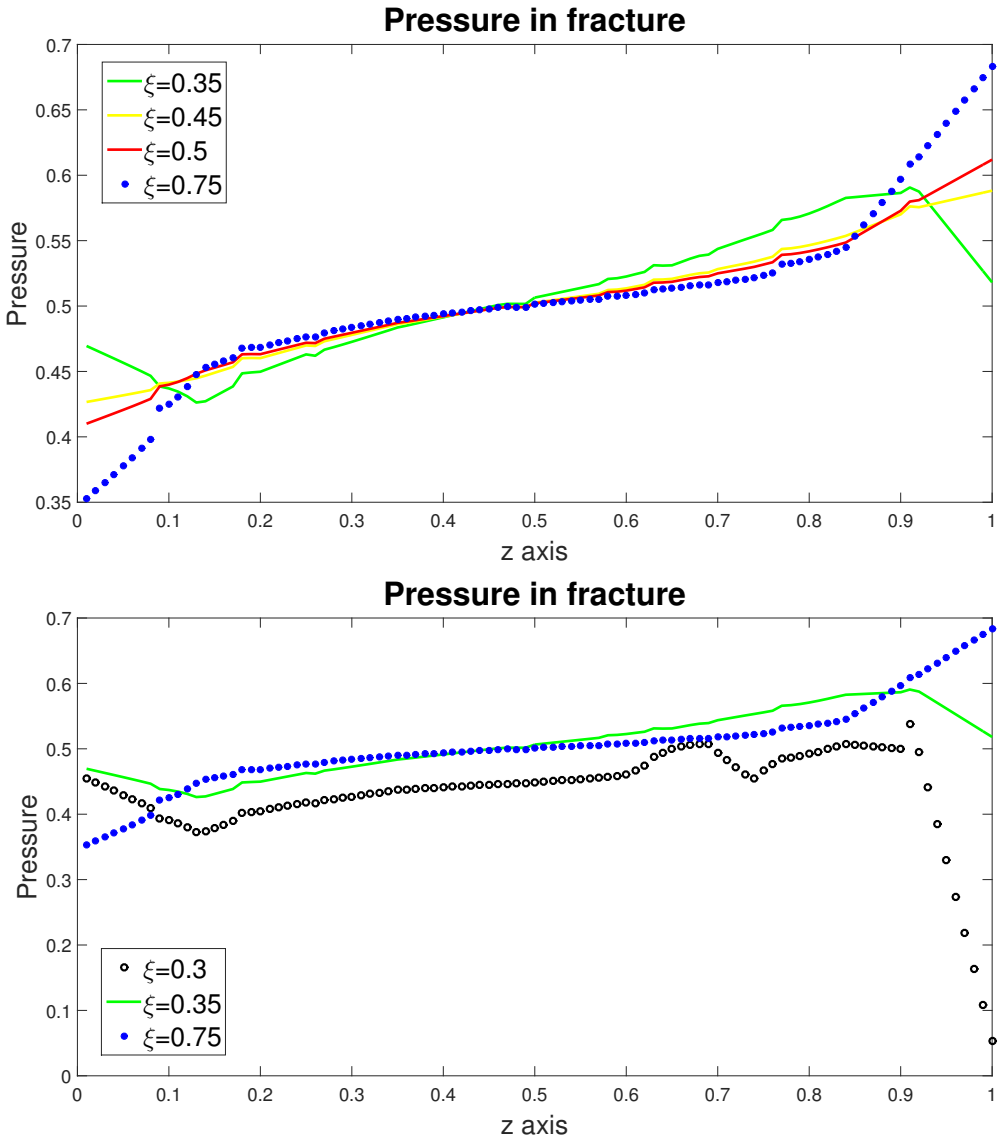


Figure 5.6: Pressure in fracture on z axis for different values of ξ. The values have been obtained after a proper interpolation with `Paraview`.

(b) where the fracture has a negligible effect on the bulk flow. More interesting is the case of a conductive fracture (the (c) image). In this case we have interaction between the fracture flow and the bulk one and the fracture becomes a preferential path for the flow. Indeed we have a significant over-pressure zone near the top of the fracture and a significant under-pressure zone near the bottom of the fracture. See [52] for a nice 2D visualisation of the velocity field in the bulk and in the fracture; indeed our test in the case of conductive fracture is the three-dimensional counterpart of the one presented in [52].

A study on the coupling coefficient

Now we study the influence of the parameter ξ on the numerical solution and on the preconditioners. We fix the parameters $k_\tau^* = 10^{-2}, k_n^* = 1$, i.e. we consider the

	BlockTr_D		ILU_D		HSS(1e-3)	
ξ	It	Res	It	Res	It	Res
1	34	8.88e-07	27	8.43e-07	37	8.30e-07
0.75	34	8.89e-07	27	8.40e-07	44	7.71e-07
0.5	38	8.75e-07	27	8.63e-07	63	9.48e-07
0.45	42	8.45e-07	27	9.01e-07	73	9.52e-07
0.35	71	8.69e-07	38	8.90e-07	194	9.82e-07
0.3	X	X	7817	9.99e-07	X	X

Table 5.6: Number of iterations to reach convergence and computed residual of GMRES for different preconditioners and by varying the coupling parameter ξ. BlockTr_D and ILU_D refer to the block triangular and the block ILU preconditioners based on the diagonal part of M_c; HSS(α) refers to the Hermitian and skew-Hermitian splitting preconditioner with α coefficient. X means that the GMRES has not reached convergence in the prescribed number of iterations. For the GMRES we have set: MaxIt = 20000, tol = 10^{-6}, restart = 300. Remind that for HSS the FGMRES has been used.

case of fracture that behaves like a barrier, because the coupling coefficient plays an important role especially when the pressure jump is non zero. The mesh is again the same of the previous analysis, i.e. $h = 0.208$. We know from the derivation of the model for the fracture that $\xi \in [0, 1]$, moreover to prove the well-posedness of the problem the additional constraint $\xi \in (0.5, 1]$ is required. We have considered two values of ξ (1 and 0.75) in the interval $(0.5, 1]$, two values (0.5 and 0.45) near to the lower bound of the interval and other two values (0.35 and 0.3) significantly far away from the lower bound 0.5.

The number of iterations to reach convergence and the computed residual are listed in Table 5.6. Note that, as long as $\xi \in (0.5, 1]$, the number of iterations to reach convergence for Block_D and ILU_D does not change, a slight increase for HSS is observed. For $\xi = 0.5, 0.45$, i.e. near the lower bound of the interval, there is no significantly changes in the convergence of GMRES for Block_D and ILU_D, while a more considerable increase of iterations for HSS is registered. If ξ get away from the interval, in the case $\xi = 0.35$ the iterations increase for all the preconditioners, while in the case $\xi = 0.3$ the number of iterations enlarges a lot, so that with BlockTr_D and HSS the convergence is not reached and with ILU_D 7817 iterations are needed.

The numerical experiments do not show a dramatic behaviour of the iterative solver for values of ξ outside $(0.5, 1]$, but there is a threshold value for which the conditioning of the system worsens a lot and solving the problem iteratively becomes hardly. We stress the fact that using values near 0.5, for which the iterative method performs well, does not mean that the pressure is correctly reproduced. Regarding this, see Figure 5.6 that shows the pressure in fracture on the z axis, i.e. on the line $(x, y, z) = (0, 0, z),\ 0 \leq z \leq 1$. The solution in the fracture depends only on z, so we can make considerations on its values on the z axis without loss of generality. Note that the discrete pressure shows a correct behaviour at the boundaries only in the case $\xi = 0.75$, where we see a significant increase of pressure near $z = 1$ and decrease near $z = 0$. For $\xi = 0.5, 0.45$ we can observe a "flattening" of the profile at the boundaries so that the boundary conditions are not well reproduced.

Then for $\xi = 0.35$ we start seeing oscillations at the boundaries and the numerical solution is not stable. In the bottom graph of Figure 5.6 the pressure for $\xi = 0.3$ is reported; in this case we see that the pressure profile is completely wrong with a huge oscillation near $z = 1$.

The simulations confirm the theoretical bound $\xi \in (0.5, 1]$ and the fact that $\xi = 0.75$ is optimal. We note that in the discrete setting one may prove well-posedness also for values of ξ smaller than 0.5, with a lower bound that depends on the mesh size, and which tends to 0.5 when $h \to 0^+$, see [34].

5.2.2 A first case of network of fractures

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 Γ consists of seven fractures with several intersections; the bulk domain and the network are shown in Figure 5.8. 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 homogeneous Neumann boundary conditions. A polyhedral mesh of diameter $h = 0.193$ is employed and the dimension of the system is 43834.

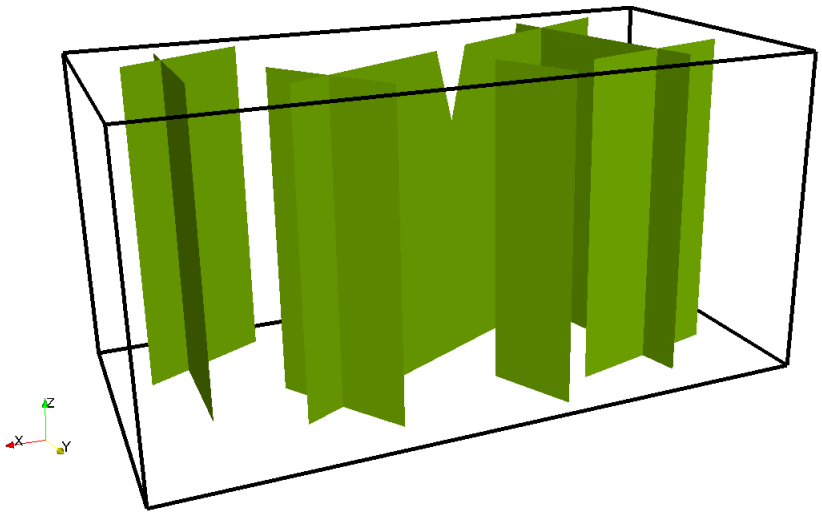
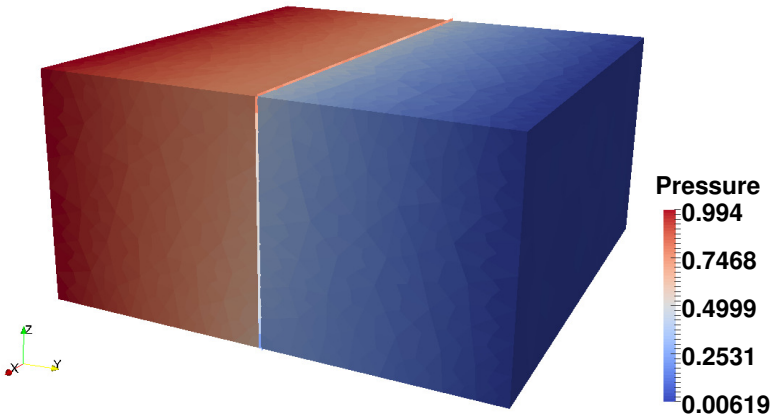
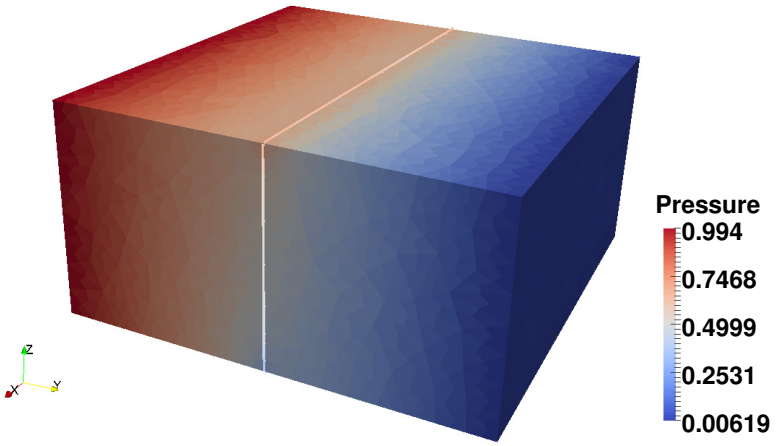


Figure 5.8: Porous medium with network of fractures.

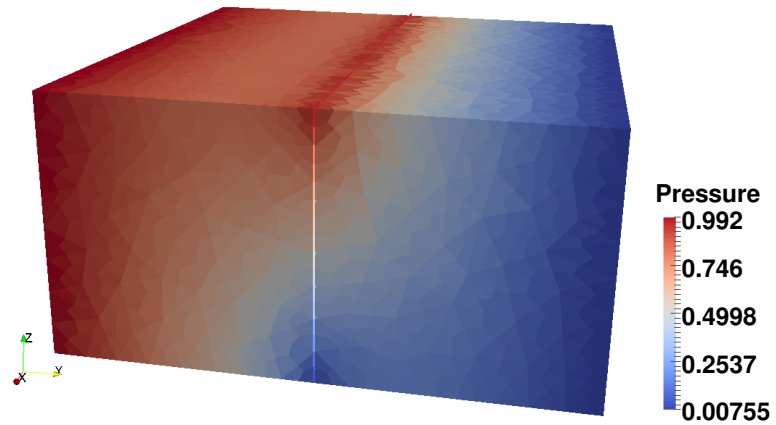
We have considered different values of the parameters k_τ^* and k_n^* with respect to the single fracture case. This is because $k_\tau^* = 1 = K$ (K is the bulk permeability) is not enough to reproduce a significant interaction between the bulk and the fracture flow in this case (see Remark 1.1 in Chapter 1). Therefore the threshold value of the parameters that changes the underlying physical phenomenon is case dependent and in the single fracture test we have imposed the pressure at the boundaries of the fracture and this, surely, made easier the interaction. So in this test we consider greater contrasts between the equivalent permeabilities k_τ^* and k_n^* and the bulk permeability.



(a) $k_\tau^* = 10^{-2}$ and $k_n^* = 1$



(b) $k_\tau^* = 10^{-2}$ and $k_n^* = 10^2$



(c) $k_\tau^* = 1$ and $k_n^* = 10^2$

Figure 5.7: Pressure field in the bulk domain for different values of k_τ^* and k_n^*.

	k_τ^*=1e-3, k_n^*=1e-1		k_τ^*=1e-3, k_n^*=1e3		k_τ^*=10, k_n^*=1e3	
	It	Res	It	Res	It	Res
Diag	13097	9.96e-07	X	X	X	X
BlockTr_D	37	8.96e-07	39	8.97e-07	39	8.61e-07
ILU_D	27	8.02e-07	29	7.46e-07	30	7.14e-07
HSS(1e-3)	102	9.55e-07	148	8.87e-07	259	9.85e-07

(a) Number of iterations for convergence and computed residual.

	k_τ^*=1e-3, k_n^*=1e-1	k_τ^*=1e-3, k_n^*=1e3	k_τ^*=10, k_n^*=1e3
UmfPack	21.53	20.10	19.39
Diag	274.48	X	X
BlockTr_D	0.59	0.62	0.62
ILU_D	0.51	0.54	0.55
HSS(1e-3)	2.80	4.33	8.18

(b) Computing time [sec].

Table 5.7: (a) Number of iterations to reach convergence and computed residual of GMRES for different preconditioners and by varying the model parameters. (b) Computing time to solve the system with UmfPack and GMRES for different preconditioners and by varying the model parameters. Diag means the diagonal part of the whole system; BlockTr_D and ILU_D refer to the block triangular and the block ILU preconditioners based on the diagonal part of M_c; HSS(α) refers to the Hermitian and skew-Hermitian splitting preconditioner with α coefficient. X means that the GMRES has not reached convergence in the prescribed number of iterations. For the GMRES we have set: MaxIt = 20000, tol = 10^{-6}, restart = 300. Remind that for HSS the FGMRES has been used. The model parameters are $k_\tau^* = k_\Gamma^\tau l_\Gamma$ and $k_n^* = k_\Gamma^n / l_\Gamma$.

The number of iterations to reach convergence and the computed residual are reported in (a) of Table 5.7, while the computing time is shown in (b) of Table 5.7. The BlockTr_D and ILU_D preconditioners show again to be not sensitive in terms of convergence to the parameters of the model. The HSS preconditioner, as in the single fracture case, presents a strong dependency on the parameters, especially passing to a conductive network of fractures. Note that in this latter case the HSS provides a moderate improvement of performance with respect to the direct solver UmfPack: 8.18 against 19.39 seconds; while we observe a great improvement with, for example, the ILU_D: 0.55 against 19.39 seconds. Finally, note the opposite behaviour of Diag with respect to the single fracture test: here the case that requires less iterations is the low permeable one.

Now we briefly comment about the results of the computations in the case of a network of barriers and of a conductive network. Let us start from the first case. The bulk pressure is reported in Figure 5.9. Note that the action of the fractures as barriers implies a strong discontinuity of the pressure across the fractures, which, physically, is simply due to a mass accumulation. The pressure inside the fractures is reported in Figure 5.10 and we can see pressure discontinuities also in the fractures. Finally the pressure inside the bulk and the network for a particular cut of the domain is reported in Figure 5.11. The bulk pressure in the case of a network of conductive fractures is reported in Figure 5.12. Note that in this case the distribution of pressure in bulk follows the main network direction, since

the fractures attract the bulk flow. In Figure 5.13 the pressure in the fractures is reported and in Figure 5.14 the pressure in bulk and fractures for a particular cut of the domain is reported.

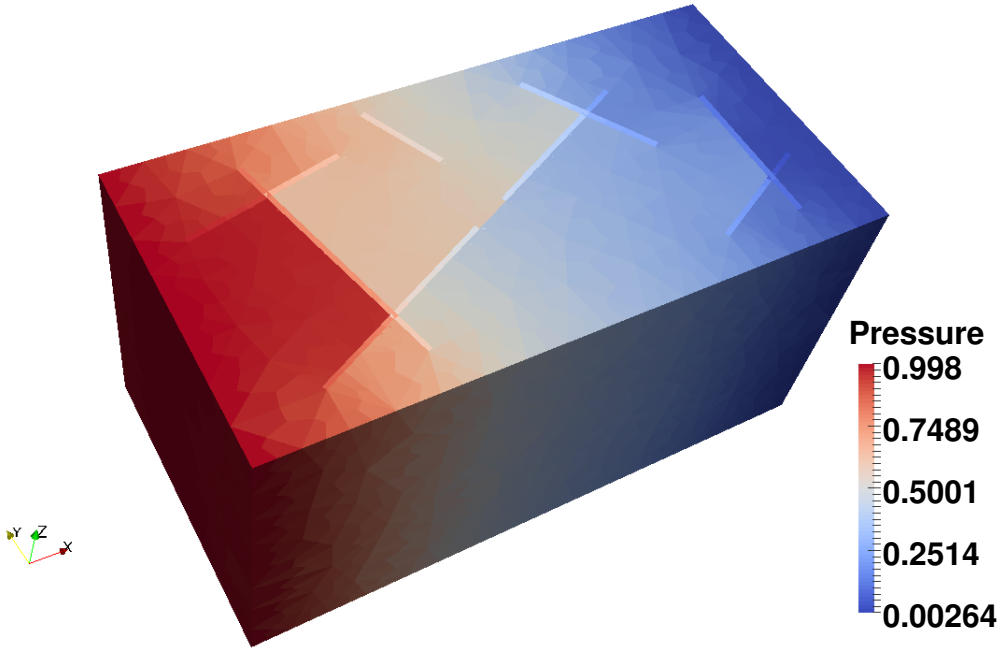


Figure 5.9: Pressure distribution in the porous medium - sealed network case: $k_\tau^* = 10^{-3}$, $k_n^* = 10^{-1}$.

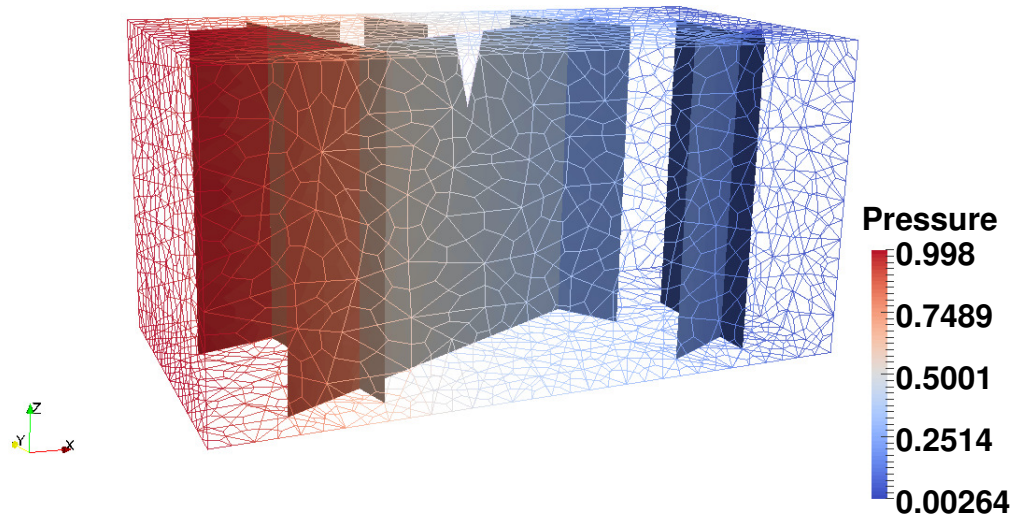


Figure 5.10: Pressure distribution in the fractures network - sealed network case: $k_\tau^* = 10^{-3}$, $k_n^* = 10^{-1}$.

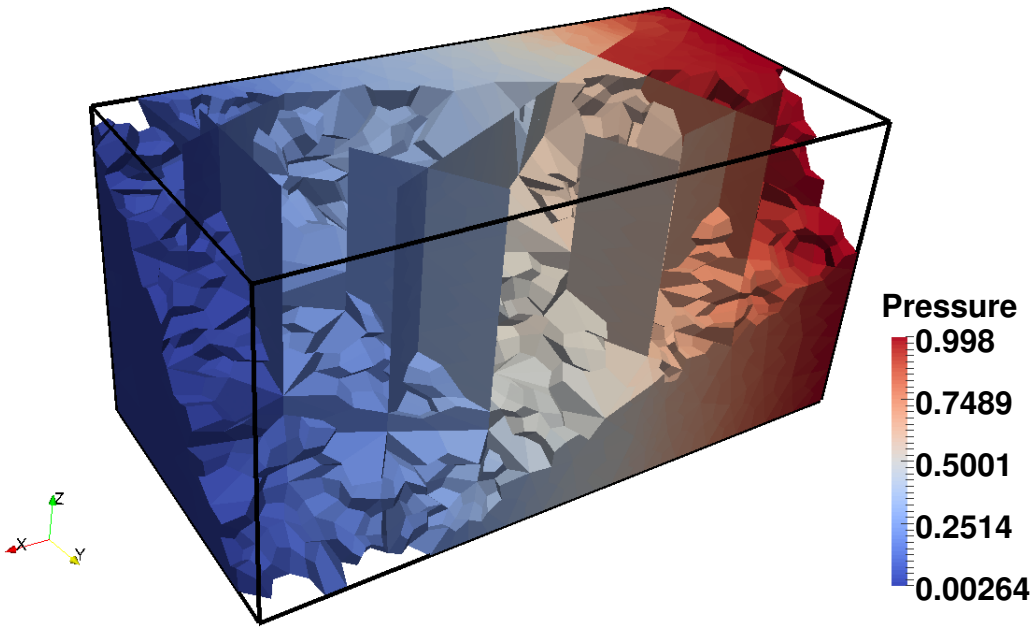


Figure 5.11: Pressure distribution inside the porous matrix and fractures on a cut of the domain - sealed network case: $k_\tau^* = 10^{-3}$, $k_n^* = 10^{-1}$.

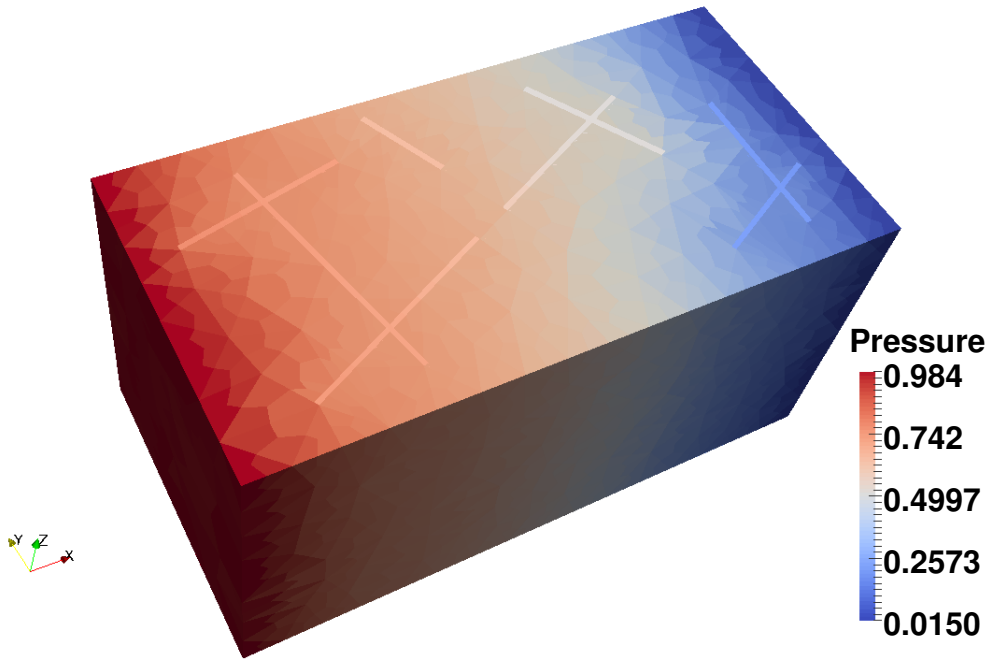


Figure 5.12: Pressure distribution in the porous medium - conductive network case: $k_\tau^* = 10$, $k_n^* = 10^3$.

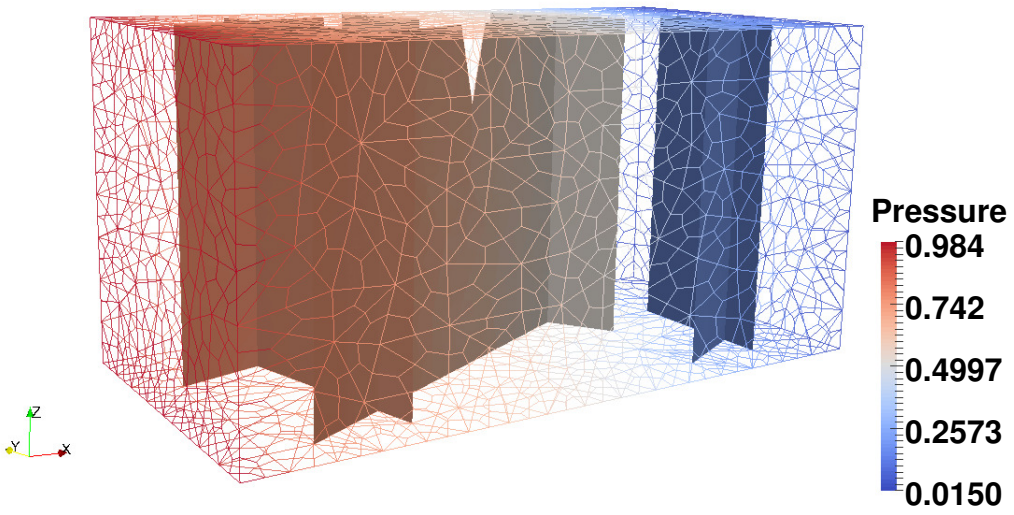


Figure 5.13: Pressure distribution in the fractures network - conductive network case: $k_\tau^* = 10$, $k_n^* = 10^3$.

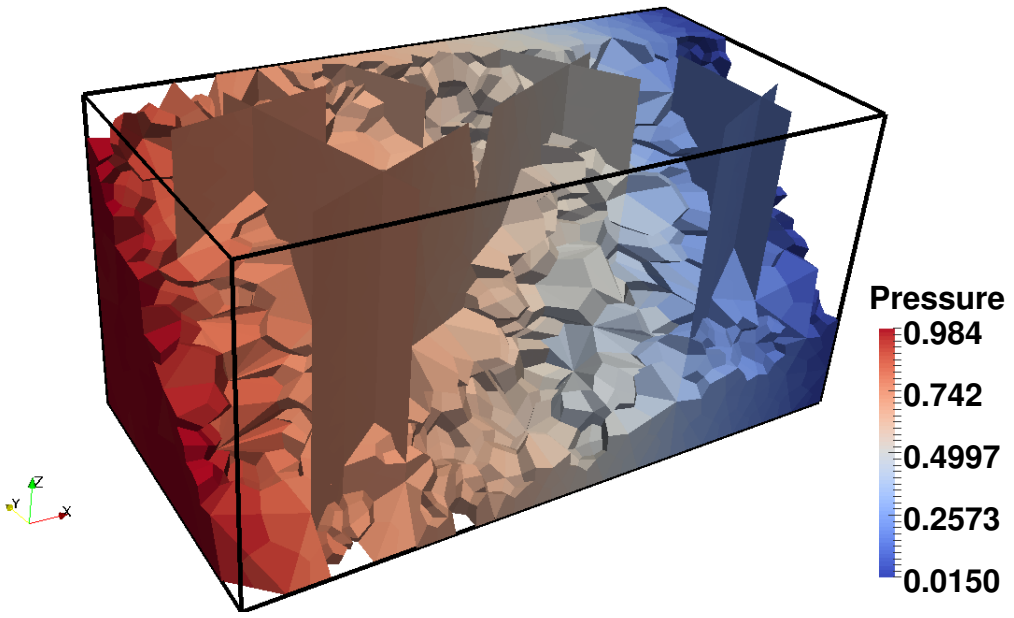


Figure 5.14: Pressure distribution inside the porous matrix and the fractures on a cut of the domain - conductive network case: $k_\tau^* = 10$, $k_n^* = 10^3$.

5.2.3 A second case of network of fractures

Now we consider another case of network of fractures. The bulk domain is $\Omega = (0,1) \times (0,1) \times (0,1)$ and the network of fractures Γ consists of nine fractures with several intersections; the bulk domain and the network are shown in Figure 5.15. With respect to the previous test, here we consider different boundary conditions and a non zero source/sink term. In particular we impose homogeneous Dirichlet boundary conditions on all $\partial\Omega$ and we assume a forcing term given by

$$f(x,y,z) = \begin{cases} 15 & \text{if } (x-0.3)^2 + (y-0.3)^2 + (z-0.8)^2 \leq 0.04 \\ -15 & \text{if } (x-0.5)^2 + (y-0.5)^2 + (z-0.3)^2 \leq 0.04 \\ 0 & \text{otherwise,} \end{cases}$$

which means one spherical source and one spherical sink in proximity of the network. A polyhedral mesh of diameter $h = 0.137$ is employed; the dimension of the system is 33156.

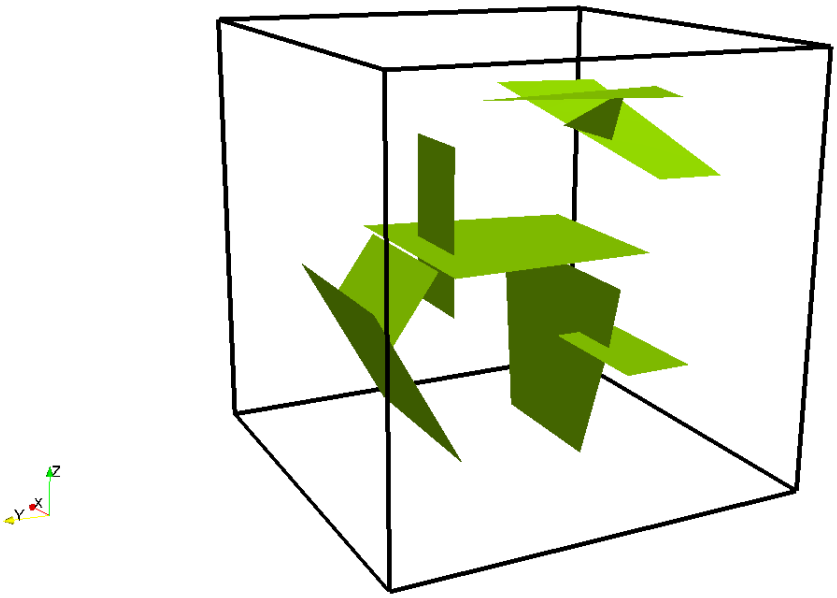


Figure 5.15: Porous medium with network of fractures.

The number of iterations to reach convergence and the computed residual are reported in (a) of Table 5.8, while the computing time is shown in (b) of Table 5.8. We confirm that, also in this case where more contrasts between the fracture equivalent permeabilities k_τ^* and k_n^* and the bulk permeability are considered, BlockTr_D and ILU_D are not sensitive, in terms of convergence of GMRES, on the variation of k_τ^* and k_n^*. Whereas the HSS preconditioner is strongly affected by the variation of the parameters. In particular the low permeable fractures case is the easiest to be handled iteratively with the HSS preconditioner, while the high permeable fractures case is the most difficult: we have more than doubling of iterations and computing time passing from one case to the other. Take into

	k_τ^*=1e-4, k_n^*=1e-2		k_τ^*=1e-4, k_n^*=1e4		k_τ^*=1e2, k_n^*=1e4	
	It	Res	It	Res	It	Res
Diag	14136	9.98e-07	X	X	X	X
BlockTr_D	41	9.97e-07	43	9.58e-07	43	9.05e-07
ILU_D	29	9.02e-07	32	8.77e-07	32	8.16e-07
HSS(1e-4)	247	9.64e-07	407	9.67e-07	565	9.89e-07

(a) Number of iterations for convergence and computed residual.

	k_τ^*=1e-4, k_n^*=1e-2	k_τ^*=1e-4, k_n^*=1e4	k_τ^*=1e2, k_n^*=1e4
UmfPack	36.53	36.29	35.24
Diag	242.97	X	X
BlockTr_D	0.44	0.46	0.46
ILU_D	0.36	0.39	0.39
HSS(1e-4)	7.42	13.09	18.98

(b) Computing time [sec].

Table 5.8: (a) Number of iterations to reach convergence and computed residual of GMRES for different preconditioners and by varying the model parameters. (b) Computing time to solve the system with UmfPack and GMRES for different preconditioners and by varying the model parameters. Diag means the diagonal part of the whole system; BlockTr_D and ILU_D refer to the block triangular and the block ILU preconditioners based on the diagonal part of M_c; HSS(α) refers to the Hermitian and skew-Hermitian splitting preconditioner with α coefficient. X means that the GMRES has not reached convergence in the prescribed number of iterations. For the GMRES we have set: MaxIt = 20000, tol = 10^{-6}, restart = 300. Remind that for HSS the FGMRES has been used. The model parameters are $k_\tau^* = k_\Gamma^\tau l_\Gamma$ and $k_n^* = k_\Gamma^n / l_\Gamma$.

account that the problem of conductive fractures is the most frequent and so the well-behaviour of the preconditioner in this case is important. Finally note that in this case the optimal value of α for the HSS is 10^{-4} and, as already pointed out, it does not depend on the fracture parameters.

Regarding the results of the simulations, in Figure 5.16, 5.17, 5.18 the pressure inside the porous matrix and the fractures is shown with proper cuts of the domain. Note the pressure jump across the fractures in the sealed network case. Whereas, for the other two cases, the pressure distribution is similar, obviously continuous, but we have greater positive and negative peaks of pressure in the case where the network has no influence. This makes sense because in the conductive case the pressure spreads out more inside the porous medium, thanks to the fractures that attract and carry most of the flow.

5.3 Test on the spectral properties of the system

In this section we present numerical experiments in order to investigate the spectral properties of the system. First, we study the spectral properties of the transmissibility matrix T to verify the assumption made in Section 3.5 of Chapter 3. Then, we estimate the condition number of the global matrix of the problem to verify the theoretical estimate presented in the same chapter. To estimate the

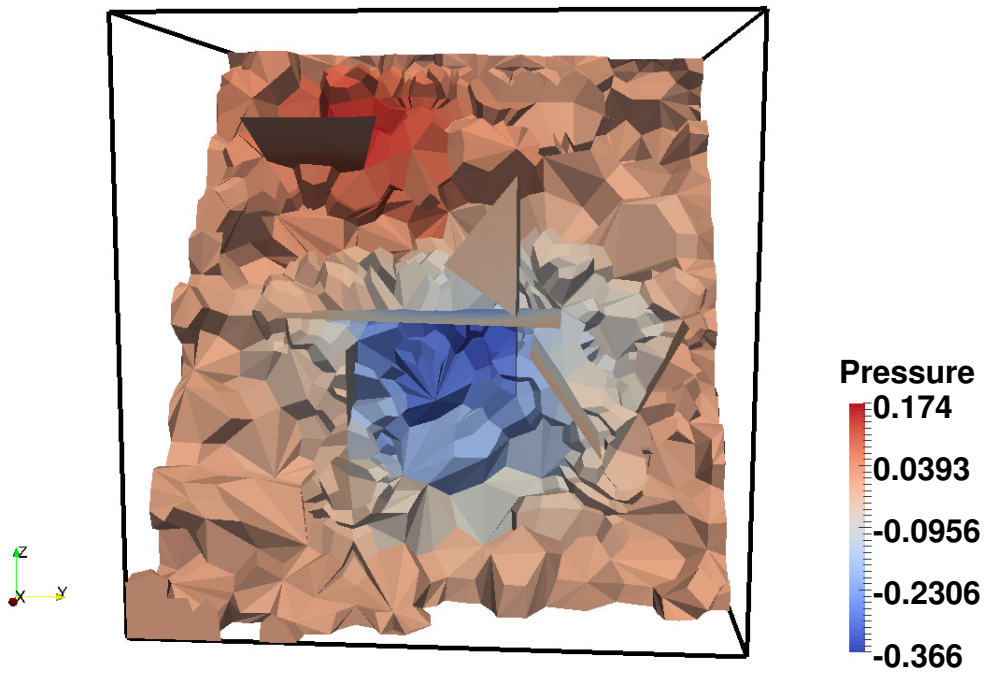


Figure 5.16: Pressure distribution inside the porous matrix and the fractures on a cut of the domain - sealed network case: $k_\tau^* = 10^{-4}$, $k_n^* = 10^{-2}$.

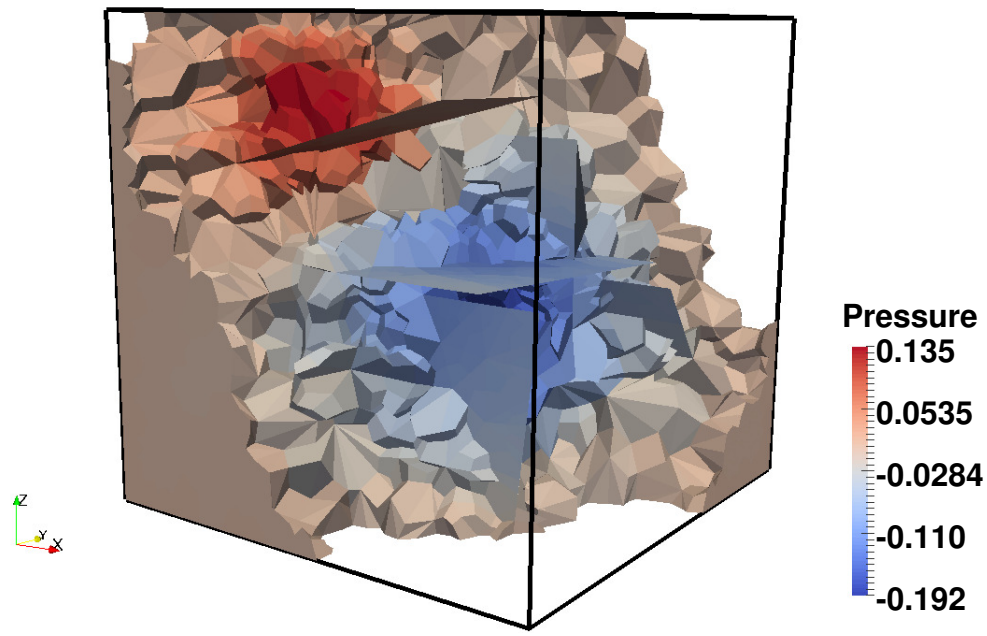


Figure 5.17: Pressure distribution inside the porous matrix and the fractures on a cut of the domain - no influence of the network: $k_\tau^* = 10^{-4}$, $k_n^* = 10^4$

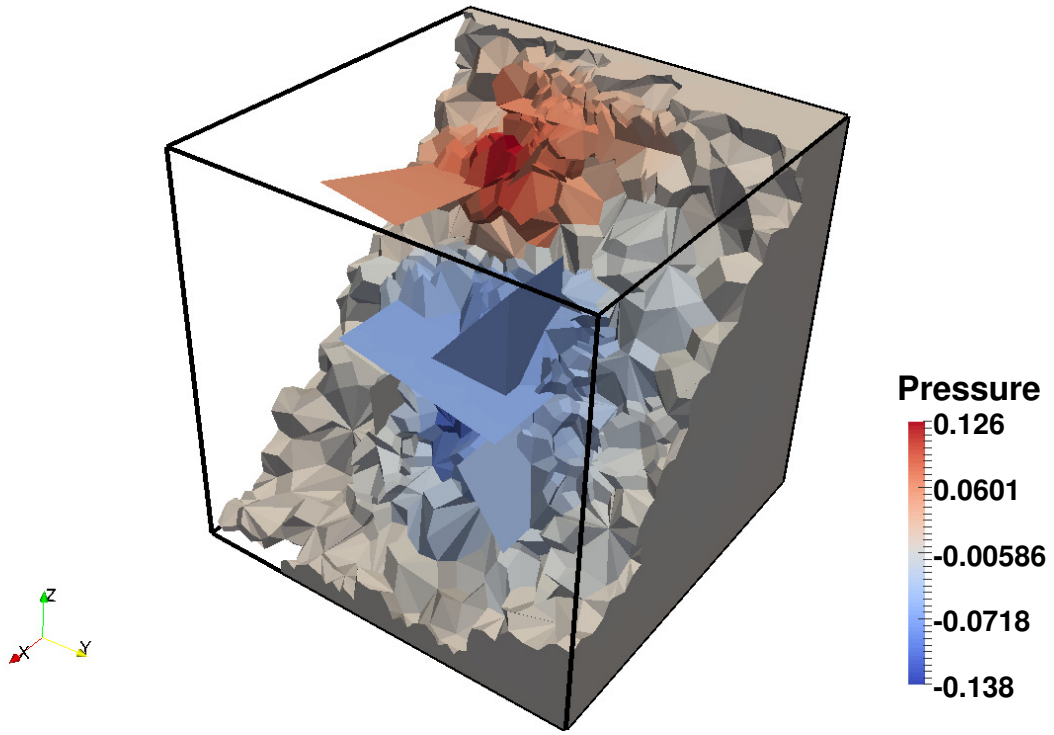


Figure 5.18: Pressure distribution inside the porous matrix and the fractures on a cut of the domain - conductive network case: $k_\tau^* = 10^2$, $k_n^* = 10^4$

maximum and minimum eigenvalues of the sparse matrices the function `eigs` of `Matlab` has been used.

We consider the same test case presented in Section 5.1 of this chapter, which is a single fracture configuration with full Dirichlet boundary conditions, the same hypotheses considered in the theoretical spectral analysis. We remind that the bulk permeability tensor is $\mathsf{K} = \mathsf{I}$, while for the fracture we consider $\mathsf{K}_\Gamma = k_f \mathsf{I}$, where k_f is a positive real number that may vary and enhances the permeability contrast between the porous matrix and the fracture. Moreover, different types of meshes are used in the numerical analysis: hexahedral (Cartesian), tetrahedral and polyhedral grids.

5.3.1 Spectral properties of the transmissibility matrix

For the spectral analysis of the linear system we have made the hypothesis that the spectral properties of the transmissibility matrix, which arises from the TPFA discretization of the fracture problem, are equivalent to those of a linear finite element scheme. More precisely, we have supposed the following spectral bound

$$M_* h^n \leq \lambda(\mathsf{T}) \leq M^* h^{n-2}, \tag{5.6}$$

where n is the dimension of the problem discretized by TPFA. From the above spectral estimate we have the bound for the condition number of T given by

$$K_2(\mathsf{T}) \leq \frac{M^*}{M_*} h^{-2}. \tag{5.7}$$

h	$\lambda_1(\mathsf{T})$	$\lambda_{N_\Gamma}(\mathsf{T})$	$K(\mathsf{T})$
0.25	8.00e-05	7.38e-06	10.84
0.125	8.00e-05	1.91e-06	41.96
0.0625	8.00e-05	4.81e-07	166.46
0.0313	8.00e-05	1.19e-07	672.27

(a) Hexahedral (Cartesian)

h	$\lambda_1(\mathsf{T})$	$\lambda_{N_\Gamma}(\mathsf{T})$	$K(\mathsf{T})$
1.118	9.74e-05	9.93e-06	9.81
0.563	1.02e-04	3.09e-06	33.10
0.314	1.02e-04	7.23e-07	141.35
0.144	1.02e-04	1.69e-07	603.55

(b) Tetrahedral

h	$\lambda_1(\mathsf{T})$	$\lambda_{N_\Gamma}(\mathsf{T})$	$K(\mathsf{T})$
0.559	9.10e-05	3.61e-06	25.17
0.265	9.75e-05	1.03e-06	95.12
0.137	1.00e-04	2.36e-07	424.38
0.067	1.03e-04	0.53e-07	1943.40

(c) Polyhedral

Table 5.9: Spectral properties of matrix T for different types of grids.

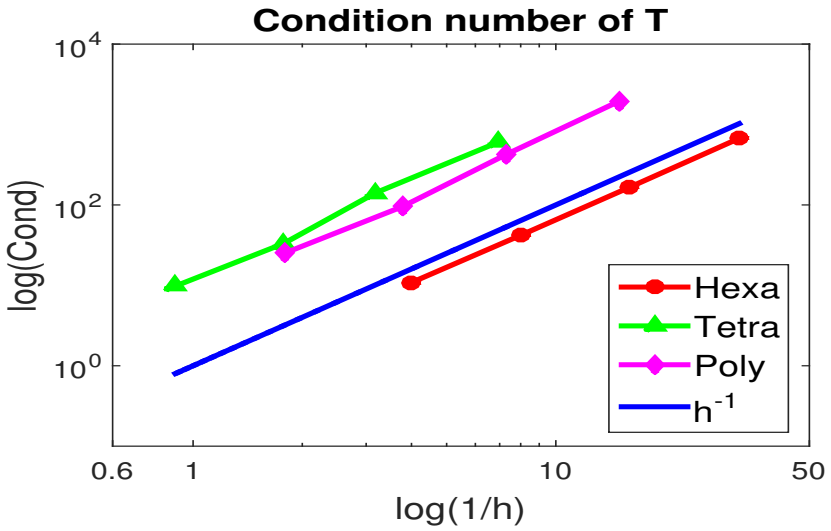


Figure 5.19: Condition number of T as a function of the mesh size (loglog) for different types of grids. Hexa, Tetra, Poly stand for hexahedral (Cartesian), tetrahedral, polyhedral grids, respectively.

In this test we consider $k_f = 10^{-3}$. In Table 5.9 we have reported the estimated maximum and minimum eigenvalues and the condition number of T, which has been computed as $K_2(\mathsf{T}) = \lambda_1(\mathsf{T})/\lambda_{N_\Gamma}(\mathsf{T})$ because T is symmetric and positive definite. The dimension of the fracture problem is $n = 2$ and the theoretical bound (5.6) provides a constant upper bound. As can be seen in the table, $\lambda_1(\mathsf{T})$ is constant

h	k_f=1e-3	k_f=1	k_f=1e+3
0.25	4.31e+01	2.77e+01	1.02e+04
0.125	8.54e+01	8.79e+01	8.20e+04
0.0625	1.69e+02	6.58e+02	6.55e+05
0.0313	3.41e+02	2.64e+03	5.32e+06

(a) Hexahedral (Cartesian)

h	k_f=1e-3	k_f=1	k_f=1e+3
1.118	3.38e+02	3.38e+02	3.74e+04
0.563	7.04e+02	6.91e+02	3.14e+05
0.314	1.35e+03	2.54e+03	2.51e+06
0.144	2.76e+03	1.01e+04	2.21e+07

(b) Tetrahedral

h	k_f=1e-3	k_f=1	k_f=1e+3
0.559	1.12e+03	1.11e+03	2.09e+05
0.265	4.17e+03	4.18e+03	1.78e+06
0.137	8.69e+03	1.47e+04	1.47e+07
0.067	2.52e+04	5.85e+04	1.15e+08

(c) Polyhedral

Table 5.10: Condition number of $\mathcal{A}$ by varying the mesh size and the fracture permeability for different types of grids.

for all the three types of grid, while in Figure 5.19 the dependence h^{-2} of the condition number can be observed in all the cases. Equivalent results have been observed by varying the fracture permeability k_f. Therefore we can conclude that our assumption on T is validated numerically and the spectral properties of the TPFA version of finite volumes are equivalent to those of linear finite elements.

5.3.2 Numerical estimate of the condition number

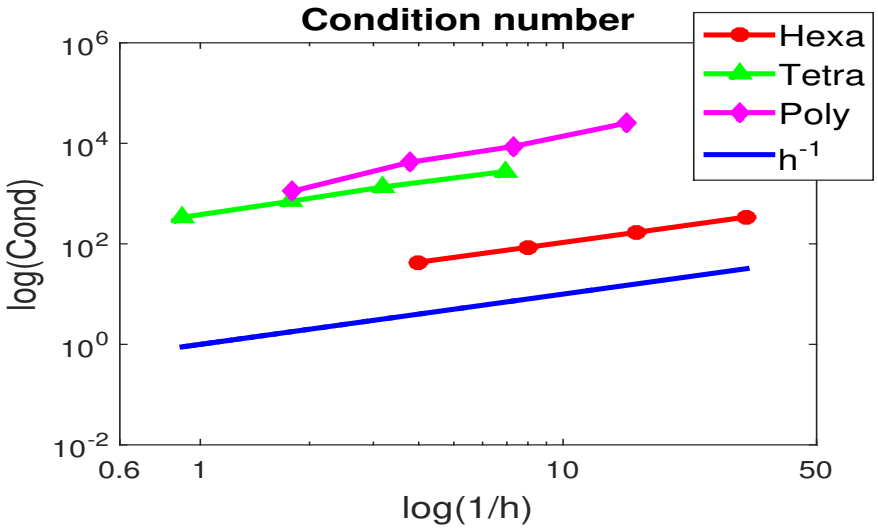
We focus now on the numerical validation of the estimate of the condition number for the matrix of the system $\mathcal{A}$ stated in Theorem 3.5 in Section 3.5 of Chapter 3. In particular, the estimate is

$$K_2(\mathcal{A}) \leq \frac{k_1}{k_2} h^{-4}. \tag{5.8}$$

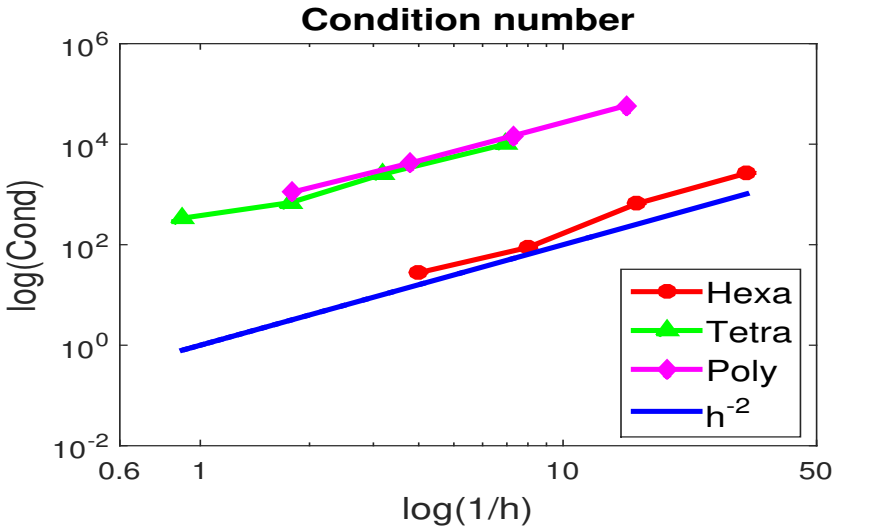
Thanks to the symmetry of $\mathcal{A}$, in the tests the condition number has been computed as

$$K_2(\mathcal{A}) = \frac{\max_{\lambda_i} |\lambda_i(\mathcal{A})|}{\min_{\lambda_i} |\lambda_i(\mathcal{A})|}.$$

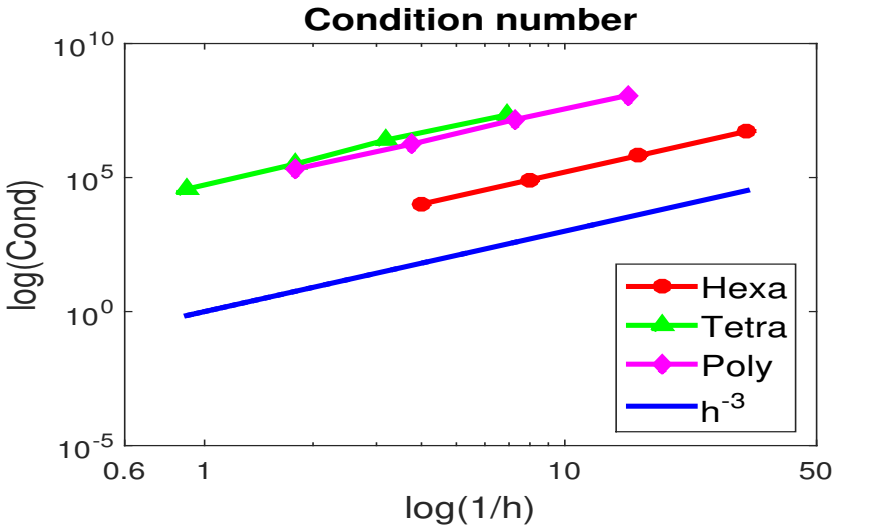
In Figure 5.20 the condition number as a function of the mesh size is reported for different types of meshes and by varying k_f. We see that the h-dependence changes with the fracture permeability: h^{-1} for $k_f = 10^{-3}$, h^{-2} for $k_f = 1$, h^{-3} for $k_f = 10^3$. This is probably due to the fact that the meshes are not enough refined. Indeed the theoretical estimate (5.8) is asymptotic and with a mesh size not sufficiently



(a) $k_f = 10^{-3}$



(b) $k_f = 1$



(c) $k_f = 10^3$

Figure 5.20: Condition number of $\mathcal{A}$ as a function of the mesh size (loglog) by varying the fracture permeability $k_f = 10^{-3}, 1, 10^3$. Hexa, Tetra, Poly stand for hexahedral (Cartesian), tetrahedral, polyhedral grids, respectively.

small probably the constants in the intervals I^- and I^+ (see (3.54) and (3.55) in the proof of the estimate) have a major impact changing the h-dependence of the condition number. These constants depend on the fracture permeability and may change a lot with k_f. In any case the theoretical estimate in (5.8) is always fulfilled but from the results we can conjecture that it is not sharp, because also in case of a high permeable fracture it provides an overestimation of one order.

In Table 5.10 we have reported the condition numbers computed in the tests. Note that in general with hexahedral meshes the condition numbers are significantly lower and this makes because the hexahedral case involves structured grids. We also note that for all the grid types, increasing the fracture permeability passing to a conductive fracture ($k_f = 10^3$), the condition number increases a lot (3 orders of magnitude in most cases).

The spectral analysis requires surely further developments. Note that in our work we have not considered the particular form of $\widetilde{\mathsf{T}}$ (see (2.36)) that has three zero blocks. Moreover, the dimension of the block (1,1) coincides with N_{P} that scales as h^{-3}, while the block (2,2) has dimension N_Γ that scales as h^{-2}. This may be taken into account in the spectral analysis and may lead to a better estimate. Another point that is not completely clear is the relation between an "asymptotic" estimate and the fracture model. Indeed, if $h \to 0^+$, we would have bulk cells of a diameter smaller than the aperture l_Γ and the model would lose its consistency. Also this aspect requires a deeper investigation.

Conclusions

In this thesis we have implemented a mimetic code for 3D flows in fractured porous media, with three preconditioners for the discrete problem, which have been widely tested numerically. Moreover we have studied the conditioning of the linear system arising from the discretization, from both a theoretical and a numerical point of view.

A mixed formulation, discretized with MFD, is employed in the porous matrix, while a primal formulation, discretized with FV-TPFA, is adopted in fractures. The MFD are designed for general polyhedral meshes and are robust with respect to grid anisotropy. This is very important in numerical approximation of flows in fractured porous media, where mesh generation involves several difficulties because of the fractures conformity constraint. The choice of TPFA in fractures is due especially to the easier handling of intersections. In this thesis we have reviewed, in the case of a single fracture, the mathematical model and a well-posedness result [7]. Then we have described the employed numerical methods, the mimetic framework and the TPFA scheme, and we have reviewed a non-singularity result [7] for the linear system resulting from the discretization.

Direct methods scale poorly with system dimension and in 3D computations the use of a suitable preconditioned iterative solver becomes mandatory. The linear system governing our discrete problem can be formulated as a generalized saddle point problem, leading the way to several preconditioning techniques [20], and it is solved with GMRES. After an introduction to Krylov subspace methods and preconditioning, we have described three possible techniques: a block triangular preconditioner, a block ILU preconditioner and an Hermitian/skew-Hermitian splitting preconditioner. The usage of MFD makes it difficult to find suitable approximations of the modified inner product matrix M_c and of the Schur Complement S for the block triangular and block ILU preconditioners, because no optimal choices are available in literature. We have considered simple diagonal approximations of M_c, the lumped matrix and the diagonal part, and we have approximated S accordingly. Then we have presented a theoretical estimate of the condition number of the system with its proof, which is a generalization of the one considered in [51]. This is an important result of the thesis and it allows to characterize the conditioning of the system, which is significant for preconditioning.

Next, we have described the implementative features of the C++ code. The main programming effort in this thesis has concerned the development of a well-designed class structure for the MFD method and for the iterative solvers and preconditioners. We mention the class architecture for the assembly of the linear system, which is more complex if compared with code works in [60] and [61], and

allows a more efficient construction of the matrices. Another interesting feature of the code is the capability of handling the storage format of the matrix of the system at run time, depending on the type of solver selected by the user: with a direct solver the monolithic matrix of the system is stored, while in case of an iterative solver only the single blocks of the saddle point matrix are stored, with resulting savings of memory and cheaper construction of the preconditioners.

Now we summarize the numerical results. First, in a case with a theoretical solution, the pressure errors, both in bulk and fracture, and the velocity errors have been computed for different mesh types. The results confirm the theoretical convergence orders. We stress the lower convergence order of the fracture pressure that arises in some cases where the K-orthogonality constraint is not fulfilled.

Next, we have tested numerically the performances and the behaviour of the preconditioners. The block triangular and block ILU preconditioners have given very good results if applied with the diagonal of M_c; instead with the lumping of M_c they have turned out to be completely inadequate. They have proved to be slightly affected from the mesh size and independent from the fracture model parameters. We point out that they always converge in a few iterations (less then 45) providing a radical time-savings with respect to direct computations already for medium size problems. For what concerns the HSS preconditioner, in general it provides good computing time performances, but it shows a significant dependence both from the mesh size and the fracture parameters, in particular the conductive fractures case requires more iterations with a significant worsening of computing time.

Moreover, the spectral properties of the system matrix have been analysed numerically. The results in terms of condition number are strongly affected by the fracture permeability, but the theoretical bound for the condition number is respected anyway. However, from the numerical experiments we can conjecture that the theoretical estimate is not sharp. Therefore, a more in depth study of the spectral properties of the system matrix is necessary.

Now let us mention some possible developments of this thesis. An important extension in a 3D framework is the implementation of a parallel solver, which allows to enhance the computational efficiency. The presence of fractures turns out in a natural partition of the domain, so that the problem can be suitably formulated for parallel resolution with Domain Decomposition methods [2]. In cases of complex networks of fractures, finding a domain partition may be an hard task, another possible approach is working on the connectivity graph of the matrix of the system to obtain such a partition [33]. We stress that block preconditioners developed in this thesis can be extended to parallel solvers [46].

An useful extension is the implementation of a reconstruction operator for the velocity in order to make possible its visualization. Theoretical reconstruction operators for polyhedral meshes exist [14], but their practical implementation is again a subject of study. Another remarkable development consists of considering a proper 1D model for the fracture flow along the intersections; a cell-centred FV numerical scheme can be considered also along fractures intersections.

From the point of view of the numerical model, a more accurate scheme can be considered. For instance, the mimetic formulation can be extended also in fractures, in such a case the pressure and flux continuity across intersections have to be

enforced in the discrete formulation [60, 34]. Also the implementation of Virtual Element Method in bulk, leading the way to higher order approximations, would be a notable work. Some steps in this direction have been already made [15, 16, 35].

Considering more complex physical phenomena in fractured porous media is another interesting development. In this sense a possible extension consists of studying the transient of a compressible flow, in such a case not to neglect the gravitational effect would change further the equations; also extending the model to handle two-phase flows would be remarkable. A first step in this direction may be to include a model for passive transport, propaedeutic to the implementation of the model for saturation necessary in two-phase flows. In that respect, a proper interface condition between bulk and fractures for transport of a passive scalar has been studied in [27]. Finally, we mention that also the porous matrix can be considered compressible, coupling the fracture flow problem with the poroelasticity equations; this is probably one of the most challenging extensions.

In conclusion, we may state that the work carried out in this thesis is a fundamental step for investigations in several directions, all very relevant to real life problems.

Bibliography

[1] O. Al-Hinai, S. Srinivasan, M. F. Wheeler, *Mimetic finite differences for flow in fractures from microseismic data*, In SPE Reservoir Simulation Symposium, 23-25 February, Houston, Texas, USA. Society of Petroleum Engineers, 2015.

[2] C. Alboin, J. Jaffré, J. E. Roberts, C. Serres, *Domain decomposition for flow in porous media with fractures*, In Proceedings of the 11th International Conference on Domain Decomposition Methods in Greenwich, England, 1999.

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

[4] P. Angot, F. Boyer, F. Hubert, *Asymptotic and numerical modelling of flows in fractured porous media*, ESAIM: Mathematical Modelling and Numerical Analysis, 43(2):239-275, 2009.

[5] P. F. Antonietti, L. Beirão da Veiga, N. Bigoni, M. Verani, *Mimetic finite differences for nonlinear and control problems*, Math. Models Methods Appl. Sci., 24(8):1457–1493, 2014.

[6] P. F. Antonietti, N. Bigoni, M. Verani, *Mimetic discretizations of elliptic control problems*, J. Sci. Comput., 56(1):14–27, 2013.

[7] 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, 50(3):809-832, 2016.

[8] P. F. Antonietti, M.Verani, L. Zikatanov, *A two-level method for mimetic finite difference discretizations of elliptic problems*, Computers & Mathematics with Applications, 70(11):2674-2687, 2015.

[9] T. Arbogast, J. Douglas, Jr, U. Hornung, *Derivation of the double porosity model of single phase flow via homogenization theory*, SIAM Journal on Mathematical Analysis, 21(4):823–836, 1990.

[10] Z.-Z. Bai, G. H. Golub, M. K. Ng, *Hermitian and Skew-Hermitian Splitting Methods for Non-Hermitian Positive Definite Linear Systems*, SIAM J. Matrix Anal. & Appl., 24(3):603–626, 2003.

[11] J. Bear, C.-F. Tsang, and G. de Marsily, *Flow and contaminant transport in fractured rock*, Academic Press, San Diego, 1993.

[12] L. Beirão da Veiga, F. Brezzi, A. Cangiani, G. Manzini, L. D. Marini, A. Russo, *Basic principles of virtual element methods*, Math. Models Methods Appl. Sci., 23(1):199–214, 2013.

[13] L. Beirão da Veiga, K. Lipnikov, G. Manzini, *Arbitrary-order nodal mimetic discretizations of elliptic problems on polygonal meshes*, SIAM Journal on Numerical Analysis, 49(5):1737–1760, 2011.

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

[15] M. F. Benedetto, S. Berrone, S. Pieraccini, S. Scialò, *The virtual element method for discrete fracture network simulations*, Comput. Methods Appl. Mech. Engrg., 280:135–156, 2014.

[16] M. F. Benedetto, S. Berrone, S. Scialò, *A globally conforming method for solving flow in discrete fracture networks using the virtual element method*, Finite Elements in Analysis and Design, 109:23– 36, 2016.

[17] M. Benzi, *Preconditioning Techniques for Large Linear Systems: A Survey*, Journal of Computational Physics, 182(2):418-477, 2002.

[18] M. Benzi, M. J. Gander, G. H. Golub, *Optimization of the Hermitian and skew-Hermitian splitting iteration for saddle-point problems*, BIT Numerical Mathematics, 43(5):881-900, 2003.

[19] M. Benzi, G. H. Golub, *A preconditioner for generalized saddle point problem*, SIAM J. Matrix Anal. & Appl., 26(1):20–41, 2004.

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

[21] S. Berrone, S. Pieraccini, S. Scialò, *A PDE-constrained optimization formulation for discrete fracture network flows*, SIAM J. Sci. Comput., 35(2):B487–B510, 2013.

[22] S. Berrone, S. Pieraccini, S. Scialò, F. Vicini, *A parallel solver for large scale DFN flow simulations*, SIAM J. Sci. Comput., 37(3):C285–C306, 2015.

[23] F. Brezzi, A. Buffa, *Innovative mimetic discretizations for electromagnetic problems.*, J. Comput. Appl. Math., 234(6):1980–1987, 2010.

[24] F. Brezzi, A. Buffa, K. Lipnikov, *Mimetic finite differences for elliptic problems*, ESAIM: Mathematical Modelling and Numerical Analysis, 43(02):277–295, 2009.

[25] F. Brezzi, A. Buffa, G. Manzini, *Mimetic scalar products of discrete differential forms*, J. Comput. Phys., 257(part B):1228–1259, 2014.

[26] 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.

[27] F. Chave, D. A. Di Pietro, L. Formaggia, *A Hybrid High-Order Method for Darcy Flows in Fractured Porous Media*, SIAM J. Sci. Comput., 40(2): A1063-A1094, 2017.

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

[29] T. A. Davis, S. Rajamanickam, W. M. Sid-Lakhdar, *A survey of direct methods for sparse linear systems*, Acta Numerica, 25:383-566, 2016.

[30] 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.

[31] 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.

[32] H. Elman, D. Silvester, A. Wathen, *Finite Elements and Fast Iterative Solvers. With applications in incompressible fluid dynamics*, OUP Oxford, 2005.

[33] M. C. Ferris, J. D. Horn, *Partitioning mathematical programs for parallel solution*, Mathematical Programming, 80(1):35-61, 1998.

[34] L. Formaggia, A. Scotti, F. Sottocasa, *Analysis of a mimetic finite difference approximation of flows in fractured porous media*, ESAIM: Mathematical Modelling and Numerical Analysis, Received: 17 November 2016, Accepted: 05 May 2017.

[35] A. Fumagalli, E. Keilegavlen, *Dual Virtual Element Method for Discrete Fractures Network*, SIAM J. Sci. Comput., 40(1):B228-B258, 2016.

[36] A. Fumagalli, A. Scotti, *A reduced model for flow and transport in fractured porous media with non-matching grids*, In Proceedings of ENUMATH 2011, the 9 th European Conference on Numerical Mathematics and Advanced Applications, Springer-Verlag, 2012.

[37] A. Fumagalli, A. Scotti, *A numerical method for two-phase flow in fractured porous media with non-matching grids*, Advances in Water Resources, 62, Part C(0):454–464, 2013. Computational Methods in Geologic CO2 Sequestration.

[38] V. Girault, M. F. Wheeler, B. Ganis, M. E. Mear, *A lubrication fracture model in a poro-elastic medium*, Mathematical Models and Methods in Applied Sciences, 25(4):1-59, 2014.

[39] A. Greenbaum, L. Gurvits, *Max-min properties of matrix factor norms*, SIAM J. Sci. Comput., 15(2):348–358, 1994.

[40] A. Grillo, D. Logashenko, S. Stichel, G. Wittum, *Forchheimer's correction in modelling flow and transport in fractured porous media*, Computing and Visualization in Science, 15(4):169-190, 2012.

[41] N. I. M. Gould, V. Simoncini, *Spectral analysis of saddle point matrices with indefinite leading blocks*, SIAM J. Matrix Anal., 31(3):1152-1171, 2009.

[42] T.-Z. Huang, S.-L. Wu, C.-X. Li, *The spectral properties of the Hermitian and skew-Hermitian splitting preconditioner for generalized saddle point problems*, Journal of Computational and Applied Mathematics, 229(1):37-46, 2009.

[43] J. Jaffré, M. Mnejja, J. E. Roberts, *A discrete fracture model for two-phase flow with matrix-fracture interaction*, Procedia Computer Science, 4:967–973, 2011.

[44] T. Kalbacher, R. Mettier, C. McDermott, W. Wang, G. Kosakowski, T. Taniguchi, O. Kolditz, *Geometric modelling and object-oriented software concepts applied to a heterogeneous fractured network from the Grimsel rock laboratory*, Comput. Geosci., 11(1): 9-26, 2007.

[45] M. Karimi-Fard, L. J. Durlofsky, K. Aziz, *An efficient discrete-fracture model applicable for general-purpose reservoir simulators*, SPE Journal, 9(02):227–236, 2004.

[46] A. Klawonn, L. F. Pavarino, *A comparison of overlapping Schwarz methods and block preconditioners for saddle point problems*, Numerical Linear Algebra with applications, 7(1):1:25, 2000.

[47] Y. Kuznetsov, K. Lipnikov, M. Shashkov, *The mimetic finite difference method on polygonal meshes for diffusion-type problems*, Comput. Geosci., 8(4):9–26, 2007.

[48] V. Lenti, C. Fidelibus, *A BEM solution of steady-state flow problems in discrete fracture networks with minimization of core storage*, Comput. Geosci., 29(9): 1183–1190, 2003.

[49] S. Li, Z. Xu, G. Ma, W. Yang, *An adaptive mesh refinement method for a medium with discrete fracture network: the enriched Persson's method*, Finite Elem. Anal. Des., 86(0): 41–50, 2014.

[50] S. B. Lippman, J. Lajoie, B. E. Moo, *C++ Primer*, Fifth Edition, Addison-Wesley, 2015.

[51] L. Lopez, G. Vacca, *Spectral properties and conservation laws in Mimetic Finite Difference methods for PDEs* Journal of Computational and Applied Mathematics, 292:760-784, 2016.

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

[53] H. Mustapha, *A Gabriel-Delaunay triangulation of 2D complex fractured media for multiphase flow simulations*, Computational Geosciences, 18(6):989–1008, 2014.

[54] C. E. Powell, D. Silvester, *Optimal preconditioning for Raviart–Thomas mixed formulation of second-order elliptic problems*, SIAM J. Matrix Anal. Appl., 25(3):718–738, 2003.

[55] A. Quarteroni, *Numerical models for differential problems*, Second edition, Springer, 2014.

[56] A. Quarteroni, R. Sacco, F. Saleri, *Numerical Mathematics*, Second edition, Springer, 2007.

[57] Y. Saad, *A flexible inner-outer preconditioned GMRES algorithm*, SIAM J. Sci. Comput., 14(2):461–469, 1993.

[58] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Second edition, SIAM, 2003.

[59] S. Salsa, *Partial Differential Equations in Action. From Modelling to Theory* Second edition, Springer, 2015.

[60] F. Sottocasa, *Formulazione mista per flussi in mezzi porosi fratturati: approssimazione con le Differenze Finite Mimetiche*, Master thesis, Politecnico di Milano, Milan, 2015.

[61] N. Verzotti, *Flusso in un mezzo poroso fratturato: approssimazione numerica tramite le Differenze Finite Mimetiche*, Master thesis, Politecnico di Milano, Milan, 2014.

[62] S. Zonca, *A prototypal 3D Mimetic Finite Difference code for the Darcy equation on polyhedral meshes*, Mox internal report, 2015, http://mox.polimi.it.

[63] `CGAL`, https://www.cgal.org.

[64] `Eigen`, https://eigen.tuxfamily.org.

[65] `GetPot`, https://getpot.sourceforge.net.

[66] `IML++`, https://math.nist.gov/iml++/.

[67] `Matlab`, https://it.mathworks.com/products/matlab.html.

[68] `OpenFOAM`, https://www.openfoam.com/.

[69] `Paraview`, https://www.paraview.org/.

[70] `TetGen`, https://wias-berlin.de/software/tetgen/.